Invasive computing is a paradigm for designing and programming future parallel computing systems. For systems with 1,000 or more cores on a chip, resource-aware programming is of utmost importance to obtain high utilisation as well as computational, energy and power efficiency. Invasive computing provides a programmer explicit handles to specify and argue about resource requirements desired or required in different phases of execution: In an invade phase, an application asks the operating system to allocate a set of processor, memory and communication resources to be claimed. In a subsequent infect phase, the parallel workload is spread and executed on the obtained claim of resources. Finally, if the degree of parallelism should be lower again, a retreat operation frees the claim again, and the application resumes a sequential execution. To support this idea of self-adaptive and resource-aware programming, not only new programming concepts, languages, compilers, and operating systems were needed to be developed, but also revolutionary architectural changes in the design of MPSoCs (multiprocessor systems-on-a-chip) to efficiently support invasion, infection, and retreat operations. This book gives a comprehensive overview of all aspects of invasive computing.
Jürgen Teich, Jörg Henkel, Andreas Herkersdorf (Eds.)

Invasive Computing
Preface

This book gives a comprehensive introduction to the research field of invasive computing. At the same time, it summarises the results of 12 years of interdisciplinary research in this area on fundamentals, programming languages, manycore architectures, compilation techniques, operating systems, and applications performed within the Transregional Collaborative Research Centre 89 “Invasive Computing” (InvasIC) of the German Research Foundation (DFG).

We would like to thank all contributors to this work, including principal investigators, doctoral and post-doctoral researchers, the InvasIC management team as well as our universities for their steady support throughout the years. Special thanks to Frank Hannig for formatting and compiling this book and for many hours of editorial work. Thanks also to FAU University Press for revising, commenting, and providing great support to realise this work in a timely manner and for making it possible to offer an open access version to the public.

Erlangen, Karlsruhe, and Munich, Germany

The Editors
Contents

1 Introduction to Invasive Computing 1
Jürgen Teich, Jörg Henkel, Andreas Herkersdorf
1.1 What is Invasive Computing?  ........................................ 2
1.2 Long-Term Research Goals and Vision ............................ 6
1.3 Achievements in First and Second Funding Phases (2010–2018) 13
1.4 Assessment of Project Structure ................................. 41
1.5 Research Focus in the Third Funding Phase (2018–2022) .......... 43
  1.5.1 Goals and Strategy ........................................... 43
  1.5.2 Research Focus per Project Area ............................. 44
  1.5.3 InvasIC Validation and Demonstrator Road Map ............ 53
  1.5.4 Working Groups – Developing Cross-Project Synergies ....... 55

A: Fundamentals, Language and Algorithm Research 69

2 Basics of Invasive Computing 69
Jürgen Teich, Khalil Esper, Joachim Falk, Behnaz Pourmohseni, Tobias Schwarzer,
Stefan Wildermann
  2.1 Run-Time Requirement Enforcement (RRE) ....................... 69
  2.2 Taxonomy of Run-Time Requirement Enforcers .................. 71
  2.3 Application Characterisation and Enforcement Actions .......... 73
  2.4 Power, Latency, and Energy Model .............................. 74
  2.5 Energy-Minimised Timing Enforcement .......................... 77
  2.6 Distributed Enforcement ....................................... 78
  2.7 Feedback-based Enforcement Strategies ......................... 81
    2.7.1 Formal Definitions ......................................... 82
    2.7.2 Verification Goals ......................................... 85
    2.7.3 Property Specification in PRISM ............................ 87
    2.7.4 Verification Results ....................................... 88
  2.8 Conclusions and Acknowledgement ................................ 92

3 Characterisation and Analysis of Invasive Algorithmic Patterns 97
Michael Bader, Stefan Wildermann, Michael Glaß, Alexander Pöppl, Behnaz
Pourmohseni, Tobias Schwarzer, Jan Spieck, Mario Wille
  3.1 Introduction ...................................................... 97
  3.2 Algorithmic Patterns ............................................ 98
    3.2.1 The Shallow Water Equations ............................... 98
    3.2.2 SWE-X10 .................................................. 100
3.3 Hybrid Application Mapping ........................................ 105
  3.3.1 Complexity Reduction ........................................ 107
  3.3.2 Composability ............................................... 108
  3.3.3 Mapping Reconfiguration and Task Migration .............. 109
  3.3.4 Scenario-aware HAM ......................................... 109
3.4 Conclusions and Acknowledgements ............................. 119

4 Scheduling Invasive Multicore Programs Under Uncertainty 123
Nicole Megow, Alexander Lindermayr, Bertrand Simon
  4.1 Introduction .................................................. 123
  4.2 Uncertain Execution Times .................................... 124
    4.2.1 Learning-Augmented Algorithm Design .................. 125
    4.2.2 Prediction Model and Error Definition ................. 127
    4.2.3 The Preferential Time Sharing Framework ............... 129
    4.2.4 Experimental Evaluation .................................. 131
  4.3 Uncertain Task Graphs ....................................... 135
  4.4 Uncertain Processor Availability or Speed ................... 139
  4.5 Speed-Scaling for Run-Time Requirement Enforcement and Power Management ........................................ 142
  4.6 Conclusion .................................................... 145

B: Architectural Research 151

5 Adaptive Application-Specific Invasive Micro-Architectures 151
Lars Bauer, Jürgen Becker, Jörg Henkel, Fabian Lesniak, Hassan Nassar
  5.1 Introduction .................................................. 151
    5.1.1 Overview of the i-Core Architecture .................... 152
    5.1.2 Prototype and Integration ................................ 154
  5.2 Generating and Using Special Instructions (SIs) ............. 154
    5.2.1 Examples for SIs and Accelerators ....................... 155
    5.2.2 Automatic Generation of SIs During Runtime .......... 157
  5.3 Run-Time Adaptive Cache Architectures ....................... 158
    5.3.1 Intra-Tile Cache Re-Allocation .......................... 158
    5.3.2 Associativity and Cache Size Impact .................... 159
  5.4 Security Aspects of the i-Core ................................ 161
    5.4.1 Information Leakage Protection .......................... 161
    5.4.2 Isolation Modes and Mitigation of Voltage-based Attacks . 162
  5.5 Intra-Tile Multicore Extension for the i-Core ............... 164
  5.6 Near Memory Computing Extensions ............................ 165
    5.6.1 Region-based Near-Memory Computing .................... 166
    5.6.2 Near-Memory Computing (NMC) Controller ............... 168
5.6.3 Evaluation: AES Encryption Performance ........................................ 168
5.7 Task Scheduling for Runtime-Reconfigurable Processors ..................... 169
5.8 Worst Case Execution Time Analysis and Optimisations for Reconfigurable Processors ................................................................. 170
5.9 Conclusion .......................................................................................... 173

6 Invasive Tightly-Coupled Processor Arrays ........................................ 177
   Jürgen Teich, Marcel Brand, Frank Hannig, Christian Heidorn, Dominik Walter, Michael Witterauf
   6.1 Introduction ...................................................................................... 178
   6.2 Safe(r) Loops – Fault-tolerant Parallel Loop Processing .................... 180
   6.3 Orthogonal instruction processing .................................................... 182
   6.4 Run-Time Requirement Enforcement (RRE) on TCPAs ...................... 183
   6.5 Anytime Instruction Processors (AIP) .............................................. 186
      6.5.1 On-Line Arithmetic Approach ................................................. 188
      6.5.2 Bitmasking Approach .............................................................. 189
   6.6 Real-time Scheduling of I/O Data Transfers ...................................... 192
      6.6.1 Generation of Data Blocks ......................................................... 193
      6.6.2 Real-Time I/O Scheduling Problem ......................................... 195
      6.6.3 Loop I/O Controller ................................................................. 197
      6.6.4 Priority Queue ......................................................................... 197
      6.6.5 Configuration Unit .................................................................... 199
      6.6.6 DMA Engine ............................................................................ 199
   6.7 Conclusions and Acknowledgement ................................................ 200

7 Power-Efficient Invasive Loosely-Coupled MPSoCs .............................. 203
   Jörg Henkel, Andreas Herkersdorf, Heba Khdr, Martin Rapp, Mark Sagi, Mohammed Bakr Sikal, Thomas Wild
   7.1 Introduction ...................................................................................... 203
   7.2 Thermally Safe Power Constraints .................................................. 205
   7.3 Rule-based Power/Thermal Management ......................................... 206
      7.3.1 Core i-let Controller ............................................................... 206
      7.3.2 Thermal-Constrained Application Mapping and Voltage-/Frequency (V/f) Selection .......................................................... 208
   7.4 ML-based Power/Thermal Management and Modelling ...................... 209
      7.4.1 Application Mapping and V/f Scaling ....................................... 210
      7.4.2 Power- and Cache-Aware Application Migration ....................... 213
      7.4.3 Application-Aware Boosting ..................................................... 216
      7.4.4 Run-time Power Estimation and Forecasting ............................... 219
   7.5 Conclusion ....................................................................................... 225
8 Generation of Distributed Monitors and Run-Time Verification of Invasive Applications
Daniel Müller-Gritschneder, Ulf Schlichtmann, Alexandra Listl, Marcel Mettler, Li Zhang

8.1 Introduction ................................................. 229
8.2 Run-Time Verification of Invasive Many-Core Systems ........ 230
  8.2.1 Supported Run-Time Requirements ......................... 231
  8.2.2 Architectural Design ..................................... 233
  8.2.3 Benefits and Limitations of the Run-Time Verification (RV)
       System ...................................................... 238
8.3 Thermal Emulation of ASICs on FPGA Prototypes .......... 239
  8.3.1 ASIC Power Emulation .................................... 240
  8.3.2 ASIC Temperature Emulation ............................... 241
  8.3.3 ASIC Dynamic Voltage Frequency Scaling Emulation .... 243
8.4 Generation of Invasive Applications ......................... 245
8.5 FPGA-based Evaluation of Strategies for Thermal and Resource Management ........................................ 248
8.6 Conclusion ..................................................... 251

9 Invasive NoCs and Memory Hierarchies for Run-Time Adaptive
  MPSoCs
Jürgen Becker, Andreas Herkersdorf, Nidhi Anantharajaiah, Oliver Lenke, Akshay Srivatsa, Thomas Wild

9.1 Introduction ................................................... 255
9.2 iNoC as Basic Invasive Interconnect .......................... 257
9.3 Adaptive Multi-Layered NoC and Dynamic Multicast Support ... 258
  9.3.1 Multi-Layered Topologies .................................. 258
  9.3.2 Adaptive Congestion-Aware Routing Algorithm ............. 260
  9.3.3 Dynamic Block-based Multicast Routing .................... 261
9.4 Adaptive NoC based on Ant Colony Optimisation ............. 264
  9.4.1 Distributed Routing based on Ant Colony Optimisation .... 265
9.5 Region-based Cache Coherence .................................. 268
  9.5.1 Concept ...................................................... 268
  9.5.2 Features ..................................................... 268
  9.5.3 Design and Functionality .................................. 270
  9.5.4 Evaluation .................................................... 271
9.6 Near-Memory Computing .......................................... 273
  9.6.1 Concept of Near-Memory Graph Copy ......................... 275
  9.6.2 Implementation .............................................. 276
  9.6.3 Evaluation .................................................... 277
9.7 Conclusion and Acknowledgement ................................ 279
### C: Compiler, Simulation and Run-Time Support

10 Invasive Run-Time Support System (iRTSS)  
Lars Bauer, Jörg Henkel, Timo Höning, Wolfgang Schröder-Preikschat, Christian Eichler, Jeferon Gonzalez, Benedict Herzog, Tobias Langer, Sebastian Maier, Jonas Rabenstein, Phillip Raffeck, Florian Schmaus

10.1 Introduction ........................................ 285

10.2 OctoPOS .......................................... 286
  10.2.1 Memory and Address Spaces ...................... 287
  10.2.2 Processes and Threads ........................... 292
  10.2.3 Time and Energy ................................ 295
  10.2.4 Evaluation of the Concepts ..................... 296

10.3 Agent System ...................................... 298
  10.3.1 Expressing Resource Requirements and Scalability by Constraint Systems .... 298
  10.3.2 Decentralised Agent System (AS) Infrastructure .. 299
  10.3.3 Resource Management for Complex Constraints .... 300
  10.3.4 Resource Management for Malleable Applications .... 301
  10.3.5 Adaptive on-the-fly Application Performance Model .... 303
  10.3.6 Enforcing Security through Resource Management .... 304

10.4 Acknowledgement .................................. 305

11 Compilation and Code Generation for Invasive Programs  
Gregor Snelting, Jürgen Teich, Andreas Fried, Frank Hannig, Michael Witterauf

11.1 Introduction ....................................... 309

11.2 Symbolic Loop Compilation for TCPAs .............. 311
  11.2.1 Polyhedral Model .................................. 313
  11.2.2 Symbolic Mapping .................................. 314
  11.2.3 Instantiation ...................................... 317
  11.2.4 Run-time Requirement Enforcement on Loops ...... 319

11.3 Compilation for General-Purpose CPUs ............. 322
  11.3.1 Invasive X10 Front End .......................... 322
  11.3.2 PGAS Programming on iRTSS ...................... 324

11.4 Compiler Optimisations .............................. 325
  11.4.1 A New Algorithm for SSA Construction .......... 325
  11.4.2 Compiler Synthesis ................................ 325
  11.4.3 Register Permutations ............................ 327
  11.4.4 Optimising Inter-Tile Data Transfer .............. 328
  11.4.5 Near-Memory Computing .......................... 330

11.5 Acknowledgement .................................. 331
## 12 Security in Invasive Computing Systems

Felix Freiling, Wolfgang Schröder-Preikschat, Gregor Snelting, Ingrid Verbauwhede, Franziska Schirrmacher, Furkan Turan,Simon Bischof

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.1 Introduction</td>
<td>335</td>
</tr>
<tr>
<td>12.2 Attacker Model</td>
<td>336</td>
</tr>
<tr>
<td>12.3 Security Requirements and Isolation Concepts</td>
<td>337</td>
</tr>
<tr>
<td>12.3.1 Confidentiality Requirements</td>
<td>338</td>
</tr>
<tr>
<td>12.3.2 Integrity Requirements</td>
<td>339</td>
</tr>
<tr>
<td>12.3.3 Isolation Concepts</td>
<td>339</td>
</tr>
<tr>
<td>12.4 Previous Work</td>
<td>340</td>
</tr>
<tr>
<td>12.4.1 Application and Systems Software Layer</td>
<td>340</td>
</tr>
<tr>
<td>12.4.2 Hardware Layer</td>
<td>342</td>
</tr>
<tr>
<td>12.5 Information Flow Control via Control Flow Attestation</td>
<td>345</td>
</tr>
<tr>
<td>12.6 Language-based Information Flow Control</td>
<td>347</td>
</tr>
<tr>
<td>12.6.1 Timing-Sensitive Information Flow Control</td>
<td>347</td>
</tr>
<tr>
<td>12.6.2 Information Flow Control for Parallel Programs</td>
<td>348</td>
</tr>
<tr>
<td>12.6.3 Quantitative Information Flow</td>
<td>349</td>
</tr>
<tr>
<td>12.7 Run-Time Monitoring for Side-Channel Detection</td>
<td>350</td>
</tr>
<tr>
<td>12.8 Virtual Shared Memory Encryption, Integrity and Access</td>
<td>351</td>
</tr>
<tr>
<td>12.9 Hardware-based Mutual Attestation</td>
<td>352</td>
</tr>
<tr>
<td>12.10 Summary and Acknowledgements</td>
<td>353</td>
</tr>
</tbody>
</table>

## D: Applications

13 Invasive Software-Hardware Architectures for Robotics

Tamim Asfour, Walter Stechele, Nael Fasfous, Felix Hundhausen, Fabian Paus

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>13.1 Introduction</td>
<td>359</td>
</tr>
<tr>
<td>13.2 Feature Detection</td>
<td>360</td>
</tr>
<tr>
<td>13.3 Object Classification and Segmentation</td>
<td>362</td>
</tr>
<tr>
<td>13.4 Low-Latency Run-Time Adaptable Binary Neural Network Classifier</td>
<td>365</td>
</tr>
<tr>
<td>13.5 Motion Planning</td>
<td>370</td>
</tr>
<tr>
<td>13.6 Extracting Support Relations</td>
<td>374</td>
</tr>
<tr>
<td>13.7 Action Effect Prediction</td>
<td>377</td>
</tr>
<tr>
<td>13.8 Network and Robot Co-Simulation</td>
<td>378</td>
</tr>
<tr>
<td>13.9 Conclusion and Acknowledgement</td>
<td>379</td>
</tr>
</tbody>
</table>

14 Invasive Computing and HPC

Michael Bader, Hans-Joachim Bungartz, Michael Gerndt, Jophin John, Santiago Narváez Rivas, Isaías A. Comprés Ureña

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.1 Challenges and Opportunities</td>
<td>383</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>14.2</td>
<td>Invasive Computing within an HPC Node</td>
</tr>
<tr>
<td>14.2.1</td>
<td>iOpenMP Concepts</td>
</tr>
<tr>
<td>14.2.2</td>
<td>iOpenMP Implementation</td>
</tr>
<tr>
<td>14.2.3</td>
<td>Applications</td>
</tr>
<tr>
<td>14.3</td>
<td>Invasive Computing for Large Scale HPC Systems</td>
</tr>
<tr>
<td>14.3.1</td>
<td>iMPI Concepts</td>
</tr>
<tr>
<td>14.3.2</td>
<td>iMPI Implementation</td>
</tr>
<tr>
<td>14.3.3</td>
<td>Applications</td>
</tr>
<tr>
<td>14.3.4</td>
<td>Elastic-Phase Oriented Programming Model</td>
</tr>
<tr>
<td>14.4</td>
<td>Data Centre Level Invasive Services</td>
</tr>
<tr>
<td>14.4.1</td>
<td>Enabling Interactive Workloads</td>
</tr>
<tr>
<td>14.4.2</td>
<td>Power Corridor Management</td>
</tr>
<tr>
<td>14.4.3</td>
<td>Invasive Checkpointing</td>
</tr>
<tr>
<td>14.5</td>
<td>Integrating iMPI with the x86 Native OctoPOS Platform</td>
</tr>
<tr>
<td>14.5.1</td>
<td>Approach</td>
</tr>
<tr>
<td>14.5.2</td>
<td>Implementation</td>
</tr>
<tr>
<td>14.5.3</td>
<td>Applications</td>
</tr>
<tr>
<td>14.6</td>
<td>Summary and Outlook</td>
</tr>
</tbody>
</table>

15 Validation and Demonstrator

Jürgen Becker, Frank Hannig, Thomas Wild, Marcel Brand, Oliver Lenke, Fabian Lesniak

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1</td>
<td>Introduction</td>
<td>411</td>
</tr>
<tr>
<td>15.2</td>
<td>FPGA Prototyping Platform</td>
<td>413</td>
</tr>
<tr>
<td>15.2.1</td>
<td>Inter-FPGA Pin Multiplexing</td>
<td>413</td>
</tr>
<tr>
<td>15.2.2</td>
<td>Memory and Interfacing</td>
<td>414</td>
</tr>
<tr>
<td>15.2.3</td>
<td>Configuration and Debugging</td>
<td>415</td>
</tr>
<tr>
<td>15.3</td>
<td>Invasive Architecture Prototype</td>
<td>415</td>
</tr>
<tr>
<td>15.3.1</td>
<td>Invasive Network-on-Chip (iNoC)</td>
<td>417</td>
</tr>
<tr>
<td>15.3.2</td>
<td>Processor Tiles</td>
<td>417</td>
</tr>
<tr>
<td>15.3.3</td>
<td>i-Core Tiles</td>
<td>420</td>
</tr>
<tr>
<td>15.3.4</td>
<td>TCPA Tiles</td>
<td>420</td>
</tr>
<tr>
<td>15.3.5</td>
<td>Shared Memory Tile</td>
<td>421</td>
</tr>
<tr>
<td>15.3.6</td>
<td>I/O Tiles</td>
<td>421</td>
</tr>
<tr>
<td>15.3.7</td>
<td>Address Mapping</td>
<td>421</td>
</tr>
<tr>
<td>15.3.8</td>
<td>Non-Intrusive Monitoring System</td>
<td>422</td>
</tr>
<tr>
<td>15.4</td>
<td>Integration Procedure and Debugging</td>
<td>423</td>
</tr>
<tr>
<td>15.5</td>
<td>Invasive Computing Demonstrators</td>
<td>424</td>
</tr>
<tr>
<td>15.5.1</td>
<td>Enforcement Demonstrator</td>
<td>425</td>
</tr>
<tr>
<td>15.5.2</td>
<td>Prosthetic Hand Demonstrator</td>
<td>429</td>
</tr>
<tr>
<td>15.6</td>
<td>Conclusion and Acknowledgement</td>
<td>430</td>
</tr>
</tbody>
</table>
1 Introduction to Invasive Computing

Jürgen Teich, Jörg Henkel, Andreas Herkersdorf

Invasive computing is a recent research area that investigates techniques for resource-aware programming of parallel computer systems. At the same time, it is the name of the Transregional Collaborative Research Centre 89 (CRC/Transregio 89) InvasIC that has been funded by the German Research Foundation (DFG) from 2010 to 2022 in three consecutive funding phases of four years each. It received more than 30 million Euro of research funding for the three universities of FAU, KIT, and TUM and involved more than 25 principal researchers and about 50 doctoral researchers continuously during these years. Overall, more than 100 doctoral researchers received their doctor degree in InvasIC as contributors to this field of research.

In this chapter serving as an introduction to the book, we will first summarise the main ideas and concepts of invasive computing (Section 1.1). For being able to follow the concepts of invasive computing when reading individual chapters, this summary is followed by a detailed presentation of the research programme in terms of long-term research strategy, application context and basic terminology (Section 1.2).

Then, in Section 1.3, the global achievements of the CRC 89 InvasIC in the first two funding phases (2010–2018) are summarised for each of its four major so-called research areas that equally represent its main pillars of research, and the parts of this comprehensive book on invasive computing:

A: Fundamentals, language and algorithm research
B: Architectural research
C: Compiler, simulation and run-time support
D: Applications

There, major achievements in these four research areas and related projects are outlined as well as individual project-specific achievements during these years by project.

After providing an assessment of the project structure in Section 1.4, the research programme of the years 2018–2022 is finally presented from a global and a project-related perspective in Section 1.5. According to Table 1 summarising the recent structure of the CRC, this book summarises and presents the
most recent advancements and results of highlighted research goals of this 3rd and ultimate phase (2018–2022) in respective chapters.¹

This last section of this introduction closes with a description of a followed demonstrator road map and a list of working groups that helped the CRC to solve important cross-disciplinary research problems and questions.²

1.1 What is Invasive Computing?

The Idea: Invasive computing is a paradigm for designing and programming future parallel computing systems. For systems with 1,000 or more cores on a chip, resource-aware programming is of utmost importance to obtain high utilisation as well as computational, energy and power efficiency. With this goal in mind, invasive computing was introduced to provide a programmer explicit handles to specify and argue about resource requirements desired or required in different phases of execution: In an invade phase, an application asks the operating system to allocate a set of processor, memory and communication resources to be claimed. In a subsequent infect phase, the parallel workload is spread and executed on the obtained claim of resources. Finally, if the degree of parallelism should be lower again, a retreat operation frees the claim again, and the application resumes a sequential execution. To support this idea of self-adaptive and resource-aware programming, not only new programming concepts, languages, compilers, and operating systems were needed to be developed, but also revolutionary architectural changes in the design of MPSoCs (multiprocessor systems-on-a-chip) to efficiently support invasion, infection, and retreat operations.³ Latter include new concepts for dynamic processor, interconnect, and memory reconfiguration, to give some examples.

---

¹ Please note: The non-consecutive numbers of projects is the result of consecutive funding and peer review phases. Accordingly, project numbers not listed in Table 1 may either correspond to projects terminated after the 1st or 2nd funding phase or not have been funded at all.

² This book does not explicitly report on its administrative Project Z.

³ This focus on investigations on invasive MPSoCs have inspired us to give our CRC the acronym InvasIC, see http://www.invasic.de for more details.
Table 1: Projects and research areas of the CRC Invasive Computing.

<table>
<thead>
<tr>
<th>Project</th>
<th>Chapter</th>
<th>Research area</th>
<th>Principle investigators, university</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>2</td>
<td>Basics of Invasive Computing</td>
<td>Programming Paradigms, Hardware/Software Co-Design</td>
</tr>
<tr>
<td>A4</td>
<td>3</td>
<td>Characterisation and Analysis of Invasive Algorithmic Patterns</td>
<td>Program Analysis and Optimisation</td>
</tr>
<tr>
<td>A5</td>
<td>4</td>
<td>Scheduling Invasive Multicore Programs Under Uncertainty</td>
<td>Scheduling</td>
</tr>
<tr>
<td>B1</td>
<td>5</td>
<td>Adaptive Application-Specific Invasive Micro-Architectures</td>
<td>Embedded Systems</td>
</tr>
<tr>
<td>B2</td>
<td>6</td>
<td>Invasive Tightly-Coupled Processor Arrays</td>
<td>Hardware/Software Co-Design</td>
</tr>
<tr>
<td>B3</td>
<td>7</td>
<td>Power-Efficient Invasive Loosely-Coupled MPSoCs</td>
<td>Integrated Systems, Embedded Systems</td>
</tr>
<tr>
<td>B4</td>
<td>8</td>
<td>Generation of Distributed Monitors and Run-Time Verification of Invasive Applications</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>B5</td>
<td>9</td>
<td>Invasive NoCs and Memory Hierarchies for Run-Time Adaptive MPSoCs</td>
<td>On-Chip Interconnect and Memory Hierarchy</td>
</tr>
</tbody>
</table>


<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>10</td>
<td>Invasive Run-Time Support System (iRTSS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Distributed Systems and Operating Systems, Embedded Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bauer; KIT Henkel; KIT Hönig; FAU Schröder-Preikschat; FAU</td>
</tr>
<tr>
<td>C3</td>
<td>11</td>
<td>Compilation and Code Generation for Invasive Programs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Programming Paradigms, Hardware/Software Co-Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Snelting; KIT Teich; FAU</td>
</tr>
<tr>
<td>C5</td>
<td>12</td>
<td>Security in Invasive Computing Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Architecture of Parallel and Distributed Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Freiling; FAU Schröder-Preikschat; FAU Snelting; KIT</td>
</tr>
<tr>
<td>D1</td>
<td>13</td>
<td>Invasive Software-Hardware Architectures for Robotics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Industrial Applications of Informatics and Microsystems, Integrated Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Asfour; KIT Stechele; TUM</td>
</tr>
<tr>
<td>D3</td>
<td>14</td>
<td>Invasive Computing and HPC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scientific Computing, Architecture of Parallel and Distributed Systems</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bader; TUM Bungartz; TUM Gerndt; TUM</td>
</tr>
<tr>
<td>Z2</td>
<td>15</td>
<td>Validation and Demonstrator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Validation, Demonstration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Becker; KIT Hannig; FAU Wild; TUM</td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td>Central Services of the CRC/Transregio and Public Relations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Administration, Management and Public Relations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Teich; FAU</td>
</tr>
</tbody>
</table>

**Necessity and Early Achievements:** As predicted at the start of our journey in 2010, we see systems integrating more than 1,000 processor cores on a single chip today. Yet, programming such large-scale processor systems is a nightmare if resource awareness is a must and certain execution qualities must be guaranteed. Using invasive computing, a programmer may specify and, if
1.1 What is Invasive Computing?

available, the application will obtain as many exclusive resources to deliver a desired quality of execution. This dynamic and application-driven isolation is unique. Starting off from scratch in terms of invasive processor hardware, language, compiler, and operating system, we have truly fostered the fundamentals of invasive computing in the first funding phase (2010–2014): These include the definition of programming language elements for invasion primitives as well as constraints to argue about number, types, and state of resources that may be invaded (the invasive command space, Project Area A). A first language based on the language X10 by IBM as well as a compiler for translation of invasive X10 programs (Project Area C) onto invasive multi-tile architectures (architected by Project Area B) and a run-time system (iRTSS) have become available on an FPGA-based prototype (Project Z2, see Chapter 15). Invasive applications exploiting different types of processor and communication resources of an invasive network-on-chip (iNoC), see, e.g. Figure 2, have shown considerable gains in resource utilisation and efficiency in comparison with their non-invasive counterparts.

The Next Challenge: Predictability. Or: Sharing is Not Caring! By the fact that resources are temporally claimed (by default) in an exclusive manner, interference by other applications due to resource sharing may be reduced if not avoided completely. This isolation, combined with run-to-completion as the default mode of thread execution and bandwidth guarantees on communication links, allow us to provide predictable QoS also for communication. In the second funding phase (2014–2018), we played out this ace systematically by tackling (a) predictability of (b) multiple execution qualities of parallel invasive programs and including their (c) mapping optimisation. Here, our findings include new language constructs to define so-called requirements on desired, respectively amended qualities of execution. Addressed qualities include performance (e.g. execution time, throughput, etc.), security and fault tolerance. Through the analysis of application requirements from different domains including stream processing and malleable task applications, not only efficiency but also predictable execution qualities were demonstrated for applications stemming from robotics, imaging, as well as HPC. As another new yet very important facet of invasive computing, a special focus of the second funding phase was devoted to the problem of dark silicon and energy- and power-efficient computing.

The Missing Link: Beating Run-Time Uncertainties and Run-Time Requirement Enforcement. The isolation gained by invasive computing is essential to establish composability. This, in turn, paved the way for an independent and static analysis of individual program qualities in dependence
on only claim properties, giving an unprecedented gain in predictability. Yet, even if this *-predictability (boundedness of any of the above non-functional properties through the invasion of resources) can be shown to hold, (a) the effective bounds (either lower or upper) as well as (b) their variability might still be too big or too coarse to be desirable or affordable in practical application fields such as embedded real-time control. Also, claiming resources exclusively might keep these either underutilised (in case of low application workload demands) or inefficiently used (e.g. when running a claim always at maximal processor speeds) in order to safely guarantee timing bounds also for the worst case input. The third funding phase (2018–2022) was therefore dedicated to the missing link: Beating the uncertainty caused by program input, machine state and environment variation at run time. The envisioned solution: Run-time requirement enforcement. Here, we investigated hybrid techniques combining (a) static analysis of the robustness of desired qualities in dependence on input and state fluctuations and (b) systematic generation of suitable run-time requirement enforcers (RRE) (additional code that either locally or globally observes and controls the satisfaction of requirements in respective corridors at run time). This also includes the generation of necessary program-specific run-time requirement monitors (RRM). With these techniques, we reached our final goals and vision formulated already at the beginning of our mission: Invasive computing has become a—if not the—vehicle for providing resource awareness for a mixture of best-effort and predictable quality applications. We do believe huge application and business fields in embedded systems will become accessible for multicore technology through the foundations of invasive computing.

1.2 Long-Term Research Goals and Vision

Invasive computing has been introduced by our CRC as a unique novel paradigm for bringing research on parallel program design and research on multicore processing, communication and memory architecture closer together. Our research on resource-aware programming of heterogeneous multicore architectures with focus on adaptivity involved basically all levels of hardware and software.

Our distinction from mainstream parallel computing approaches has paved the road map for our long-term research strategy that may be summarised by the following ideas and fields of investigation:

---

4 See [77] for proper definitions.
1.2 Long-Term Research Goals and Vision

- **Parallel program design:** *Think invasively!* Dynamic compilation and parallelisation of sequential imperative legacy code sounds appealing and is undoubtedly economically very important. However, we believe that the problem complexity of detecting and reorganising computations at run time only by observation is in general too high. On the other hand, the output, for example in terms of observable speedups, is much too low or, in the case of performance guarantees, even just not possible. In invasive computing, an algorithm developer and programmer can actively choose which variant to select for execution depending on constraints on available resources. By being able to express such knowledge either by experience or from static analysis and profiling results, the run-time decision space becomes evidently much smaller.

- **Processor architecture:** *Exclusiveness and reconfigurability.* We investigated new tiled heterogeneous multicore architectures, the main reasons being scalability as well as power and energy efficiency. This is, of course, not a distinguishing factor. Rather the fact that a claim of resources allocated to an application during a phase of execution is non-shared by default, distinguishes invasive computing from mainstream processor architectures. The latter are still designed to give the best possible average performance by sharing and overlaying the processing of multiple instructions and even threads on another. Guarantees on the quality of program execution such as a maximal execution time or a minimal throughput processing, are not achievable, also in view of complex cache architectures and influences of run-time scheduling effects. In invasive computing, hardware design and operating system allow applications to acquire non-shared resources through reconfiguration. This improves the predictability of non-functional execution qualities a lot.

- **Optimisation goals:** *Truly multi-objective, application vs. system?* A big debate between processor and firmware designers on the one side and customers and application developers of multicore systems on the other side is the important question which objective or constraint each piece of hardware and software in a complex manycore system shall contribute and be tuned to. To make manycore systems ready to be employed in the multi-billion dollar market of embedded and cyberphysical systems, it is definitely not system objectives alone such as a maximal utilisation of resources, that need to be optimised. A minimal peak or average chip temperature so to guarantee a long lifetime of the MPSoC or a good computational efficiency on average are also rather soft
In view of application-level requirements such as a guaranteed execution time bound or security requirements that need to be enforced. What is currently done independently in hardware architecture, run-time and operating-system design is not at all straightforward to understand and sometimes questionable to reach any of these individual goals. In invasive computing, we do handle *multiple application objectives* and *constraints* such as execution time or throughput, security aspects, and levels of fault tolerance. Here, we showed how to specify such requirements, how to translate them into constraints on resources that need to be established and checked at run time through monitoring and, possibly requiring reconfiguration.

- **Multi-disciplinary prototyping:** To accomplish the goals mentioned above, InvasIC designed individual processor and system architectures and required extensions, including reconfigurable caches and memories, claimable processors, and a network-on-chip with invisible bandwidth links. Furthermore, a run-time system to realise claim allocation and constraint observation through monitors was necessary. On top of that, programming language and compiler design to compile invasive programs had to be aligned.

In our CRC, we developed a complete infrastructure for designing heterogeneous invasive multi-tile architectures, for implementing a language and simulation framework as well as a hardware prototype for experiments. Before introducing the most recent research results, we summarise the general application context of our work as well as our long-term vision.

**Application context**  Miniatrurisation in the nano era makes it possible already now to implement billions of transistors, and hence, MPSoCs with up to 100 processor cores or processing elements, see, e.g. Figure 2. There is a huge economic benefit expected if such MPSoCs, having become part already of visual computing systems, gaming and signal processing devices, become mainstream also for other types of systems, i.e. embedded systems. In the years after this CRC, we will experience 1,000 or more processors available on a single chip. Note that some GPU designs have already reached this threshold today. However, we can anticipate several major bottlenecks and shortcomings when obeying existing and common principles of designing and programming MPSoCs. The challenges related to these problems have motivated our research in invasive computing:

- **Programmability:** How to map algorithms and programs to 1,000 processors or more in space and time to benefit from the massive parallelism
available and by dealing with defects and manufacturing variations concerning memory, communication and processor resources properly?

- **Adaptivity:** The computing requirements of emerging applications to run on an MPSoC may not be known at compile time. Furthermore, there is the problem of how to dynamically control and distribute resources among different applications running on a single chip, in order to satisfy high resource utilisation and performance requirements. How and to what degree should MPSoCs therefore be equipped with support for adaptivity, for example, reconfigurability, and to what degree (hardware/software, bit, word, loop, thread, process-level)? We were able to analyse gains in resource utilisation through run-time adaptivity and temporary resource occupancy. Yet run-time adaptivity will also be necessary in order to enforce tight non-functional requirement corridors.

- **Scalability:** How to specify algorithms and programs and generate executable programs that run efficiently without change on either 1, 2, or \( N \) processors? Is this possible at all?

- **Physical constraints:** Heat dissipation is another bottleneck. We need sophisticated methods and architectural support to run algorithms at different speeds, to exploit parallelism for reduction of power density and to manage the chip area in a decentralised manner.

- **Reliability and fault tolerance:** The continuous decrease of feature sizes will not only inevitably lead to higher variances of physical parameters, but also affect reliability, which is impaired by degradation effects e.g. through device ageing. In consequence, techniques are required to compensate and tolerate such variations as well as temporal and permanent faults, that is, the execution of applications shall be immune against these. Furthermore, the control of such a parallel computer with 100s to 1,000s of processors would also become a major performance bottleneck if centrally controlled.

With the above problems in mind, we introduced and investigated a new concept of dynamic and resource-aware programming under the notion of invasive computing, which required a radical change in processor architecture, system software, and also programming language.

**Definition** ([75]) Invasive programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication and memory resources in the neighbourhood of its actual
computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.

A typical invasive program executes the following phases: (1) invade, (2) infect and (3) retreat. In an invade phase, an application asks the operating system to allocate temporally a set of processor, memory and communication resources called a claim. In a subsequent infect phase, the parallel workload is spread and executed on the obtained claim of resources. Finally, if the degree of parallelism should be lower again, a retreat operation frees the claim again and the application may terminate or resume sequential execution. This basic unit of invasive-parallel execution is called “invasive-let”: in short i-let\(^5\), see also [79] for a collection of common terms.

The chart depicted in Figure 1 shows the typical state transitions that occur during the execution of an invasive program. Depending on the level of abstraction considered, different i-let entities and associated properties are distinguished:

**candidate** (a) prospect out of a family of algorithms for the same problem to be solved, (b) potential cause of a specific operating mode of the (parallel) processor as to be enforced by the invasive run-time support system iRTSS, see Chapter 10, and (c) possibly represented and maintained as a separate source module.

**instance** (a) medium of activity of an invasive-parallel program, (b) specification of a virtual processor for it and (c) possibly represented and maintained as a separate object module.

---

\(^5\) This conception goes back to the notion of a “servlet”, which is a (Java) application program snippet target for execution within a web server.
incarnation (a) characteristic of the mode of operation to be realised by iRTSS, (b) ground anchor for the resources virtually needed for making progress in parallel processing and (c) possibly represented and maintained as a separate load module.

execution (a) actual disposition of a portion of an invasive-parallel program running on a real processor, (b) effective unit of processing implemented in soft-, firm-, or hardware (c) associated with a dedicated memory image.

Given these notions of i-lets and taking an operating system’s point of view, candidates and instances are user-level entities while incarnations and executions are system-level entities. At system level, two more terms have been established which manifest in corresponding iRTSS abstractions:

claim designates a particular set of hardware resources made available to an invading process on demand and according to selected constraints.

team designates a particular set of i-let entities (i.e. incarnations) associated with a specific claim and having the same system properties (e.g. protection, safety/security, consistency or scheduling domain) in common.

These two abstractions aid applications in the description of (static/dynamic) resource demands, the indication of the operating mode of the computing machine, and the modelling of a certain run-time behaviour of the constituting i-lets.

Invade, infect and retreat constitute the basic operations that aid a programmer to manipulate the execution behaviour of a program on the underlying parallel hardware platform.

On the other hand, invasive computing helps the programmer to decide whether to invade at a certain point of program execution in dependence of the state of the underlying machine. For example, such a decision might be influenced by the availability of resources, by the current load, by certain permissions to invade resources and, most importantly, also by the correct functioning of the resources. Taking into account such information from the hardware up to the application-level provides an interesting feedback-loop as shown in Figure 2 that enables resource-aware programming. For example, the decision to invade a set of processors may be taken conditionally at a certain point within a given invasive program depending on whether the temperature of a processor is exceeding a maximum temperature, e.g. 80 °C and if there are processors around with permission to be invaded and an average load under 50%.
Figure 2: Resource-aware programming is a main feature of invasive computing. By providing a feedback-loop between applications (apps) and the underlying hardware platform, an application program/thread also called i-let, may decide if and which resources to invade, infect, or retreat at run time; depending on the current state of the underlying parallel hardware platform. Examples of properties one can exploit are permissions, speed/performance as well as utilisation monitor information, but also power and temperature information and, most importantly, also information about faults and errors.

Long-term vision

- **Impact factor: Processor architecture of future multicore systems**: Even though we cannot compete in absolute performance of our developed FPGA demonstrator prototype, see Chapter 15, with highly integrated MPSoC designs as developed by teams of 100 and more designers at processor companies, we believe that several of our architectural inventions will influence their way of how to design and program large processor systems in the future.

- **Impact factor: Parallel programming of manycore systems**: Similarly, our paradigm of invasive computing and resource-aware programming with the provided capability to reason about requirements on non-functional aspects of program execution such as execution time, power consumption, or security properties, will have an impact on future programming languages and programming environments for the devel-
Achievements in First and Second Funding Phases (2010–2018)

Global achievements

A. Fundamentals, language and algorithm research Invasive computing (www.invasic.de) has been proposed as a novel programming paradigm that allows application programs running on an MPSoc architecture to request and distribute their workload themselves based on temporal computing demands, temporal availability of resources, and other state information of the resources (e.g. temperature, faultiness, utilisation, permissions).

However, in order to estimate and evaluate the benefits of this computing paradigm properly, the way of application development including algorithm design, language implementation and compilation tools needed to change to a large extent. On the one hand, the idea of allowing applications to...
spread their computations dynamically on a set of claimed resources and later free them again sounds promising. Early demonstrated benefits include increases of speedup (with respect to statically mapped applications) as well as higher resource utilisation, hence computational efficiency. These efficiency numbers, however, were needed to be analysed carefully and traded against the overhead caused with respect to statically mapped applications. On the other hand, being able to claim the exclusive access to sets of processing, memory and communication resources during execution time frames has been shown to allow to make multicore program execution predictable with respect to non-functional properties such as execution time, fault tolerance and security properties.

The paradigm of invasive computing itself, integrating research on algorithm and program design as well as micro- and macro-architectural changes of MPSoCs to support invasive programming was proposed first by Teich in [75]—see also [78] for concepts, [81] for overhead analysis, [27] for a language implementation in the language X10 developed by IBM and OpenMP [16], and [35] for architecture developments. Technically, the developed language extensions enable a parallel program to explore and claim resources and to copy its program and possibly data to such places in a phase of invasion, and then to execute the given problem in parallel based on the invaded claim of processing resources. Through invasion, an application is enabled to spread its computations for parallel execution based on the availability and the actual state of processing resources. For execution phases of reduced degree of available application parallelism, the application may itself perform a retreat to free occupied resources in order to optimally exploit all resources and make them available for other applications.

A claim denotes a set of processor resources that the application can use for its parallel execution. Claim construction is done by calling the invade primitive. After that, infect is used to start the actual application code in the form of a team on the previously allocated claim. The team’s code and data (i.e. its i-lets) is spread onto infected resources for subsequent parallel execution. Once the execution on all claimed cores finishes, the number of cores of the claim can be altered by calling invade or retreat to either expand or shrink the application’s claim. In the case of retreat, the processing elements are cleaned up from the i-let entities that have been set up by infect. Alternatively, if the degree of parallelism does not change, it is also feasible to dispatch a different program (i.e. team) onto the same set of cores by issuing another call to infect. When a program finishes its execution, its initial claim gets empty and the program terminates. Notably, a claim may not only contain
1.3 Achievements in First and Second Funding Phases (2010–2018)

Figure 3: Example of a configuration of a multi-tile InvasIC MPSoC including I/O tiles, memory tiles, RISC-compute tiles, tiles with i-Cores and TCPA (tightly-coupled processor array) compute tiles. These are interconnected by an invasive on-chip network called iNoC. Shown is an instance in time where currently three different applications have invaded the tiled resources.

processing resources, see Figure 1 for illustration. but also memory as well as communication resources.

A temporal snapshot of an MPSoC occupied currently by three application programs each with different number and types of resources (each claim highlighted by a different colour), is shown in Figure 3. We decided that, by default, a claim is not shared, the reason being that through the separation of resources, predictability in multiple qualities of execution may be gained for an individual application as interferences between concurrent executions of multiple applications may be avoided by construction. This does not only hold for time-sensitive workload but also for isolation of information flows on an MPSoC for the purpose of security.

As to what concerns the language implementation, we decided to embed the basic primitives of invasive computing into an existing language, X10, rather than to develop a new language. X10 supports the model of partitioned global address space (PGAS), which is also the assumption for our architectural developments on heterogeneous multi-tile invasive MPSoCs in project area B. Also, it contains all required mechanisms and constructs for concurrent execution of program entities called activities, synchronisation, and means to specify where to spawn child activities through the notion of places. A place has a natural correspondence with a tile of processor and memory resources in an invasive multi-tile architecture. Finally, we decided rather not to change X10
1 Introduction to Invasive Computing

itself in order to profit also from a compatibility and language conformance in
the future and therefore implemented the concepts of an i-let as well as the
invasive, infect and retreat instructions following a library-based approach [27].
This implementation is called InvadeX10. An example application is given
below. The code is part of an invasive JPEG-decoding case study. The algorithm
computes an Irreversible Colour Transform (ICT), which converts an image
from the RGB to the YCbCr colour space.

```scala
def forwardIctTCPA(i: Int, img: Image): Unit = ... 
def forwardIctCPU(img: Image): Unit = ... 
```

The program behaviour may be simulated using our developed MPSoC simu-
lation framework InvadeSIM [66, 68]. Alternatively, the program may be
compiled using our compiler developed in Project C3, see Chapter 11, to run on
the prototype of the architecture shown in Figure 3 that has been developed
jointly by the teams in project area B.

The static method Claim.invade takes as argument a data structure called
constraints and returns a claim object, which represents the set of allocated
resources. A claim provides an infect method to distribute computations
across the claimed processing resources. The argument of infect is the i-let
object, which contains the code to execute together with initial data. The
infect call blocks the program, until all spawned i-lets finish. These i-lets
then build the team that is associated with the infected claim. Afterwards,
the retreat method frees all resources within a claim, such that the claim is
empty. In the sample program above, the application tries to invade a tightly-
coupled processor array (TCPA) [26] (Project B2, see Chapter 6) with 10×10
PEs in order to speed up the ICT operation. The program also indicates the
philosophy of invasive computing to put it into the hands of the programmer
to decide what to do in the case a desired claim is not found or returned by
the run-time system. In the above case, the application decides to choose the CPU variant of the code of the ICT and run it on its current claim instead.

*-Predictability: Predictability of non-functional properties of program execution are currently still neglected in programming language design. In [77], we generalised this term that is widely used synonymously with timing predictability to coin the aspect of boundedness also for other qualities of interest as *-predictability:

**Definition** ([77]) Let $o$ denote a non-functional property of a program (implementation) $p$ and the uncertainty of its input (space) given by $I$ and environment by $Q$. The predictability (marker) of objective $o$ for program $p$ is described by the interval

$$o(p, Q, I) = [\text{INF}_o(p, Q, I), \text{SUP}_o(p, Q, I)]$$

where $\text{INF}$ and $\text{SUP}$ denote the infimum and supremum under variation of state $q \in Q$ and input $i \in I$, respectively.

As an example, a program $p$ that under the uncertainty of dynamic power management and input leads to a total power consumption $P$ between 1 and 2 W is described by $P(p, Q, I) = [1, 2]$. Moreover, the variability may be refined, e.g. by assuming either a uniform distribution or by any other discrete or continuous distribution.

In Figure 4, five program variants $p_1, \cdots, p_5$ of one and the same program are shown for a two-dimensional space of objectives $o_1$ and $o_2$. For example, let $o_1$ denote power consumption $P$ and $o_2$ execution time or latency $L$. Moreover, the figure shows a so-called requirement on maximal latency and a minimal and maximal power consumption requirement. It can be seen that programs $p_1$ and $p_2$ are infeasible as they violate either the latency requirement ($\text{SUP}_L(p_1, Q, I) \geq L_{\text{max}}$) or the power requirements ($p_2$ with $\text{INF}_P(p_2, Q, I) \leq P_{\text{min}}$). Another interesting observation is that other program implementations/variants are non-comparable. For example, although program $p_4$ has a lower latency $o_2$ than $p_3$ for all variations of input and hardware state ($\text{SUP}_L(p_4, Q, I) < \text{INF}_L(p_3, Q, I)$), its power consumption $o_1$ may be either higher or lower under the same variation of $Q$ and $I$. One can see also that optimising for predictability in the sense of minimising its predictability interval is completely orthogonal to minimising the objective values. For example, a design point or program variant may have a very low latency in the best case, but may terribly vary under the uncertainty parameters. Therefore, in decision making, preference shall be given to a design or program $p$ whose worst-case latency is minimal rather than the best case.
Requirements vs. constraints: One major investigation in project area A at the language and programming level was to replace constraints by so-called requirements on execution qualities. We discovered that it might be difficult or even impossible for a programmer to specify by hand constraints on number and type of resources to achieve a desired quality of program execution. Rather, a programmer would like to specify so-called requirements on non-functional program execution properties instead. The following code listing gives an example of a class of requirements that may precede an invade section of a program. These requirements, such as a timing requirement of a maximal latency bound of 65 ms, or a throughput requirement of minimally 25 frames/s (fps) are analysed at compile time.

```c
/* Examples of Performance Requirements */
@REQUIRE(Latency(0, 65, "ms", "soft"))
@REQUIRE(Throughput(25, 100, "fps", "hard"))
/* Example of a Reliability Requirement */
@REQUIRE(PFH(0.001, 0.0000001))
/* Example of a Power Requirement */
@REQUIRE(Power(1, 2, "W", "soft"))
```

For example, a power requirement, as shown above, is satisfied for an i-let \( p \) or code segment therein, if the constraint set of a suitable claim of resources will guarantee \( \text{INF}_p(p, Q, I) \geq 1 \) and \( \text{SUP}_p(p, Q, I) \leq 2 \). Note also that we distinguish between soft and hard requirements: Hard requirements must never be violated for whatever reason. Whereas soft requirements shall be
satisfied most of the times, but their occasional violation may be tolerated. The code listing example above also demonstrates a reliability requirement in terms of bounds on tolerable probabilities of failure per hour (PFH) as well as a soft power consumption requirement of the resources claimed. The satisfaction of the first one may require triggering the invasion of fault-tolerant spares [44], or task re-execution or duplication and error checking. Our design flow as developed in the second funding phase automatically determines suitable claim constraints that adhere to a set of given requirements and then replaces the requirement annotations in the X10 source code directly by the respective set of claim constraints. For example, the power requirement can be translated into constraints on the power management of the claim, e.g. a restriction of allowed DVFS settings on the claimed cores.

**Hybrid application mapping and ActorX10:** To fulfil the aforementioned user requirements for applications to be executed at run time, we proposed the following design flow, see Figure 5. Obviously, to perform a search for suitable claims that fulfil a given number of constraints at run time might be very time-intensive. Hence, we decided to follow a hybrid application mapping (HAM) approach by combining design-time analysis of application mappings with run-time management [85].

Figure 5 gives an overview of our HAM approach that combines static performance analysis during a design space exploration with dynamic (run-time) mapping. Currently, the design flow works for applications that are programmed following an actor programming model. Therefore, we shortly present our language development of ActorX10 [67], an X10 library of a formally specified actor model based on the PGAS principles. For formal analysis, the actor model of an application is expressed by an application graph, which models the tasks as executed by each actor as well as the data dependencies between them. Formally, an application graph of a program \( p \) is an acyclic, directed, bipartite graph \( G_A(V, E) \). \( V \) is the union of the set of tasks \( T \) and the set of messages \( M \), \( V = T \cup M \). Each directed edge \( e \in E \) connects a task \( t \in T \) with a message \( m \in M \) and vice versa. Each \( m \in M \) has exactly one predecessor and one successor. Once admitted, the application graph is executed periodically with the period \( \pi \) and the end-to-end latency \( L \) has to meet a certain deadline.

A considered multi-tile NoC-based MPSoC target platform, see e.g. Figure 5, is also modelled as a directed graph \( G_{arch}(U, E_U) \) which consists of a set of heterogeneous tiles \( u \in U \). Each tile has a certain resource type \( r \in R \) which can be derived by the function \( \text{type} : U \rightarrow R \). Communication between tiles is routed over the NoC. Therefore, each tile is connected with a router and routers are connected with each other to form a 2D mesh. Each directed
connection is modelled by a link \( l \in E_U \) in the graph. Each link \( l \) has the link width \( LW \) and is operated at a frequency \( f \). If a task on the tile \( u_1 \) sends a message to a task on another tile \( u_2 \), the data is partitioned into flits, which have the size of \( LW \), and is routed over the NoC along a route of consecutive routers.

Non-functional properties of an application and their bounds depend on the mapping of the application \( G_A(V, E) \) onto the architecture \( G_{arch}(U, E_U) \). The mapping involves (a) the binding of tasks onto the available tiles. This is described by a function \( \beta : T \rightarrow U \), and (b) routing of messages over a set of connected links in the NoC, which form a path from the tile executing the sending task to the tile executing the receiving task. This is formalised by a function \( \rho : E \rightarrow 2^{E_U} \). The number of visited routers is defined by the hop count function \( hops(u_1, u_2) \). During design space exploration (DSE), the space of mappings is explored and each candidate evaluated regarding non-functional properties, such as timing, energy or resource usage, at design time.

Figure 5: Overview: At design time, a design space exploration is carried out that analyses multiple user-given non-functional objectives of execution, e.g. real-time requirements. Based on resource isolation as provided through invasive computing, each application, modelled by a task graph, may be analysed independent of each other, thus providing composability. Based on this static characterisation of classes of equivalent run-time embeddings, also the search space for run-time mapping is greatly reduced. The information of each Pareto point is a graph with annotated constraints on which type and number of resources to be claimed at run time. This data structure is passed to the run-time system for search of claim constellations with preanalysed intervals of uncertainty. Figure adapted from [77].
time. The set of Pareto-optimal mappings, so-called operating points, are stored and handed over to the run-time mapper. Each of these operating points is accompanied with a so-called constraint graph. A constraint graph specifies which tasks need to be mapped together onto which resource type and constraints regarding the routing for the communication between task clusters. Any run-time mapping which meets the specified constraints complies to the bounds analysed at design time. In [85], we proposed a backtracking algorithm as an example of a run-time mapper. Overall, this hybrid application mapping approach heavily relies on the concept of composability [1] which is based on resource isolation in invasive computing. This allows us to analyse different applications independently from each other. On top, compositional analysis techniques may be applied such as compositional timing analysis to infer global timing properties of an application from the local analysis of their tasks.

As another breakthrough result on HAM [85], we were able to reduce the search space for run-time mappings drastically by elimination of symmetry from the search space, see [71]. Each operating point in Figure 5 stands for a class of equivalent mappings in terms of the explored objectives.

B. Architectural research Figure 3 shows an instance of a heterogeneous multi-tile invasive MPSoC. Hardware resources are partitioned into tiles which are connected by an invasive network-on-chip (the iNoC). The following tile types have been developed from which a NoC-connected concrete invasive SoC architecture may be arbitrarily composed: I/O tiles, memory tiles, RISC-compute tiles, tiles with i-Cores (Project B1, see Chapter 5) and TCPA (tightly-coupled processor array) (Project B2, see Chapter 6) compute tiles. Also shown in the figure is a state of invasion where three applications have currently claimed all the available processing resources creating a partitioned MPSoC. The invasive network-on-chip (iNoC) (Project B5, see Chapter 9) represents the communication interconnect backbone of any invasive MPSoC architecture. It is notable that not only application data, but all requests to invade either memory space, communication capacity on iNoC internal or external I/O links, or processor resources within tiles, all pass through this on-chip network. In order to provide QoS support between communicating tiles, the iNoC provides so-called guaranteed bandwidth connections as a unique feature. This enables predictable bandwidth guarantees not only between two, but also multi-hop invaded links. All types of tiles contain a network adapter that provides a fast interface for all inter-tile communication demands handled by the iNoC.
In the following, we present a timing jitter analysis technique for iNoC-based MPSoC embeddings of communications. The jitter represents the difference between best-case and worst-case timing and is a predictability marker for latency $L$. The analysis is then applied to the mapping of complete applications that are modelled and programmed as communicating actors with periodic execution behaviour and mapped to the NoC-based architecture as presented above. Based on this jitter analysis, the explored Pareto front consists of operating points with interval jitter in each objective. Further, we detail how the predictability marker NoC throughput $Tr_{NoC}$ can be calculated.

The invasion of NoC resources [31] has been realised through state-of-the-art virtual channels and a weighted round-robin (WRR) arbitration [29]. An application can reserve a certain budget $SL(m)$, also denoted as service level, for the message $m$ from the maximal available budget of a link $SL_{max}$. The budget refers to time slots which are assigned to messages in the interval of $SL_{max}$. By this reservation, the state space $S$ is reduced and the guarantees for NoC latency $L_{NoC}$ and throughput $Tr_{NoC}$ become much tighter than without a reservation scheme. During the invasion phase, the availability of time slots and virtual channels is checked and then assigned to the communication flows according to their request. In global TDMA, the position of the time slots inside the arbitration interval depends on the time slot position in the preceding link. Æthereal, used in CompSOC [20], is an example for such a TDMA NoC.

It has very tight latency guarantees but degrades the throughput, utilisation, and it is too complex to find a feasible run-time assignment. For this reason, we use WRR arbitration where only the number of time slots is fixed not their position (see Figure 6). If a message does not utilise its reserved time slot, it can be used by another message. This increases the overall throughput while preserving static analysability and composability.

In the best case, all time slots can be utilised by message $m$ exclusively and there is no interference by other messages on the links. The best-case throughput of a message is then:

$$INF_{Tr_{NoC}} = f \cdot LW$$

---

6 A WRR-arbitrated NoC leads to higher utilisation rates at slightly reduced tightness of the latency bounds than global TDMA NoCs.

7 A NoC without any reservation scheme could lead to unbounded NoC latency as certain packets may suffer from starvation and never arrive at the destination.
Figure 6: Shown is a tiled architecture with two applications \(a\) and \(b\) sharing the iNoC. In order to provide guaranteed service, each message may invade a percentage of available throughput. In the example above, the green application \(a\) claims \(SL(m) = 4\) of the available throughput \((SL_{max} = 8)\) between the left tile and the right tile. Even if not the same slots may be reserved in each link \(l\), the minimum and maximum communication times of each message are bounded and may be statically analysed.

The NoC latency \(INF_{LNoC}\) only depends on the number of flits per message \(flits(m)\), the frequency \(f\), the router delay \(\Delta_R\), and the hop count \(hops(\rho(m))\):

\[
INF_{LNoC}(m, \rho(m)) = \frac{flits(m)}{f} + hops(\rho(m)) \cdot \frac{\Delta_R}{f} \tag{3}
\]

In the worst case, only the time slots of the allocated budget \(SL(m)\) from the maximal budget \(SL_{max}\) can be utilised which decreases the throughput as follows:

\[
SUP_{TrNoC}(m) = \frac{SL(m)}{SL_{max}} \cdot INF_{TrNoC} \tag{4}
\]

The NoC latency increases accordingly to:

\[
SUP_{LNoC}(m, \rho(m)) = INF_{LNoC} + \left(\left\lfloor \frac{flits(m)}{SL(m)} \right\rfloor - 1 + hops(\rho(m))\right) \cdot \frac{(SL_{max} - SL(m))}{f} \tag{5}
\]

For a compositional calculation of the worst-case/best-case end-to-end latencies \((SUP_L/INF_L)\) of an application \(p\), the state space \(S'\), and the input space
1 Introduction to Invasive Computing

$I'$, NoC latencies ($SUP_{LNoC} / INF_{LNoC}$) and computation latencies ($SUP_{LComp} / INF_{LComp}$) on the critical path path of the mapped application are summed up. For the sake of simplicity, we assume in the following that only one task $t$ is executed on one tile with analysed best-case $INF_{LComp}(t, \beta(t))$ and worst-case $SUP_{LComp}(t, \beta(t))$ execution latency. The best-case end-to-end latency $INF_L$ and worst-case end-to-end latency $SUP_L$ can then be analytically calculated as follows:

\[
INF_L(path, \beta, \rho) = \sum_{t \in path \cap T} INF_{LComp}(t, \beta(t)) + \sum_{m \in path \cap M} INF_{LNoC}(m, \rho(m))
\]

(7)

\[
SUP_L(path, \beta, \rho) = \sum_{t \in path \cap T} SUP_{LComp}(t, \beta(t)) + \sum_{m \in path \cap M} SUP_{LNoC}(m, \rho(m))
\]

(8)

Apart from the predictable iNoC, we devised a three level addressable memory hierarchy for our developed invasive MPSoC architecture prototypes (Project Z2, see Chapter 15):

- **Scratchpad** – each core within a RISC compute tile has its private 8 kB scratchpad memory which allows for low-latency accesses managed by the application programmer. Scratchpad memory is non-cached.

- **Tile-local memory (TLM)** – each tile contains an 8 MB SRAM TLM that is shared among all RISC cores on that tile. Regions of the TLM may be invaded and accesses to the TLM are generally cached. Cores from remote tiles can access the TLM using (flit-based) message passing services or explicit load/store commands through the iNoC. However, cache coherency will only be guaranteed for cores within one tile (or X10 place, if the X10 place is bounded by the borders of a compute tile).

- **Partitioned on-chip and external memory** – this type of memory can also be accessed through message passing services and ld/st over the iNoC. Partitioned global memory either consists of 8 MB on-chip SRAM tiles or 256 MB off-chip SDRAM modules. Note that although these types of memories can be accessed from all compute tiles, their address space is partitioned among different X10 places.

The only non-addressable types of memories are:

- **L1-core cache** – each core has an 8 kB instruction cache and 32 kB data cache. Again, coherency is guaranteed among cores within one tile.
C. Compiler, simulation, run-time support and security  For our definition of the language InvadeX10 [27] for invasive programming as described briefly before, the compiler technology was successfully developed by Project C3 (see Chapter 11) to target as well RISC as i-Core CPU cores and TCPAs. The compiler directly translates InvadeX10 to the intermediate representation as provided by the libfirm code generator. The use of libfirm enabled a variety of specific optimisations, such as the minimisation of inter-tile data transfers, or the support for i-Core special instructions in X10. The compiler was extensively validated and partially verified.

Before prototyping invasive programs on an invasive MPSoC architecture, simulative support for single and multi-tile invasion as well as invasion management strategies through agent strategies belonging to the iRTSS (Project C1, see Chapter 10) was necessary. The simulation framework InvadeSIM as investigated and developed in Project C2 does provide the required backbone for fast turnaround times also of novel architectural and monitoring components, as well as design space exploration of architecture variants and invasion strategies before hardware prototyping may start. Figure 7 presents the InvadeSIM framework [66]. Significant speedups in simulating concurrent applications on manycore architectures are possible through proper parallelisation techniques presented in [68]. The simulation threads are assigned for parallel execution to the cores of the host machine as indicated in Figure 7 by red arrows from the InvadeSIM kernel to the host machine. The first parallelisation technique introduced in [68] is called signalling. It is based on performance counter signals that are raised if a certain simulation thread passes the simulation time of a waiting simulation thread. In this case, the simulation thread is stopped and it is checked, whether the waiting thread can be waked up. If yes, it is activated and both threads run concurrently. Nevertheless, this technique may drastically reduce the number of simultaneously running simulation threads. This disadvantage could be compensated by an improvement called luck-shot in which snapshots of the current simulated time of all running simulation threads are determined and used for comparison at a synchronisation point. The third investigated parallelisation
technique is called *heartbeat*. It is based on a dedicated thread, which periodically checks, whether simulation threads can be activated. Here, the choice of a proper period has been shown to be essential for the resulting speedup and a statically determined period might be not the best choice. Thus, we finally proposed an *adaptive heartbeat* technique, which dynamically changes its period according to the dynamic behaviour of the application to maximise the speedup gained through parallel simulation.

A working group on system software was very active here in reasoning about important design decisions that had to be taken prior to any implementation work on compilation and on concerning the operating-system and run-time–system layers. In the following, we do therefore message important cross-disciplinary results concerning important terms such as “application”, “thread”, and “granularity” issues of invasion and infection.

Discussions within the CRC revealed that the generally usual notion of “application” has quite different meanings in the diverse technical disciplines. The range goes from a single “thread” within a (non-sequential, multithreaded) program looking into a very dedicated task to a (possibly complex formation of a) logically self-contained assembly of programs that jointly performs a certain computation or control function. By way of example, the former case relates to read-out of a sensor device and the latter case to some feedback control system consisting of many sensors, actuators, and (hardware/software) means...
1.3 Achievements in First and Second Funding Phases (2010–2018)

for human-computer interaction. Within the CRC, “application” much more corresponds to the latter than the former.

The question of the adequate granularity of invasion—namely core or tile—of invasion, as significant at invade time, and infection, as significant at infect time, in terms of the hardware units affected by the respective measures was a central issue of the discussions. These two actions while executing an invasive (parallel) program establish the moment of allocation of hardware units requested by an application (entity) and dispatching of i-lets to some processing element (i.e. core), respectively. A common understanding was to differentiate between these moments (“separation of concerns”) and, as a further consequence, to accept different granularity depending on the level of abstraction considered.

From an operating-system point of view, invasion is a complex and expensive operation, where generality and/or usability must be balanced against performance and implementation simplicity. Furthermore, future hardware will perhaps offer a very high number of tiles with a small number of cores on each tile. This claim-based differentiation is made for better control of interference in the case of multi-programmed compute tiles that are claimed (i.e. shared) by applications of different and possibly conflicting quality requirements in terms of non-functional properties (such as timing, jitter, energy or noise). Once a team of i-lets is assorted for a specific claim—after return from a successful call to invade, but before the call to infect for that very claim—the association between i-let and claim identifier or wildcard, respectively, is established. During infection, OctoPOS implementing the mechanisms of iRTSS (see Chapter 10) then tags all i-lets with the identifying information related to the claim of their team.

Within the second funding phase, Project C5 (see Chapter 12) began to lay the groundwork for establishing security properties in invasive computing systems. A comprehensive attacker model was devised that specifically included the case of a compromised operating system (root level attackers), a plausible assumption in the context of run-to-completion semantics. On a requirements level, the concept of $\varepsilon$-confidentiality [11] was defined. This property allows to express a desired bound on the amount of information leakage of at most $\varepsilon$ bits per second from within the application. It therefore generalises known confidentiality concepts and adapts them to contexts that share resources.

The involvement of the Mercator Fellow Ingrid Verbauwhede from KU Leuven, Belgium, was critical to the success of Project C5. Her expertise in hardware-based security ideally complemented the more software-security oriented
expertise of the other members of the project and resulted in a number of strong results that were published in excellent venues, such as a novel software protection technique for embedded devices that enables predictability against root-level attackers (Soteria, published at ACSAC 2015 [21]), as well as a technique to guarantee also control-flow integrity against root-level attackers (Sofia, published at DATE 2016 [9] and an extended version in the journal Computers & Security [8]). A great outcome of this collaboration was also a foundational overview article on trusted computing architectures published in [52].

D. Applications and demonstration The projects in area D aim at demonstrating invasive computing and its advantages within their application domain.

Project D1 explored benefits and limits of invasive computing in humanoid robotics. It combined contributions from various projects into a common scenario of visually guided grasping, such demonstrating dynamic load balancing between multiple RISC cores, i-Cores and TCPA, based on invasive computing by support of iRTSS (OctoPOS and agent system).

For a robot object detection chain [62] based on camera images, an ActorX10 [67] implementation was developed, encapsulating each of the object detection tasks by an actor. Thereby, pipelining is exploited, which allows all of the tasks running in parallel while each is processing a different image instance

Figure 8: Example of an application mapping of the object detection (light grey) task chain as well as multiple Monte Carlo simulation instances (red) on a $4 \times 4$ NoC-based architecture.
and subsequently sending data to the next stage and waiting for new data from the previous stage. In order to show the advantages of invasive computing in terms of predictability and ability to reduce execution time and throughput jitter, an application scenario was considered, where the object detection application is executed concurrently with several communicating and parallel Monte Carlo simulations calculating an approximation of the number $\pi$. The target architecture is a $4 \times 4$ NoC-based architecture comprising of different types of tiles as well as different types of processors on these tiles. One mapping of the applications on the target architecture is shown in Figure 8. Additionally, the used NoC links between the communicating tasks of each application are highlighted in colour. It can be seen that there are overlapping paths between the object detection application and the Monte Carlo simulations (all horizontal links in the second column of the tile-based architecture).

It can be observed that the number of objects to process by some actors (i.e. SIFT feature descriptor computation) is dependent on the input image and number $obj$ of corners detected. For a worst-case analysis, we assume in the following that this number is bounded from below by zero and above by a maximum of $max(obj)$ corners. Without these restrictions, the execution time might not be bounded at all.

The following experiments describe the simulation of the above application on a 4×4 NoC architecture using InvadeSIM [66, 68]. In order to guarantee bandwidth requirements between the tasks of the object detection application, guaranteed service (GS) NoC channels with different service levels must be reserved for messages between communicating tasks after the mapping of the actors to tiles has been accomplished by the agent system. In contrast, the two interfering Monte Carlo simulations only use best effort (BE) channels between communicating tasks. They are configured to claim up to eight BE transmissions in parallel to increase the utilisation of the NoC.

The observed latency and throughput variability is shown in Figure 9 and Figure 10. The latency $L$ denotes the time it takes to process one image frame. Throughput, measured in fps, denotes the frequency of how many frames can be processed by the pipeline per second. The service level reserved for each GS channel of the object detection application is plotted for values of $SL(m) = 1$ to $SL(m) = 6$, $\forall m$. What was already analytically evaluated for just two communicating tasks of the application shown in blue in Figure 8, can also be seen in the simulation of the complete application: In both cases, the objective values and predictability markers significantly improve when increasing the reserved communication bandwidth of messages:
1 Introduction to Invasive Computing

![Figure 9: Plot of the observed (simulated) overall application latency $L$ and its variations of the object detection task chain for different service levels $SL$ allocated for inter-task communications.](image)

![Figure 10: Plot of the observed (simulated) overall application throughput $Tr$ and its min/max values of the object detection task chain in dependence of different service levels $SL$ allocated for inter-task communications.](image)

For the latency marker, $L(p, \{SL(m)=1, \forall m\}, I) = [111.34, 162.34]$ drops to $L(p, \{SL(m)=6, \forall m\}, I) = [89.66, 90.66]$, and for the throughput predictability marker from $Tr(p, \{SL(m)=1, \forall m\}, I) = [18.88, 20.65]$ to $Tr(p, \{SL(m)=6, \forall m\}, I) = [24.02, 25.63]$. If the user would formulate a requirement of an end-to-end latency of at most 100 ms, the operating point with $SL(m) = 5 \forall m$ could be preferred as further increasing the service levels does not further improve the throughput and its variation, but would reduce the bandwidth available for other applications. Still, the simulation-based evaluation only allows to determine observed predictability markers and is therefore only applicable for soft user requirements.

What also could be seen is that although not sharing resources helps to give predictable execution times, the jitter due to a variable number of objects (uncertainty of input) may still lead to quite some undesired variability. Therefore, techniques for lowering such variability by run-time requirement enforce-
1.3 Achievements in First and Second Funding Phases (2010–2018)

Figure 11: Adaptive mesh refinement in sam(oa)² for a parallel simulation of the Tohoku 2011 tsunami (image from [33]).

*ment for tightening of predictability intervals, e.g. through input-adaptive power management on the tiles, were investigated in our CRC as well as their verification.

Project D3 (see Chapter 14) explored the potential of the invasive computing paradigm in the context of HPC applications. The application range of invasive computing was significantly increased by extensions of MPI to allow applications to invade and retreat resources. A small number of new MPI calls were added to the message passing interface to enable an MPI application to detect and adapt to resource changes [10]. These extensions were implemented in MPICH with a special focus on latency hiding mechanisms. MPI applications are executed on the hardware via a batch scheduling system that selects suited jobs from the batch queues, allocates resources, and initiates the application. The most widely used batch scheduler SLURM was extended to manage an invasive partition of the HPC system and to dynamically reallocate resources among invasive jobs executing in that partition.

In [33], a large-scale malleable tsunami simulation (see Figure 11) realised on a developed invasive MPI infrastructure was presented. This fluid-dynamics application is a representative, real-world scientific application implemented with adaptive mesh refinement. It represents the class of tightly coupled parallel applications that typically have spatially-dependent data and a non-trivial communication topology. Malleability of this application was achieved with small communication overhead and led to a significant improvement on the application’s resource efficiency utilising hundreds of MPI ranks.

Finally, the major role of Project Z2 (see Chapter 15) was to develop an FPGA-based prototyping infrastructure for validating the principles of invasive computing. For this purpose, the contributions of the different projects across
all project areas have been integrated into one heterogeneous invasive platform in order to demonstrate the advantages of invasive computing such as a quantifiable quality of service, resource utilisation, and speedup [43, 62]. One prototyping platform called proFPGA from ProDesign is in use at each site (FAU, KIT, TUM) as validation and demonstration platform. Each system allows for prototyping of multi-million gate designs of up to 80 standard RISC processors and/or accelerators (i-Cores [34], TCPAs [26]), and thus, hardware prototypes of novel invasive multicore architectures [2, 35]. Several projects contributed a substantial amount of manpower to the integration and debugging of the demonstrators, see Chapter 15 for details.

Achievements per project area

A. Fundamentals, language and algorithm research  The main achievements of project area A may be summarised as follows.

- Project A1 (see Chapter 2) investigated the basics of invasion—the fundamental programming model for invasive and resource-aware computation. A major focus of the second funding phase was on the topic of predictability of non-functional properties of invasive parallel program executions. On the language side, the language X10 has been extended to handle a changing number of places by adapting its run-time and standard library [3]. Also, the previously described methodology called hybrid application mapping (HAM), [85] is supported. Furthermore, we introduced an interface for asynchronously-malleable applications that are able to react to external resource adaptations at any time [7]. Such applications can directly adapt to resource changes and increase the overall efficiency. We also developed a memory model [94] for X10, which is a foundation for modelling the semantics of our programming language. Through the isolation of applications by invasion of exclusive resources, predictability (boundability) of performance (e.g. latency, throughput) and other non-functional properties such as reliability and security [86] was shown to be achievable, denoted by the term *predictability [77]. For specifying such desired bounds on execution qualities, we extended the language InvadeX10, which is based on the PGAS language X10 from IBM, to include so-called requirement tags. This is an annotation to an invasive program segment in which the application programmer specifies a desired range of a non-functional execution property for that proper segment. Previously, the application programmer needed to specify complex sets of constraints on resource numbers and types that need to be invaded to provide a desired qual-
ity of program execution. Now, the programmer may just specify the quality. Through HAM [85], these requirements may be automatically replaced by respective constraint sets at compile time by a static design space exploration. These may then be used by the run-time resource management for claim search and assignment, see e.g. [87]. The methodology is particularly tailored to multi-media and stream processing applications with execution time and throughput requirements [88].

In order to characterise such streaming applications, the previously mentioned actor-based language called ActorX10 has been defined [67] in cooperation with Project A4 and Project C2.

- Project A4 (see Chapter 3) characterised, investigated existing and developed novel invasive algorithmic patterns, which are motivated by applications from project area D. Using design space exploration (DSE), operating points to achieve desired quality numbers (performance, energy consumption, etc.) of applications are precomputed. The following research problems were addressed: (1) How can we derive knowledge about an application’s performance and pass it to the run-time system? Together with Project A1 and Project C2, the ActorX10 language and library [67] have been developed, see above. The use of actor-based design for InvadeX10 applications realises a separation of computation and communication and automatically serves as a design entry for a subsequent DSE. (2) How can we prepare online scheduling decisions based on this knowledge, to improve the system performance w.r.t. various performance metrics? Here, after covering the parameter space of the application w.r.t. resource requirements during DSE [71, 85], Pareto-optimal operating points [65] are distilled and subsequently passed to the run-time system to guide the resource management. (3) How can invasion-aware algorithmic patterns be developed that benefit from invasion on heterogeneous platforms? For this purpose, SWE-X10 [63, 64], a proxy application for space- and time-adaptive tsunami simulation, was examined and implemented as well as a prototype of an object detection image processing application chain. Both proxy applications were written in InvadeX10 [27] using the ActorX10 library [67] and serve to evaluate novel invasive algorithmic patterns and how to exploit operating points distilled by DSE. The developed design flow allows the run-time system to execute applications with predictable performance (in terms of upper and lower bounds on various quality numbers [77]), where applications are statically optimised under worst-case assumptions and for invasive MPSoCs that allow for a static timing analysis (see, e.g. [87]).
**B. Architectural research**  The main achievements of project area B may be summarised as follows.

- Project B1 (see Chapter 5) investigated mechanisms that provide run-time adaptivity both in the microarchitecture (μArch) and by using a run-time reconfigurable fabric. The focus on research here was on efficiency and predictability of execution on the i-Core. To improve efficiency, we introduced the COREFAB (COncurrent REconfigurable FABric utilisation) technique [23] that allows general-purpose CPUs in a reconfigurable shared-memory multicore (which corresponds to a tile in the InvasIC architecture) to access the fabric of a reconfigurable core. Compared to state-of-the-art, COREFAB improves performance of the LEON cores in an i-Core tile by 1.3× on average, without reducing the performance of the i-Core itself. Other efficiency improvements tackled the identification of memory access patterns in kernels to support automatic generation of accelerators at compile time [25] and online binding of nodes in a dataflow graph to accelerators on the reconfigurable fabric [24]. A further explored approach to improve efficiency is the concept of *Auto-SIs*, a concept that is dynamic and transparent. During run time, the executed program is observed and loops are identified. To speed up these loops, suitable accelerators are identified and loaded into the reconfigurable fabric of the i-Core. On the memory side, a dynamic cache architecture [82] has been developed that enables the parametrisation and the resource allocation of cache memory resources between cores during run time. It was shown in evaluations that only a slight hardware overhead was needed to enable a dynamic run-time cache architecture and that it is possible to achieve an optimally utilised cache memory [83].

- Project B2 (see Chapter 6) investigated invasive computing on tightly-coupled processor arrays (TCPAs) [26]. These have been shown to provide highly energy-efficient [37, 72] and, at the same time, timing-predictable acceleration [14, 77] for many computationally intensive applications from diverse areas such as scientific computing, and image and signal processing. Predictability of non-functional program properties of execution such as *performance, reliability, and security* (denoted as *-predictability in [77]) was a focal research area of our CRC/Trans-regio research in the second funding phase. In terms of latency and throughput, TCPAs are predictable at the cycle level when executing loop programs in parallel. The investigations were therefore also centred around other non-functional objectives with a particular focus on safe(r)
loop processing, i.e. the investigation of fault-tolerance schemes that become active on-demand and energy reduction in view of the emerging problem of dark silicon. In the latter area, it was shown that TCPAs provide an excellent IP in accelerator-rich MPSoC designs to fight against dark silicon by not only being able to save dynamic, but also static power by powering a claimed TCPA region up only at invade time, see, e.g. [37, 72], and [60]. In case of high reliability requirements, fault-tolerance schemes such as DMR and TMR are needed. Here, to avoid modifying the hardware architecture of TCPAs, it was proposed to implement redundancy only based on the principles of invasion of either (a) a non-redundant, (b) a dual-replicated, or even (c) a triple-replicated array instance copy for computing a parallel loop nest in lock-step mode. Designs on how to achieve this including (a) signal replication of input and output signal streams, (b) voting (hard-wired vs. software-based voting), as well as (c) error detection and recovery techniques (memory and communication error protection hardware) have been published in [45], [74], and [44].

- Project B3 (see Chapter 7) investigated methodologies to maximise the overall system performance under power density and temperature constraints on loosely-coupled MPSoCs. The means of optimisation to achieve such a goal are mostly Dynamic Voltage and Frequency Scaling (DVFS), Dynamic Power Management (DPM) (to decide when to active/deactivate a core), core heterogeneity (different core types have different power and performance characteristics), run-time diagnostics, and task migration. Here, techniques and algorithms were developed based on application and hardware models [55]. Particularly, a novel thermal-aware power budgeting concept called Thermal Safe Power (TSP) was proposed [57, 58] which is an abstraction that provides safe power and power density constraints as a function of the number of active cores. Furthermore, a lightweight method for computing transient temperature peaks, called MatEx [56], was presented. It is based on analytically solving a system of thermal differential equations by using matrix exponentials and linear algebra, instead of using regular numerical methods. On the topic of boosting, an efficient and lightweight run-time boosting technique based on MatEx, called seBoost was presented in [59]. Moreover, a dark-silicon–aware resource management technique [36] has been developed that distributes the chip’s resources (cores and power) among different applications under a thermal constraint in order to maximise the overall system performance, while considering the Thread Level Parallelism (TLP) and the Instruction Level
Parallelism (ILP) of the applications. In collaboration with Project B2 (see Chapter 6), a resource management technique [37] was derived that introduces power density as a system level constraint in order to avoid thermal violations, and then assigns applications to tiles by choosing their degree of parallelism and the voltage/frequency levels of each tile, such that the power density constraint is satisfied. Finally, in collaboration with Project B4 (see Chapter 8) and Project C1 (see Chapter 10), a framework [60] was established for power and performance management and optimisations done by DaSiM (e.g. thermal-aware application mapping, sequential phase-aware boosting).

- Project B4 (see Chapter 8) investigated the monitoring of invasive architectures. Here, the focus was on “fix it before it breaks”. Intelligently using the data of ageing, temperature and power monitors, the system predicts the point in lifetime when a component approaches a hardware failure. A key challenge was identified that ageing [38] impacts every fabricated integrated system differently due to different stress profiles and random process variations during manufacturing [50, 51, 69]. Statistical analysis of the timing behaviour of integrated circuits is valuable for such analysis [47, 48]. Ageing, process variations, varying environmental operating conditions as well as different workloads will result in different timing margins. As a result, different fabricated systems may fail at different points in time. Yet, using post-silicon tunable clock buffers [49, 91, 92, 93], timing margins on fabricated systems can be equalised. The monitoring features were demonstrated with an enhanced FPGA monitor demonstrator [17, 18, 19]. It has been shown how they can be used for system-level decision making [60].

- Project B5 (see Chapter 9) investigated invasive Networks-on-Chip. The developed iNoC provides the communication infrastructure backbone between compute tiles, enabling and supporting invasive concepts from the software level down to hardware realisation. Enabler of predictable bandwidth on links of a NoC were the introduction of virtual channels that may be invaded by an application on demand and that provide not only an analysable Quality-of-Service [14, 31, 32]. This feature is heavily used in real-world real-time predictable distributed control applications and has been showcased on the proFPGA prototyping platform. Another focus was on fault-tolerant communication realised by transparent hardware mechanisms [28, 30, 46] and mapping techniques for the secure mapping of tasks, see, e.g. [86]. Furthermore, the distributed memory architecture of invasive multi-tile architectures was also investigated
1.3 Achievements in First and Second Funding Phases (2010–2018)

and improved by enabling applications to efficiently scale beyond tile borders through a novel *region-based cache coherence* (RBCC) scheme. It allows the user to set-up/tear-down coherency regions dynamically based on application requirements. RBCC introduces a hardware coherency region manager module that provides scalable coherence for large MPSoCs by reducing administrative overheads compared to other global coherence schemes [54]. RBCC provides scalable coherence for large MPSoCs by reducing administrative overheads compared to other global coherence schemes.

C. Compiler, simulation and run-time support The main achievements of project area C may be summarised as follows.

- Project C1 (see Chapter 10) investigated the invasive Run-Time Support System (iRTSS). It consists of two major building blocks: Agent system (AS) and OctoPOS. In this setting, the AS takes over the strategic function of global resource management and OctoPOS provides the mechanisms required for the execution of application programs for a given resource allocation. Investigated was a new online learning scheme [39] that allows to automatically adapt application speedup curves at run time. Speedup curves were used for resource allocation and now allow to reflect the current system load, as this affects the application scalability. For applications that demand guaranteed service, support for a new constraint system that is used by the design space exploration tool of Project A4 (see Chapter 3) has been developed. In addition to managing compute cores, this also comprises invasion of infrastructure, e.g. communication resources [32]. The project also provided support for special *pre-invade* requests for Project D1 (see Chapter 13) and detailed interaction with the dark-silicon management of Project B3 (see Chapter 7) [60]. In addition to providing considerable engineering support in the integration process of an FPGA-based demonstrator platform, OctoPOS has been extended by functions for dynamic reconfiguration of fundamental kernel-level components. A key aspect here was to provide on-the-fly program isolation by dynamically enabling and disabling of memory protection and, thus, MMU-based partial virtualisation at run time [13]. Depending on the constraints given with a particular application, OctoPOS changes its mode of operation at run time in order to enable Project C5 (see Chapter 12) to deal with the trade-off between memory protection (security) and strong predictability (safety). Coordination of all necessary cooperation and communication in this context is a low-latency operation and largely
benefits from non-blocking and, partly, wait-free synchronisation such as guarded sections [12].

- Project C3 (see Chapter 11) investigated compilation techniques for invasive computing architectures. Its central role was the development of a compiler framework to support invasive computing including code generation as well as program transformations and optimisations for a wide range of heterogeneous invasive architectures including RISC cores, TCPAs (tightly-coupled processor arrays) and the i-Core.

Regarding compilation and code generation for TCPAs (see Project B2, Chapter 6), investigated were techniques to (a) significantly reduce the code size of loop programs and to (b) symbolically parallelise nested loop programs. On one hand, the results of (a) are necessary because the cores of a TCPA provide only tiny program memories and our approach allows to keep the length of the generated assembly code sequences independent of the length of the loop schedule (latency) and problem sizes (loop bounds and TCPA array size). On the other hand, the results of (b) are essential for invasive programming on MPSoCs because the claimed region of processors, whose shape and size determines how to tile the iteration space of a loop during parallelisation, is not known until run time. The award-winning breakthrough result in [80] that the set of latency-optimal schedule functions—parametric in the sizes of a claimed processor array region—may be determined statically has been extended to multi-level (hierarchical) tiling, see, e.g. [73], and even to allow for the pipelined execution of loop iterations (symbolic software pipelining), see [89].

Regarding the code generation for RISC targets, the project focused on verification and optimisation, both on the level of the intermediate representation (IR) and in the language run-time. It was proven that the greedy code generation approach for exploiting permutation instructions, which was proposed in our early research in cooperation with Project B1 (see Chapter 5), is indeed optimal [6]. Moreover, the correctness of a SSA construction algorithm was formally verified in [5]. The machine-checked formalisation uses the theorem prover Isabelle/HOL and also proves quality guarantees: the algorithm always constructs pruned SSA form, and minimal SSA form in case of reducible control flow graphs. As it allows the application of local optimisation rules during construction of the IR, a generator for finding correct local optimisations has been proposed in [4]. The developed tool Optgen enumerates all local optimisations up to a given pattern size and verifies them using
an SMT solver, thereby guaranteeing completeness and correctness of the generated rule set. On the level of the language run-time, it was shown in collaboration with Project C1 [53] a user-level scheduler for X10 activities is not needed due to the execution and isolation model of OctoPOS. Furthermore, a novel technique to avoid serialisation when copying pointered data structures between shared memory partitions on non-cache-coherent architectures has been proposed in [54]. In a joint effort between our two groups, we have shown in [90] the benefits of invasive computing for providing fault tolerance on demand; see also the article [46] of a special issue on invasive computing.

- Project C5 (see Chapter 12) explored security aspects of invasive computing and resource-aware programming with focus on ensuring confidentiality, integrity and availability of the invasive computing system in the presence of untrustworthy programs that compete for resources and can contain malicious functionality, thereby closing a gap in the system architecture. The project started in the second funding phase in which the groundwork for security concepts had to be laid and the project began to devise an attacker model and define specific security properties for invasive applications. Specific to invasive computing, the attacker model comprises an absent or untrusted system layer accessing shared resources. On a requirements level, the new concept of \( \varepsilon \)-confidentiality [11] is of particular interest as it expresses a desired bound on the amount of information leakage of at most \( \varepsilon \) bits per second from within the application. The system satisfies \( \varepsilon \)-confidentiality if the strongest known attack leaks less than \( \varepsilon \) bits per second. The requirement therefore depends on context (known attacks) but otherwise is agnostic towards the meaning of leaked bits.

The project then devised several new mechanisms that achieved different levels of isolation at different architectural layers against attacker models with different assumption coverage: at the hardware layer [11, 21] (i.e. isolating applications running on the same core against system level attackers in collaboration with the Mercator Fellow), systems software layer [22] (isolating memory abstractions against application level attackers in collaboration with Project C1, see Chapter 10), and the application layer [11] (integrate security requirements as constraints into the invade phase of an invasive program to protect against X10-level attackers in cooperation with Project C3, see Chapter 11. The memory isolation concept was integrated in a robotics demonstrator together with Project D1 (Chapter 13). In collaboration with Project A1
(Chapter 2), Project A4 (Chapter 3), and Project B5 (Chapter 9), a hybrid mapping methodology was proposed that attempts to ensure spatial isolation by mutually-exclusively allocating resources to applications in an MPSoC [86].

D. Applications  The main achievements of project area D may be summarised as follows.

- The main research topic of Project D1 (Chapter 13) was the exploration of benefits and limitations of invasive computing in humanoid robotics. This topic was addressed by analysing the influence of this new paradigm on different levels of a humanoid robot architecture. Here, invasive computing techniques were analysed on both the OctoPOS operating system specific for invasive computing systems and on the state-of-the-art robotic software framework ArmarX. Based on this approach, the project gained insight on both low-level invasive application chains as well as on high-level robot actions. Investigated was a chain of robotic vision algorithms, including Harris Corner detection, SIFT feature extraction and SIFT feature matching [61, 62]. In a study on an Intel Xeon server with 4 sockets and 96 virtual cores running OctoPOS, invasive resource reservation has enabled to adapt the execution of real-time vision not only to the available resources but also to varying workload scenarios [88]. Within the robotic software framework ArmarX, in which high-level robot actions are represented as hierarchical finite state machines as described in [84], it was shown that future robot actions and their associated resource usage can be predicted based on statistical models which are learned from sensorimotor experience and resource usage profiling information [40, 42]. Additionally, a novel resource-aware algorithm for collision free motion planning was developed, which is capable of adjusting its internal degree of parallelism to the difficulty of the planning problem and the currently available resources [41].

- Project D3 (Chapter 14) investigated the potential of the invasive paradigm for high-performance computing (HPC), i.e. when used for large-scale HPC applications, on the respective systems and with the standard programming models. Dynamic resource allocation is a big and unresolved issue in the HPC context. The target application class are simulations with an obvious need for dynamic resource allocation.

Having put the focus on shared-memory systems in the first funding phase, with the development of an extension of OpenMP and the demonstration of its feasibility as the most important outcome, the
Project D3 targeted distributed-memory systems in the second funding phase. A minimal set of new MPI operations, such as MPI_Init_adapt, MPI_Comm_adapt_begin, and MPI_Comm_adapt_commit, were designed to enable efficient resource adaptations in distributed-memory software. These were implemented in the novel iMPI library with its supporting infrastructure [10]. To demonstrate the benefits of iMPI, the application focus was widened towards larger and malleable distributed-memory applications (2.5D, tsunami, shallow water, and porous media simulations [33, 70]). The potential of invasive applications in the context of dark silicon was studied and presented at the Dagstuhl Seminar [15] organised by this CRC. For the FPGA-based demonstrator platform, an invasive version of malleable distributed-memory fluid dynamics (CFD) simulation was implemented in the X10 language.

1.4 Assessment of Project Structure

Invasive computing is a novel paradigm for parallel processing. In order to exploit all its enormous benefits fully, as well new programming as architectural issues of manycore computing architectures need to be jointly solved. Therefore, the CRC required a wide range of expertise that goes far beyond pure computer architecture, pure programming, and pure algorithmic issues of parallel computing. In fact, one of the key novelties in invasive computing is that both hardware and software synergistically adapt to each other in order to deliver a high degree of efficiency in manycore computing, and/or to enable non-functional program execution qualities to be enforced. As such, the expertise needed to build an invasive computing system spans from algorithms through compilation and operating systems to diverse architectures like embedded and high performance as well as to necessary principles like run-time adaptation and hardware reconfiguration. It became soon clear that one site alone could not cover all areas of expertise in an excellent way. The CRC with its three sites (FAU, KIT, TUM) involved was the way to go for our mission. All the areas of expertise have been covered with the top experts in their respective fields. Also when looking at the focus on predictability of non-functional program execution qualities in our second funding phase, and on run-time quality enforcement of these qualities in corridors in view of uncertainties of input, machine and environment states, this focused interdisciplinary composition of expertise is a must but also unique, not only in Germany, but world-wide!

In this respect, the three sites Erlangen, Karlsruhe, and Munich offered a unique combination of key competences also for the third funding phase.
with the following competencies necessary for the successful continuation: (1) Erlangen: (i) massively parallel processor co-design and fault tolerance (Teich), (ii) energetic system software and operating systems (Hönig, Schröder-Preikschat), (iii) multi-objective optimisation and design space exploration (Wildermann), (iv) security (Freiling). (2) Karlsruhe: (i) adaptive embedded computing (Becker, Henkel), (ii) programming languages and software (Snelling), (iii) low-power design (Henkel), (iv) humanoid robotics (Asfour). (3) Munich: (i) network processors and on-chip communication (Herkersdorf), (ii) parallel algorithms and design patterns (Bader), (iii) monitoring, reliability, and technology abstraction (Schlichtmann), (iv) high performance computing (Bungartz, Gerndt). (4) Bremen: (i) scheduling with uncertainty (Megow).

The project structure of the second funding phase was continued as has been shown to be successful and by maintaining the above four project areas A, B, C, and D. Moreover, all projects except of Project C2 continued, yet some with a slight change in title, all with a new focus on the challenges of our third mission, and composition. Moreover, project area A was enriched by a new project, Project A5 (see Chapter 4). For this project, we were able to win Nicole Megow from University of Bremen (previously TUM) as a true expert and leader in the area of scheduling with uncertainties. Her expertise was very important for our investigations on run-time enforcement in view of uncertainties in program input, machine and environment states. Moreover, we continued our work in the area of security on demand on multicore platforms in Project C5 (Chapter 12) with external support by Ingrid Verbauwhede from KU Leuven as a Mercator Fellow. Her interaction and involvement was very successful in bringing together software and hardware expertise for security on manycore systems. In Project A4 (Chapter 3), we obtained novel contributions on fundamentals of design time characterisation and multi-objective optimisation as well as invasive parallel program pattern analysis with the combination of Stefan Wildermann and Michael Bader. Concerning the projects, only Project C2 was discontinued. This important project has created among others very successful and fast simulation technology for invasive programmers with InvadeSIM [66, 68]. This framework allowed us to model architecture templates and to run invasive programs on these architectures virtually. In the third funding phase, this framework has been extended by methods to simulate in particular the new concepts of power and timing enforcers and to study run-time monitoring and enforcement techniques from the scratch. Yet, as this support in new models and functionality of the simulation engines are rather technical than basic research, we decided to move these
new important but mainly engineering efforts to our demonstration project Project Z2 (Chapter 15).

1.5 Research Focus in the Third Funding Phase (2018–2022)

1.5.1 Goals and Strategy

Our common global and interdisciplinary research questions on invasive computing of the third funding phase (the recurrent themes) have manifested themselves as follows. The chapters of this book concentrate mainly on these research questions and solutions developed within this phase.

1. **Exploitation of existing knowledge** for the resource-aware operation of heterogeneous invasive manycore computing systems. This included the analysis of important program and system properties at compile time and in combination with run-time decision making,

2. **Beating the uncertainty**: Even if the isolation of resources through temporally invasive (exclusive) usage of resources by a program helps a lot to analyse non-functional program properties interference-free from other programs, composability alone does not help to reduce still observable large fluctuations in execution time, power consumption and other properties of programs on multicore systems. These remaining uncertainties are mainly due to variations of workload (e.g. program input), as well as machine (e.g. power manager) and environmental states.

3. **Run-Time Requirement Enforcement (RRE)**: As increasing the predictability of non-functional qualities of program execution is a must in order to use manycore technology in the multi-billion dollar market of embedded and cyber-physical systems, we started to introduce requirement corridors on non-functional properties of execution at the programmer’s level. Yet, these properties must be continuously monitored and enforced at run time during execution (infect) of the program on the claimed resources. Here, hybrid techniques combining program-specific static analysis (SA) with run-time enforcement (RRE) are needed. We studied how to generate and implement such program-specific enforcers that need to nest in between the application program and the hardware and OS before execution in order to either completely avoid (strict enforcement) or counterreact (loose enforcement) to any violation of a given quality requirement. For strict enforcement, obviously model-based formal verification techniques were needed to be devised.
that are able to prove that a generated program-specific quality enforcer will never leave a given requirement corridor. For loose enforcement, techniques such as run-time verification that continuously monitor the execution of a program seemed to be the right choice. These allow for a spectrum of techniques to be investigated on how to prevent or react to any violation.

4. Robust multi-objective optimisation and design space exploration techniques to analyse not only timing and timeliness, but also the aspects of security, and power, respectively energy constraints for as well special-purpose as universal computing systems. A special focus here was in analysing the robustness of explored solutions with respect to workload and machine state variations.

According to the above classification of challenges related to algorithms, notation, language design and extensions as well as architectural research, the project structure and its four project areas A, B, C, and D were kept.

In the following, we summarise the central research problems for each of these four focal research domains that are central and follow the four manifested research goals quite closely. The corresponding solutions to these goals and research questions are then described in detail throughout the individual book chapters.

Figure 12 depicts the four major project areas (vertical axis), projects and their relations. The horizontal axis represents the granularity of invasive computations. Remember that the principle of invasion is applicable on all levels of computational granularity, ranging from simple ALU type of operations to complex HPC jobs. As it can be seen in the figure, each project contributes to a specific subset of levels. The only exception are the cross-cutting topics treated in project area A, which are dealing with fundamentals of invasive computing.

In the following, the major research topics, related questions answered and a short description of the projects that tackled these problems is given.

1.5.2 Research Focus per Project Area

A. Fundamentals, language and algorithm research The main topics of investigation of project area A included the investigation of basic techniques for handling uncertainties of non-functional qualities of program execution of invasive parallel programs and reducing the variability of these fluctuations through proper hybrid (mixed static/run-time) techniques to
control such qualities. Apart from program-specific basic run-time enforcer techniques and their generation and implementation (Project A1, see Chapter 2), scenario-based transitions between operating point in explored design spaces of mappings were considered in Project A4 (Chapter 3) as well as novel scheduling techniques that can deal with the uncertainty of workload (e.g. program input) and machine state to provide predictable execution times (Project A5, see Chapter 4). In particular, the three involved projects investigated the following issues:

- Project A1 (see Chapter 2): The static invasion of resources may provide desired bounded execution qualities (predictability). Yet the price may still be high, as isolation alone may not reduce undesired jitters, e.g. in execution times, typically caused by uncertainty stemming from unknown or varying input and environmental execution conditions. Consequently, resources may be underutilised if claims are composed to reflect the worst case of input and environment. As a remedy, we investigated by which techniques requirements may be enforced at run time. This focal CRC topic is called Run-Time Requirement Enforcement (RRE) [76]. Here, Project A1 investigated novel centralised as well as distributed RRE techniques that are based on sub-corridor partitioning. In the first, a central control instance is generated to enforce the desired non-functional qualities of execution based on statically analysed knowledge about the criticality and thus importance of certain
design decisions such as computational requirements of an actor, etc. In the latter, requirement ranges were identified and statically partitioned into intervals called corridors. Moreover, in the case of hard requirements, techniques for strict enforcement were investigated. Here, sound formal proof techniques have been derived at compile time that an RRE technique will control a given program quality (e.g. by DVFS) to never leave a given requirement corridor. An example of such a requirement to be verified is a hard execution deadline of an application graph as specified in ActorX10. Other important non-functional properties for which bounds were formally proven include memory requirements (tile-local and global memory), communication times, and load. In the case of soft requirements, temporal violations might be tolerable. Here, multi-objective control theory with the control laws derived from an analysis of the importances of tasks and their mapping on the quality under control do provide efficient enforcement solutions. Also, probabilistic properties of enforcer automata were formulated and formally verified for loose requirement enforcement.

- Project A4 (see Chapter 3) focused on programming and characterising applications with highly-varying execution phases and changing workloads on hardware (called scenarios in the following) as well as faults. The main goal was to explore optimised yet also robust application executions by dynamically adapting the application behaviour and structure (i.e. the actor model) as well as the set of allocated resources by means of re-invasion in reaction to program workload variations as well as fault scenarios. Moreover, new algorithmic patterns typical for applications considered in project area D were investigated, particularly focusing on different options to improve fault tolerance: in HPC, respective options range from simple checkpoint-restart up to numerics- and physics-based a-posteriori fault detection (and recomputation); for robot vision algorithms, the project focused on redundancy approaches as well as re-invasion of fault-free resources and migration of actors. On the other hand, novel scenario-based design space exploration techniques were investigated to determine robust operating points for each scenario of a complex application. Here, a combination of learning techniques and simulative evaluation were used to determine algorithm and resource settings with robust execution qualities.

- Project A5 (see Chapter 4) explored algorithms and analytic techniques for scheduling and resource management with uncertain input. Here, algorithms were developed that can handle, e.g. uncertain task sets
and uncertain task execution times while giving provable performance guarantees regarding non-functional properties. The newly proposed project was of foundational character and aimed for theoretical guarantees by building on methods from algorithms theory and mathematical optimisation. Its main focus lied on provable worst-case guarantees. When predictability is crucial, e.g. in safety-critical applications, most multicore systems still rely on single-core usage. The reason is that the current state-of-the-art approaches in real-time scheduling do not capture the difficulties in scheduling parallel workloads in a predictable way. In this project, algorithms were developed with guarantees on predictability and resource utilisation as is required to exploit the full power of parallel computing and particularly the benefits of invasive computing—also for safety-critical applications. The uncertainties that were considered include unknown task sets, unknown arrival and task execution times. In cooperation with Project A1 (Chapter 2), it was shown that execution time variations of tasks may be compensated to enforce execution time requirements by not only varying the schedule, but also the speed of the involved processors dynamically.

B. Architectural research The main topics of investigation of project area B may be summarised as follows: All projects took up the CRC challenge of minimising the variability of non-functional program requirements such as on the latency $L$, power $P$, or reliability $R$, respectively of enforcing these at run time for the different hardware resources as investigated ranging from $i$-Core to TCPA and NoC. In particular, the five involved projects investigated the following issues:

- The main focus of Project B1 (see Chapter 5) was in the realm of the big picture: Run-time enforcement of WCETs for hard real-time applications on the $i$-Core in view of interference stemming from best-effort applications and on security aspects on other LEON cores on the same tile. For the first problem area, the interference of best-effort applications sharing an $i$-Core tile were to be monitored and properly controlled such that static WCET analysis will always be valid for the hard real-time applications. Moreover, the concept of special instructions (SIs) has been generalised to include also complex control flow instructions. Concerning the memory hierarchy reaching its limits, a novel network in a network structure has been developed in collaboration with Project B5 (see Chapter 9) to provide dedicated access from the network adapter to the TLM and from the cores to the TLM. Finally, building upon the results of Project C5 (see Chapter 12), isolation concepts within the $i$-Core
were leveraged to dynamically enforce security requirements such as data integrity and bounded information leakage.

- Run-time enforcement of non-functional execution qualities for parallel loop programs when executed on TCPAs was in the focus of the research of Project B2 (see Chapter 6). In order to enforce a given set of non-functional requirements of a loop nest when executed in parallel on an invasive TCPA, an overprovisioning of resources (the invaded region of TCPA processors) shall be greatly avoided. To do so, completely novel techniques were needed to be investigated summarised as (a) **Self-invasion** of claim sizes of latency-bound programs. With self-invasion, the system (rather than currently the programmer) computes and executes the invasion such to neither overprovision nor underutilise TCPA resources. (b) **Self-Power Adjustment**: Alternatively or in combination with the above run-time computation of a required minimal claim size to be invaded for satisfying requirements on latency $L$, also hardware concepts for auto-adjustment of the power $P$ to stay in a desired corridor or to minimise the needed power were investigated, e.g. by determining the voltage/frequency of a claimed processor region at run time. (c) **Self-selection of redundancy scheme**: Instead of choosing a replica scheme such as TMR and DMR upon worst case environmental assumptions, a run-time enforcement of such properties was achieved by autonomous selection of a proper scheme. Other investigations included the exploitation of **approximate loop computing** on TCPAs in order to stay within execution time bounds or to save energy and **invasive floating-point** TCPAs that have opened a new dimension of applications.

- The major scientific scope of Project B3 (see Chapter 7) was on machine learning and self-awareness in resource and power management. The use of machine learning is strongly motivated for that class of applications whose execution behaviour can be very hard to analyse or predict statically (i.e. before run time). Related to the CRC topic of **run-time verification**, the project has built upon existing supervised learning techniques to derive power and thermal models for our invasive computing architecture components. Particularly, calibration runs were conducted for applications with known characteristics and a distributed thermal modelling approach based on supervised learning was applied. In the case multiple (best-effort) applications are not running in isolation on the same tile/chip, a statically analysed performance of such an application might not be correct. Here, historically measured data can be fed to unsupervised machine learning algorithms in order to derive power,
1.5 Research Focus in the Third Funding Phase (2018–2022)

Performance, and/or temperature profiles for the executed applications at run time. Finally, appropriate system state conditions, sensor event sequences as well as spatial and temporal correlations among them were investigated in view of an invasive diagnosis on chip (iDoC).

- Project B4 (see Chapter 8) extended the existing functionality of the monitoring system to support run-time verification of non-functional program properties, especially their timely execution. Many programs have variable non-functional properties, which depend on the input data and the system state during execution. Run-time verification addresses this challenge by checking the non-functional properties—as the name implies—at run time. A so-called verdict is supplied that indicates if the properties are within predefined constraints. As long as the verdict is positive, the system can execute its invasive application with high resource utilisation employing smaller margins. If the verdict is negative, the system can take appropriate actions to allow safe operation (see Project A1 (Chapter 2) for techniques of enforcement). Run-time verification was already implemented for various safety-critical systems. The major challenge addressed in this project has been the application of run-time verification for highly complex and dynamic invasive architectures and how to specify and generate such program and property-specific monitor constellations.

- Project B5 (see Chapter 9) aimed at techniques to dynamically optimise the interconnect infrastructure and the memory hierarchy of invasive multi-tile architectures. Investigated were techniques to dynamically reconfigure the iNoC at different levels at run time based on changing requirements and constraints. Examples of such reconfigurations range from simple changes in routing and flow control to more complex changes like complete replacement of certain network layers. These flexibilities lead to different operating points that need to be considered already during design-time characterisation in Project A4 (Chapter 3). Another objective was to move computations closer to data. This was achieved via two complementary methods: In-memory computing and data migration. For in-memory computing, conventional memory controllers needed to be enhanced by either software-programmable or additional processing elements acting as programming in memory (PIM) extensions. These extensions can be either tightly coupled to or integrated in the memory controller. This allows simple yet efficient operations like array initialisation, data increment/decrement, pattern search, etc., thereby alleviating costly memory transactions. With hard-
ware assisted data migration, data can be brought closer to the cores in order to minimise remote access latencies.

C. Compiler, simulation and run-time support

The main topics of investigation of project area C with its three involved projects may be summarised as follows:

- Project C1 (see Chapter 10) focused on the problem of enforcement of quality criteria of mixed criticality as to timing and energy consumption by means of application-oriented resource allocation strategies as well as mechanisms at iRTSS level. Beyond that, already existing and still proposed fundamental concepts, abstractions, and techniques of invasive computing were brought to the domain of state-of-the-art manycore systems in order to prove general applicability and field performance of the solutions created. The major topics of investigation included (1) enforcement of agent system (AS) properties (currently no guarantees may be given on execution time bounds for claim determination and quality of returned claims), (2) hardening the AS against malicious re-invaders, (3) energy-efficient process coordination and virtual shared memory, and (4) control of background noise as to OctoPOS functions that provide transparent access to the different main-memory subsystems of the PGAS model as defined for invasive computing. Measured overheads as to timing and energy consumption are externalised as system parameters (i.e. non-functional properties) in order to be able to give a precise prognosis on the effectiveness of a particular i-let schedule. The investigations targeted small-, medium-, and large-scale multicore systems of different processor and memory architecture to obtain a broad parameter set for real machines suitable for the execution of invasive programs.

- Regarding the focal theme of run-time enforcement of multiple non-functional execution qualities, such as user-specified performance and energy corridors—the following research problems were investigated in Project C3 (see Chapter 11) for TCPA targets: (1) constant latency/-throughput loop processing. In order to achieve this, either the invaded claim may be adjusted at invade- or infect-time. Alternatively, loop iterations may be skipped (e.g. in the case of any-time algorithms), (2) approximate loop processing. Here, language extensions were necessary to specify imprecise loop variables as well as symbolic techniques for the analysis of propagation errors based on data-dependence analysis, and (3) techniques for symbolic code generation. A code generator
for TCPA targets has been developed that implements symbolic loop schedules (i.e. assembly code which branch bounds that will materialise at run time only). For i-Core targets, the semantics of special instruction (SI) specifications were extracted from the source program and use program synthesis techniques to generate more efficient SIs. Finally, programs were analysed for parallel programming patterns that can be transformed into invasive resource-aware variants. Interesting and worthwhile parallel programming patterns were devised in cooperation with Project A1 (Chapter 2) and Project A4 (Chapter 3).

- Project C5s (see Chapter 12) major objectives of research were to extend intra-tile to inter-tile security and to provide mechanisms that can reliably and provably enforce security properties that have been requested by applications at run time even if attacker assumptions are violated, thereby increasing the assumption coverage. The major topics of investigation were: (1) combining control flow attestation and information flow control for invasive applications in order to guarantee confidentiality and integrity; (2) As confidentiality and integrity protection for inter-tile memory is currently guaranteed through MPU access control, on-the-fly memory encryption was investigated. Here, memory encryption was proposed where remote pages will be transparently de-/encrypted on inter-tile access and their integrity checked. (3) Run-time monitoring and destructive preemption: Leverages are the distributed monitoring system to test for the existence of potential side channels at run time. While in general it is impossible to reliably detect information leaks, the fact is exploited that the attacker must move first and devise an approach that can dynamically choose the monitors, encodings and statistical tests that it performs. Devised were a set of side channel detection tests for known channel encodings and attacks. In order to re-enforce a breached security property, actions must be taken. Here, in cooperation with the Mercator Fellow Ingrid Verbauwhede, each tile has been equipped with a minimal hardware extension capable of measuring available code and data and support attestation to neighbouring tiles. These tile-to-tile attestations include action definitions in their responses which enforce recovery from failed measurements.

D. Applications  The main topics of investigation of project area D with its two involved projects may be summarised as follows:

- The two main topics of research of Project D1 (see Chapter 13) were: (1) Invasive robot architecture based on speculative resource management
(KIT): Here, the already developed building blocks required for speculative resource management towards a complete invasive architecture for robotics were built upon. In order to address parallelism in robot actions for the first time, the existing statistical prediction models needed to be extended to use the results of a multi-objective optimisation as input for deciding about potential executions of parallel actions. Objectives that were considered are resource utilisation, execution time, and minimisation of robot movements. Making informed choices for speculative resource management additionally required the ability to anticipate the consequences of future robot actions. The accuracy of expected future world states of the robot directly influences the accuracy of robot action and resource requirement predictions. Introduced was an active perception heuristics for predicting the robot’s belief state of its environment. Active perception itself consists of exploring and manipulating objects in order to increase the knowledge of the objects and their surroundings. Current approaches focus on maximising information gain and certainty of the belief state by changing the exploration to manipulation ratio but do not take resource utilisation into account. (2) Porting robotic legacy code to invasive multicores (TUM): In order to re-use legacy code as best as possible in robot applications, a wrapper-based methodology has been developed consisting of (a) profiling of legacy code on legacy platform, (b) design space exploration on the target platform, in order to identify potential operating points, (c) shared resources partitioning, based on the application-specific access patterns, (d) establishing and configuring an invasion wrapper around the legacy applications in order to isolate access to shared resources, and (e) run-time switching between operating points, in order to enforce end-to-end latency guarantees of legacy code on invasive multicore platforms.

- Project D3 (see Chapter 14) built on the shared and distributed memory developments and performed the consequent next step: turn to large-scale hybrid systems, as they are state-of-the-art in current supercomputing centres. Hence, the overarching goal was to show that invasive computing can make a difference in HPC—not only for accelerating an individual application code, but also in terms of the daily practice of operation of large systems at supercomputing centres. In that context, two major concerns were power consumption, in particular the lack of predictability, and higher failure rates due to growing core numbers. Therefore, two first work addressed the topics of invasive power corridor management (with a focus on using the elasticity of iMPI applications to guarantee lower bounds on energy consumption) and invasive appli-
cation recovery (highly scalable, faster, and less intrusive than classical storage-based checkpointing). As a third research thread, the project has dealt with the deployment of novel manycore architectures, which increase especially the complexity of efficiently handling the memory hierarchy. Invasion of memory layers with guidance by the application, based on the OctoPOS API, allowing for improved execution. Fourth, on the application side, the project focus was on demonstrating the feasibility of invasive computing. For that, based on the experiences gained so far, the invasification of selected real application codes such as the seismic code SeisSol and the molecular dynamics code lsl mardyn was tackled. Moreover, appropriate classifications of applications and programming abstractions do help application programmers.

1.5.3 InvasIC Validation and Demonstrator Road Map

The general goal of Project Z2 (see Chapter 15) was to provide common demonstrator platforms to the CRC/Transregio for validating and showcasing the principles of invasive computing on real hardware across all involved layers, ranging from the hardware and system software to invasive applications. Project Z2 has been successfully providing this service at each site for the first two funding phases by operating appropriate multi-FPGA prototyping systems. Without the technical and engineering support of Project Z2, we would have struggled in our endeavour of co-designing multiple types of invasive heterogeneous MPSoC architectures and prototyping the real-life demonstrators that reflect the research merits of all project areas, and are more than the sum of the individual parts.

Project Z2 continued in providing and maintaining the infrastructure of demonstrator platforms and support the integration of a considerable bunch of new hardware/software components that are necessary to evaluate our new methods and results also during the third funding phase. This included not only the physical and logical integration into a common system but also debugging support for new functionality such as run-time monitoring and requirement enforcement. Past experience has shown that introducing new components into an invasive manycore architecture, which is a complex hardware/software system developed by different groups, leads to new errors or exposes already present errors in the interplay of components. These errors may occur very sporadically and only under specific conditions, which makes tracking down and eliminating them a difficult and yet important task to get demonstrators up and running. The elimination of errors is vital for getting stable demonstrators and required a significant effort for the Project Z2 staff.
Figure 13: Road map for demonstrators during the different phases of the CRC/Transregio.

In addition, Project Z2 provided generic components that helped to showcase the enforcement of non-functional properties, to verify properties and constraints at run time as well as multi-objective optimisation as the major topics of the CRC/Transregio in the third funding phase. In this realm, the invasive architecture template was extended by opportunities for introspection that make system internal information accessible to the outside world, e.g. for appropriate visualisations. Examples of such support functions include the non-intrusive measurement of timing and latency of i-lets and the emulation of appropriate power models for components of the prototyped invasive architectures.

Finally, as can be seen in Figure 13, Project B2 has taken the opportunity to launch a TCPA chip design called Alpaca.
Apart from our FPGA-based prototyping platform, the simulator InvadeSIM [66, 68] has whitnessed to be a very important operative instrument during the first two funding phases, especially for theory verification and application development in project areas A and D, respectively. As such, InvadeSIM was used as a testbed for actor-oriented programming and served as the basis for testing of design space exploration and mapping techniques. InvadeSIM was also central, heavily used and advanced in the third funding phase for supporting the major research topics of the third funding phase, i.e. to early evaluate techniques for run-time enforcement and verification, invasive algorithmic patterns and scheduling strategies as have been investigated in project area A.

Finally, beside FPGA- and simulation-based demonstrators, also other emerging massively parallel accelerator platforms were investigated for porting the foundations of invasive computing to manycore custom hardware and to link invasive architectures closer to such general-purpose systems, respectively. The latter was achieved by considering recent Intel Xeon Broadwell-EP/Arria 10 FPGA systems that allow a very tight linking of Intel cores and custom hardware (programmable logic) and have very high memory bandwidth. In addition, an Intel Xeon Phi cluster (Knights Landing/Hill) has been acquired that allows to pattern the memory hierarchy as given with the LEON3-based demonstrator system as well as to study energetic virtual shared memory (Project C1). Another acquired target platform is a 192-core system comprised of ARMv8.2 RISC processors, which has been used for studying high-performance embedded parallel operating systems with predictable run-time behaviour as developed in Project C1. Furthermore, both aforementioned platforms were used to explore the energy-efficient execution of invasive workloads.

Still, our basic research road map required our full attention to verify and showcase the enforceability of non-functional program execution requirements, such as timeliness, safety aspects (e.g. fault tolerance), and security on demand of a program or application on an MPSoC platform. Without invasive computing as a foundational layer to create the required temporal isolation and additional generated run-time enforcers to watch and control the remaining run-time uncertainties during program execution, it will not be possible in the future to apply manycore systems to applications with low allowed or desired variability in the above qualities of concern.

1.5.4 Working Groups – Developing Cross-Project Synergies

Like in previous funding phases, the development of cross-disciplinary and cross-project synergies was considered essential and supported through so-
called working groups. These working groups had been very successful from the very beginning and helped a lot to develop a common understanding of focal research questions in each funding phase and to stimulate interdisciplinary research. Working groups had regular meetings in which challenging discussions took place, and results were immediately reported in a Wiki. We therefore decided to keep working groups as a successful instrument and forum of mutual understanding and progress in solving interdisciplinary questions. A working group works on the solution of such questions, which are not just topics of an individual research project. Their results have been shown to provide vital insights on common understanding and progress in the field of invasive computing. The topics of four working groups (WG1 to WG4) as identified for the third funding phase under the direction of at least two PIs coming from different sites and also field of expertise is described in the following.

**WG1 Run-time requirement monitoring & enforcement**

Focal points of interdisciplinary investigation in this working group were questions around the lead topic of run-time requirement monitoring (RRM) & enforcement (RRE) of non-functional aspects of computation. The topics of this working group stemmed from the fact that the reduction of variability of such aspects is an all-encompassing concept that requires consideration across architecture, system software and services, up to the level of applications. Here, WG1 served as a discussion group for these topics, organised information exchange, and triggered and coordinated collaborations between projects and project areas with respect to analysis of uncertainty and variability and their reduction, respectively guidance within programmer-specified requirement corridors.

Concrete goals of this working group were: (1) Identification of the most relevant topics concerning unpredictability and solutions for robustness within the CRC and the assignment of topic chairs that coordinate individual topics. Concrete subtopics included the analysis of uncertainty and robustness, models for the variability of non-functional aspects due to workload (program input) and resource state variation such as resource management and techniques for enforcing the satisfaction of requirements through all layers of invasive architectures. The latter included formal techniques for the verification of enforcers as well as techniques for run-time verification. (2) A glossary of all relevant terms to enhance the common understanding of RRM and RRE concepts as developed in this CRC. Here, we particularly addressed a common
understanding and implications of program enforcement in the different disciplines and application domains present in the CRC. (3) A landscape that, based on the identified relevant topics, outlines challenges for run time enforcement of program qualities (that (a) can be solved by employing existing analysis techniques, (b) were tackled within, and (c) may be subject to future research directions, also within the CRC. (4) Organisation of workshops to stimulate RRE-related discussions and collaborations outside the CRC with a special focus on bringing together different communities like OS, scheduling, HPC and embedded systems.

**WG2 Memory models, architecture & management**

Memory access interference caused by concurrent applications and I/O processes execution may evolve to become the dominating performance (and predictability) bottleneck for manycore processor platforms. Hence, a holistic approach to memory organisation, considering the entire memory hierarchy from L1 caches over distributed tile-local SRAM memories to globally shared external SDRAM, had to involve multiple domains of expertise within the invasive computing paradigm. This included programming models and language (i.e. the X10 memory model) implementation as well as the underlying middleware software, operating system, and hardware architecture of the processing platform.

Topics of this working group included questions on caches (invadeable caches, region-based vs. global coherence, cache management, non-cacheable address ranges), memory coherency, strong/weak memory consistency models, memory isolation, DMA via the iNoC, near-memory computing. Predictability of memory accesses was another high priority topic on our agenda with mutual interest among several projects. It was to a high degree also interrelated to aspects of the high-capacity, low-latency, guaranteed services iNoC interconnect infrastructure.

**WG3 Benchmarking and evaluation**

Within all projects, individual results were evaluated with respect to cost, benefit, drawbacks, and limitations of invasive computing. This evaluation was based on a wide range of criteria, such as performance, power consumption, quality, dependability, safety, security, hardware overhead, or resource usage efficiency. The goal of WG3 was to identify and discuss methods for evaluation that are relevant across several projects, and to determine benchmarks—including non-invasive reference scenarios—to illustrate the benefit of invasive computing versus state-of-the-art. WG3 identified suitable test scenarios and applications, starting from a repository of standard benchmarks, running on both
invasive OctoPOS platform and on non-invasive Linux, for example. Specific benchmarks were derived from Project D1 and Project D3 with a break-down of test scenarios from D projects to individual project level. e.g. algorithms from augmented virtual reality were analysed for time vs. power trade-offs on invasive platforms as well as for studying the role and benefit of enforcement techniques to stay within desired bounds of quality such as a guaranteed number of operating hours free of recharge of a mobile MPSoC device. Identified benchmark scenarios also defined algorithmic patterns as have been studied in Project A4. WG3 brought together expertise between projects and fostered their technical exchange and synchronised on a regular basis with the demonstrator Project Z2, enabling evaluation and benchmarking on a common FPGA-based demonstrator platform.

**WG4 Power and thermal aspects**

A limiting factor for high performance has been, is, and will be the power consumption. Investigated were various aspects of this limit:

(a) The power density represents a physical limit as the amount of power that can be dissipated at a certain chip area is limited by the maximum temperature a circuit can stand without the risk of accelerated circuit degradation or even immediate irreversible damage.

(b) The energy efficiency determines how much computation (or communication) can be accomplished with a certain amount of energy. Especially in energy-limited embedded applications, it is the goal to make as much as possible use of a limited amount of energy.

(c) Power and energy under real-time constraints: while reasons (a) and (b) are already hard to accomplish, the problem grows more complex when real-time comes into play. For example, if a real-time task needs to complete at a certain time, boosting might be a preferable means. That, however, will increase peak temperature and put the circuits under non-sustainable high stress. Here, multi-objective optimisation strategies were discussed. In general, a thorough investigation of the trade-offs was a primary goal.

(d) Investigating how various scheduling and allocation algorithms match or can be adapted to invasive computer architectures in order to achieve a high efficiency.
With InvasIC having had various projects with a focus on one or more aspects of power and energy with respect to aspects (a), (b), (c) and (d), WG4 brought these various goals under one umbrella by:

(i) Coordinating these various aspects such that in various phases during execution on an invasive multicore architecture, the applied power and energy means at different components (OS, architecture, application software, etc.) were targeted towards the same goals and contradictory control loops are avoided.

(ii) Developing power and energy models that were used by all projects that dealt with the topic.

(iii) Identifying which aspects of power, energy and temperature analysis and modelling should be addressed at design time and which should be modelled as uncertainties when making scheduling and application mapping decisions (at run time). Of particular interest was the question on how far dynamic resource reservation by means of invasive computing and novel ideas about run-time requirement enforcement and run-time verification can help to reduce such uncertainties.

All of the four WGs regularly organised internal meetings in physical and video-conference form. Also, several thematic sessions were organised at international conferences and workshops to present and discuss the very goals with international experts working on similar topics.

References


1 Introduction to Invasive Computing


1 Introduction to Invasive Computing


1 Introduction to Invasive Computing


A: Fundamentals, Language and Algorithm Research
2 Basics of Invasive Computing

Jürgen Teich, Khalil Esper, Joachim Falk, Behnaz Pourmohseni, Tobias Schwarzer, Stefan Wildermann

Abstract The basics of invasive computing include the conceptualisation and development of the fundamentals of invasive programming, in particular the definition of the basic programming constructs invade, infect, retreat, their semantics and the basic properties of claims, see Chapter 1. On that basis, different concrete parallel programming language implementations were developed. On the basis of X10, InvadeX10 [12] became the first invasive programming language implementation. After such initial work, we concentrated on the attestation of the advantages of a by default exclusive resource reservation through claims delivering *-predictability [23] of non-functional properties of parallel programs with * including properties such as timing, safety and security. Such properties can be achieved by isolation of applications through exclusive resource reservations. This chapter focuses on the third major research topic related to fundamentals of invasive computing: Run-Time Requirement Enforcement (RRE) [25], i.e. reducing remaining fluctuations of non-functional program properties by expression of so-called requirement corridors while avoiding any overreservation or underutilisation of resources. In this chapter, we will first present a classification based on centralised and distributed enforcement techniques as well as provide a definition of strictness of the adherence of such program execution requirements. Based on a case study as a running example, we present methods for run-time enforcement of timing and energy properties using the concept of enforcement automata. These dynamically control desired properties as an advocate of an application program in view of or in reaction to dynamic workload variations. Presented are then formal methods to verify quality properties of such enforcement automata for concrete (based on traces) and stochastic input characterisations using probabilistic model checking [8].

2.1 Run-Time Requirement Enforcement (RRE)

In this section, we establish a theory and the semantics of techniques for RRE, distinguishing centralised and distributed approaches as well as strict and loose RRE techniques in support of hard and soft non-functional requirements, respectively. To implement enforcement strategies for diverse non-functional
properties including execution time and energy consumption of multicore application programs, the concept of enforcement automata is introduced.

To satisfy a set of given requirements as introduced in Chapter 1, the observable intervals of non-functional properties must often be reduced. In general, this can be achieved by techniques such as restricting the range of inputs of a program or using approximate computing. Alternatively, isolation techniques may be applied, such as the use of resource reservation protocols or using invasive computing [24]. Through inter-application isolation, composability can be established, which is essential for an independent analysis of individual applications [1, 13]. Formally, RRE has been defined in [25] as follows:

**Definition 1** (Run-Time Requirement Enforcer (RRE) [25]). A Run-Time Requirement Enforcer (RRE) of a requirement \( r_o(p) = [LB_o, UB_o] \) of a program \( p \) is a control technique to steer a non-functional property \( o \) of \( p \) within a corridor spanned by a lower bound \( LB_o \) and an upper bound \( UB_o \) for each execution of \( p \).

Figure 14 exemplifies Definition 1 for given latency and power requirement corridors of an implementation \( p \) of an application. An RRE is also depicted whose task is to confine the observable latency and power values of \( p \) within the corridors specified by respective requirements. Given an actual (current) input and execution state, the RRE estimates the expected latency and power consumption based on which it takes actions (outgoing arcs of the RRE), e.g. adjusting the number of active cores and/or their voltage/frequency settings with the goal to avoid any violation of the requirements, see, e.g. [19].

![Figure 14: Illustration of RRE for a given latency and a given power requirement (corridors). Upon observed inputs, e.g. monitored latency and/or power values, the RRE changes the state of a claim of processors to execute a program \( p \) to avoid any violating runs in the future, e.g. by adjusting the number of active cores and/or the voltage/frequency settings of the cores.](image-url)
2.2 Taxonomy of Run-Time Requirement Enforcers

According to [23], each requirement of an application can be either soft or hard. In case of a soft requirement, occasional violations are still considered acceptable. In this context, an RRE can be classified as either loose or strict as follows:

**Definition 2 (Loose/Strict RRE [25]).** A Run-Time Requirement Enforcer (RRE) of a requirement \( r_0(p) = [LB_0, UB_0] \) of a program \( p \) is called strict \((\nu = 1)\), if it can be formally proven that no concrete execution of \( p \) will leave the given corridor at run time. It is called loose \((0 \leq \nu < 1)\), if one or multiple consecutive violations of \( o \) are tolerable, with \( \nu \times 100 \) indicating a minimal rate (percentage) of satisfactory executions.

Independent from the above definition, an RRE can be classified as a centralised or a distributed enforcement technique:

**Definition 3 (Centralised/Distributed RRE [25]).** A Run-Time Requirement Enforcer (RRE) of a requirement \( r_0(p) = [LB_0, UB_0] \) of a program \( p \) is called centralised if a single enforcer instance is used to enforce the requirement. It is called distributed in case multiple enforcers are used to jointly enforce the requirement.

For an object detection streaming application from the area of robot vision as illustrated in Figure 15, Figure 16a shows an example of a centralised RRE of a latency requirement. The goal of RRE in this example is to steer the latency of the chain of tasks to be processed per image within a given latency corridor. The application consists of 9 tasks (actors): an image source (ISo) actor to read input images periodically at a constant rate, a greyscale conversion (GS) actor, a Sobel edge detection (ED) actor and a Harris corner detection (HC) actor to determine edges, respectively corners, in an image, a SIFT orientation (SO) actor to achieve invariance to image rotation, a SIFT description (SD) actor to extract the features in an image, a SIFT matching (SM) actor to detect objects in the image based on a previously trained set of object features, and a RANSAC (RS) actor to insert the detected objects into the image which is finally sent out by an image sink (ISi) actor. As a platform, we consider a NoC-based 3 × 3 manycore architecture and map the application’s actors on the architecture as illustrated in Figure 16a and Figure 16b. The latency of actors of the application is monitored by a local so-called Run-Time Requirement Monitor (RRM) instantiated on each of the invaded tiles. A centralised RRE technique is shown in Figure 16a. Here, the RRE receives the monitored timing
2 Basics of Invasive Computing

Figure 15: Object detection streaming application. Each actor (node) corresponds to a task that must be executed once during each execution $k$ of the image processing chain while respecting the data dependencies as shown by the directed edges.

Figure 16: Examples of centralised and distributed RRE techniques for controlling a latency requirement for an object detection video streaming application as shown in Figure 15 [25].

information of the last actor in the chain, i.e. Image Sink (ISi), from the RRM on the respective tile based on which it conducts enforcement decisions. Technically, during the execution of each frame, each RRM monitors the time elapsed for the execution of its local actor(s) for the current image frame $k$ and creates a time stamp that is sent together with the processed frame to the subsequent actor. The completion time of the last actor is then used by the centralised RRE to decide on how to react accordingly for processing the subsequent frames.

Figure 16b illustrates an example of a distributed RRE for the same application. Here, the overall time requirement per image frame is partitioned into sub-corridors (or interval budgets) which are assigned to the invaded tiles. Also, in addition to the RRM, a local RRE is instantiated per tile to enforce its assigned sub-corridor locally. Evidently, distributed RRE benefits from a simpler realisation and scalability in comparison to centralised RRE. Nonetheless, centralised enforcement could better use global information to optimise secondary goals such as energy consumption, as we will show in Section 2.3.
2.3 Application Characterisation and Enforcement Actions

In this section, we first analyse the influence of input variation on latency requirements for the previously introduced object detection application depicted in Figure 15. Subsequently, we propose several means of how to enforce latency requirements on a multicore system.

In the following, we assume that each image frame of the given time-critical application must be processed within a latency upper bound \( UB_L = 115 \text{ ms} \). All evaluations presented in this section are carried out using InvadeSIM [20, 21], a high-level functional simulation framework for multicore/manycore architectures and supporting resource-aware programming. Table 2 provides the average, standard deviation, and overall contribution of each actor’s latency when processing a sequence of 9,149 images stemming from different sources of video streams when each actor is processed in isolation on a single core and constantly running at a maximum frequency. As can be seen in Table 2, the SD and SM actors exhibit the highest degree of input-dependent variation in execution time and also the highest contribution to the overall latency. The remaining actors, on the other hand, do not exhibit a comparable execution time jitter and/or a comparable contribution to the overall application latency across the input space.

As enforcement actions, we use Dynamic Voltage and Frequency Scaling (DVFS) [3, 15, 22, 28] to enforce a latency upper bound \( UB_L = 115 \text{ ms} \) for processing each frame. In Chapter 4, DVFS is also investigated under the term *speed scaling* and presenting algorithms for energy-optimised mapping and scheduling of DAGs to MPSoCs while satisfying a given deadline. Due to the small variation and overall latency contribution of all except the SD and SM actors according to Table 2, we dedicate a time budget of 20 ms to

<table>
<thead>
<tr>
<th>latency index</th>
<th>actor</th>
<th>GS</th>
<th>ED</th>
<th>HC</th>
<th>SO</th>
<th>SD</th>
<th>SM</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>average [ms]</td>
<td></td>
<td>0.21</td>
<td>0.18</td>
<td>1.50</td>
<td>1.79</td>
<td>146.86</td>
<td>21.02</td>
<td>0.01</td>
</tr>
<tr>
<td>std. deviation [ms]</td>
<td></td>
<td>0.09</td>
<td>0.08</td>
<td>0.64</td>
<td>0.80</td>
<td>106.15</td>
<td>15.04</td>
<td>0.03</td>
</tr>
<tr>
<td>overall contribution</td>
<td></td>
<td>0.1%</td>
<td>0.1%</td>
<td>0.9%</td>
<td>1.0%</td>
<td>85.6%</td>
<td>12.3%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>
the other actors altogether, assuming that their cumulative latency per input image does not exceed this budget. This translates into a latency upper bound of $UB_L = 95$ ms for the SD and SM actors. For the explanation of distributed RRE techniques, we further decide to split this bound into two individual latency upper bounds, namely, $UB_L = 80$ ms for SD and $UB_L = 15$ ms for SM.

All RRE techniques presented in the following have the ability to adjust two control knobs prior to processing an image frame: a) the degree of execution parallelism per actor that is adjusted by setting the number $n$ of active cores that process the workload of each actor and b) the power (voltage/frequency) mode $m$ of the core(s) allocated for each actor adjusted through DVFS (for active cores) and power gating (for inactive cores). A tuple $c = (n, m)$ is called a configuration in the following. To this end, each RRE decides on a per input image basis how to distribute the workload of each actor being enforced between one and four cores available per tile according to the mappings shown in Figure 16b. At the same time, it sets the power mode of the cores of each tile to either a power-gated mode (with $f = 0$ and $V_{DD} = 0$) or 20 possible DVFS configurations (with a frequency step size of 0.2 GHz and a maximum frequency of 4 GHz) summarised in Table 3. For both actors under enforcement, SD and SM, we analysed the major source of latency variation according to Table 2 (single core, constant maximal frequency) as stemming from the variability in the number $i$ of features in each image to be processed. Therefore, this number is used as a direct indicator of the input workload to the following RRE strategies. As a figure of merit, we consider the potential energy savings of each RRE strategy in addition to evaluating its capability in enforcing the latency requirement(s).

### Table 3: Voltage/frequency (DVFS) modes of each core.

<table>
<thead>
<tr>
<th>mode</th>
<th>$f(m)$ [GHz]</th>
<th>$V_{DD}(m)$ [V]</th>
<th>mode</th>
<th>$f(m)$ [GHz]</th>
<th>$V_{DD}(m)$ [V]</th>
<th>mode</th>
<th>$f(m)$ [GHz]</th>
<th>$V_{DD}(m)$ [V]</th>
<th>mode</th>
<th>$f(m)$ [GHz]</th>
<th>$V_{DD}(m)$ [V]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>0.5</td>
<td>6</td>
<td>1.2</td>
<td>0.91</td>
<td>11</td>
<td>2.2</td>
<td>1.26</td>
<td>16</td>
<td>3.2</td>
<td>1.58</td>
</tr>
<tr>
<td>2</td>
<td>0.4</td>
<td>0.6</td>
<td>7</td>
<td>1.4</td>
<td>0.98</td>
<td>12</td>
<td>2.4</td>
<td>1.32</td>
<td>17</td>
<td>3.4</td>
<td>1.65</td>
</tr>
<tr>
<td>3</td>
<td>0.6</td>
<td>0.69</td>
<td>8</td>
<td>1.6</td>
<td>1.05</td>
<td>13</td>
<td>2.6</td>
<td>1.39</td>
<td>18</td>
<td>3.6</td>
<td>1.71</td>
</tr>
<tr>
<td>4</td>
<td>0.8</td>
<td>0.77</td>
<td>9</td>
<td>1.8</td>
<td>1.12</td>
<td>14</td>
<td>2.8</td>
<td>1.45</td>
<td>19</td>
<td>3.8</td>
<td>1.78</td>
</tr>
<tr>
<td>5</td>
<td>1.0</td>
<td>0.84</td>
<td>10</td>
<td>2.0</td>
<td>1.19</td>
<td>15</td>
<td>3.0</td>
<td>1.52</td>
<td>20</td>
<td>4.0</td>
<td>1.84</td>
</tr>
</tbody>
</table>

### 2.4 Power, Latency, and Energy Model

Our investigations of enforcement strategies involve the evaluation of power consumption, execution latency, and energy demand per actor under enforcement. To evaluate the power consumption $P(m)$ of a core in power mode
m, we use Eq. (9) in which the first summand represents the dynamic power contribution calculated based on the effective switching capacitance $C_{\text{eff}}$ and the supply voltage $V_{\text{DD}}(m)$ and operating frequency $f(m)$ of the core in power mode $m$. The second summand describes the static power consumption calculated as the product of leakage current $I_{\text{leak}}$ and supply voltage $V_{\text{DD}}(m)$.

$$P(m) = C_{\text{eff}} \cdot V_{\text{DD}}^2(m) \cdot f(m) + I_{\text{leak}} \cdot V_{\text{DD}}(m)$$  \hspace{1cm} (9)$$

For the construction of proper enforcement automata, we need to know the relation between the number $i$ of input features and the execution latency $L$ of each actor to be enforced in dependence of the number $n$ of cores and power mode $m$. Let $L(1, 1, m_{\text{max}})$ denote the latency for processing one feature on one core in power mode $m_{\text{max}}$ (highest voltage and frequency). In the following, $L(1, 1, m_{\text{max}})$ is computed by simulatively determining the execution latency of each actor per image for a representative set of 9,149 test images that fully covers the considered input space. Subsequently, the latency per feature of an actor is determined for each image by dividing its latency by the number of features $i$ in that image. Figure 17 illustrates the distribution (left) and the cumulative distribution (right) of the per-feature latency for the SD actor. Based on the obtained distribution, we then determine $L(1, 1, m_{\text{max}})$ according to the strictness of the latency requirement, which specifies the minimum rate $\nu \in [0, 1]$ of requirement satisfaction that must be achieved, specified by the user. In case of a) strict enforcement, a strictness of $\nu = 1$ is considered, and hence, the maximum observed per-feature latency among all images is used as $L(1, 1, m_{\text{max}})$. For b) loose enforcement, i.e. when $\nu < 1$, $L(1, 1, m_{\text{max}})$ is set to the lowest per-feature latency among all images such that for $\nu \cdot 100\%$ of images, the latency per feature is lower than or equal to the selected $L(1, 1, m_{\text{max}})$. In Figure 17 (right), this calculation corresponds to finding the lowest $x$-coordinate with a cumulative density of $\nu$. Having $L(1, 1, m_{\text{max}})$ determined, the following Eq. (10) can be used to determine the actor latency $L(i, n, m)$ based on the number of features $i$ to be processed within an image, the number of cores $n$ employed, and the power mode $m$ selected by an RRE scheme. In Eq. (10), $e(n)$ denotes the parallel efficiency in dependence of the number of cores $n$ employed for the computation with $e(n) = 1$ in the best case. In our experiments, we consider $e(n) = 1$.

$$L(i, n, m) = L(1, 1, m_{\text{max}}) \cdot \left\lceil \frac{i}{n \cdot e(n)} \right\rceil \cdot \frac{f(m_{\text{max}})}{f(m)}$$  \hspace{1cm} (10)$$
Note that Eq. (10) is a latency model specific to the SD and SM actors of our running application where $L(1, 1, m_{\text{max}})$ must be determined individually for each actor to be enforced. Moreover, Eq. (10) could be alternatively replaced with an elaborate manycore timing analysis, e.g. those from [2, 6, 10, 11, 18, 27], to derive tight worst-case latencies that support a variety of different resource arbitration policies and resource sharing schemes. Based on the power consumption and latency models in Eqs. (9),(10), the energy $E(i, n, m)$ required by the actor for processing an image with $i$ features using $n$ cores running in power mode $m$ is derived using Eq. (11).

$$E(i, n, m) = L(i, n, m) \cdot P(m) \cdot n$$  \hspace{1cm} (II)$$

Finally, the maximum number of features that can be processed within a given latency bound $UB_L$ using $n$ active cores running in power mode $m$ can be determined using Eq. (12), which is derived from Eq. (10), considering $L(i, n, m) \leq UB_L$.

$$i_{\text{max}}(UB_L, n, m) = \left\lfloor n \cdot e(n) \cdot \frac{UB_L}{L(1, 1, m_{\text{max}})} \cdot \frac{f(m)}{f(m_{\text{max}})} \right\rfloor$$  \hspace{1cm} (12)$$

For example, with Eq. (12), we may compute $i_{\text{max}}(80, 4, 20)$ for the SD actor which is the highest number $i$ of features of an input image for which a latency upper bound of $UB_L = 80$ ms can be enforced with a strictness of $\nu \in [0, 1]$. For instance, for loose enforcement with $\nu = 0.5$, we obtain $i_{\text{max}}(80, 4, 20) = 828$, and for strict enforcement where $\nu = 1$, the maximum enforceable workload decreases to $i_{\text{max}}(80, 4, 20) = 760$ features.
2.5 Energy-Minimised Timing Enforcement

RRE may involve to set, modify, or impose restrictions on typically OS-related techniques such as thread scheduling or memory management. In the following examples, we exemplify enforcement strategies for latency enforcement of individual actor executions or complete applications by varying the number \( n \in [1, 4] \) of cores (parallelism) and the power mode \( m \in [1, 20] \), calling a pair \((n, m)\) a configuration in the following. As, in general, multiple ways and settings for \( n \) and \( m \) might be feasible to enforce a requirement, the question becomes which requirement-adhering constellation the enforcer selects at run time. Often, this freedom of choice may be exploited by optimising one or more (secondary) objectives in addition to satisfying the given requirement. In the following, we consider energy demand as an objective to be minimised\(^1\).

Given a latency requirement \( UB_L \) and the RRE decision space of \( n \in [1, 4] \) and \( m \in [1, 20] \), a design space exploration [26] has been conducted per actor (or a set of actors) to derive, e.g. in our running example for the SD actor, the maximum number \( i_{\text{max}} \) of features that can be processed in each configuration \( c = (n, m) \) while respecting each latency requirement. Taking the SD actor with a latency upper bound \( UB_L = 80 \) ms as an example, Figure 18 illustrates the maximum workload \( i_{\text{max}} \) and the respective energy demand for each of the 80 possible \((n, m)\) configurations derived using Eq. (12) and Eq. (11), respectively in case of strict enforcement \((v = 1)\). The red line designates Pareto-optimal \((n, m)\) configurations.

Based on such a design space exploration and the resulting Pareto front of \((n, m)\) configurations, an energy-minimising enforcement automaton may be systematically constructed. Prior to each execution of the SD actor, this enforcement automaton selects a state (Pareto-optimal \((n, m)\) configuration) that is energy-minimal while satisfying the latency requirement in case input \( i \) is enforceable, thus if \( i \leq i_{\text{max}}(UB_L, n, m) \). For the example in Figure 18, the enforcement automaton has 31 states, each corresponding to one of the 31 Pareto-optimal \((n, m)\) configurations and the maximum enforceable workload \( i_{\text{max}} \) associated with that configuration. Here, the state selection is steered solely by the number \( i \) of features in the image to be processed by the SD actor\(^2\). For instance, for images with \( i \leq 9 \) features, \( n = 1 \) and \( m = 1 \)

\(^1\) Other objectives for choice of settings could be to activate the least number \( n \) of cores for increasing aspects of long-term reliability.

\(^2\) Note that in this example, the RRE could also be represented by a function table rather than an FSM, as the selection of state is only dependent on the input. More general cases such as restricting the allowed settings in each state to allow only step-wise increase or decrease of DVFS modes, can be constructed.
minimises the energy demand of the SD actor without violating the given latency requirement $UB_L = 80$ ms. For input images with $142 \leq i \leq 152$ features, an energy-minimal and requirement-adhering execution can be only realised if $n = 4$ cores are used for SD in parallel and power mode $m = 4$. Finally, a strict enforcement becomes impossible if $i > 760$, even using the configuration with the highest compute power, i.e. $n = 4$ and $m = 20$. For non-enforceable inputs, the enforcer needs to either throw an exception, stop processing (drop) the image, or process only as much as allowed by the latency bound. In the following, we present RRE techniques for distributed enforcement where each actor is individually enforced. For examples of centralised enforcement in which a more global view of the system state can be obtained by a centralised RRE instance that can take decisions affecting multiple actors and resources, we refer the interested reader to [25].

### 2.6 Distributed Enforcement

Figure 19 shows the resulting energy-minimising enforcement automata for a distributed enforcement strategy of the two individual actors SD and SM, with latency upper bounds 80 ms and 15 ms, respectively. The automata selecting the energy-minimising $(n, m)$ configurations obtained through the previously presented design space exploration are implemented as lightweight lookup tables for each actor. At run time, once an image is ready to be processed, the number $i$ of features in it becomes known. Prior to processing an image, the
2.6 Distributed Enforcement

RRE retrieves the energy-minimising configuration $c = (n, m)$ corresponding to $i$ features from the table and instructs the power manager to use these settings. As shown in Figure 19, the integration of enforcers may be achieved at the level of actors even as a model transformation by inserting the enforcer as an actor in front of each actor to be enforced, such that for each image to be processed, the energy-minimising configuration $(n, m)$ is set prior to execution of the image, and the configuration stays constant over the duration of processing this image. Employing the above enforcement strategy, the run-time manager is not compelled to run the enforced actors constantly with the maximum number $n = 4$ of cores and in the highest power mode $m = 20$ to guarantee the satisfaction of latency constraints in the presence of input variations, unless $i \geq 721$ for SD or $i \geq 985$ for SM. Also note that the given latency bounds cannot be strictly enforced for a feature count $i > 760$ for SD and $i > 1036$ for SM. Thus, the maximum workload that can be strictly enforced by both actors is limited to $i = 760$ features.

Figure 20 shows the histograms of observable latencies of the SD and SM actors a) without enforcement ($n = 4$ and $m = 20$) and b) with enforcement considering hard ($v = 1$) latency upper bounds of $UB_L = 80$ ms and 15 ms for SD and SM, respectively. The RREs choose a power mode that maximises energy savings while satisfying the given latency upper bound of each actor under enforcement as shown in Figure 18. For a variety of requirement strict-
ness levels, Table 4 finally presents the average dynamic energy consumption and the achieved dynamic energy savings of the SD and SM actors compared to the non-enforced scenario with \( n = 4 \) and \( m = 20 \). As can be seen, in case of loose enforcement, i.e. a strictness \( \nu < 1 \), the RRE achieves between 38.3\% and 41.2\% dynamic energy savings per enforced actor (respectively, between 39.3\% and 40.8\% collectively for the two actors) while satisfying latency upper bounds of 80 ms and 15 ms for the SD and SM actors, respectively. In case of strict enforcement \( (\nu = 1) \), the RRE still is able to achieve dynamic energy savings of 37.6\% for SD and 37.2\% for SM (respectively, 37.6\% collectively for the two actors) while guaranteeing that the given latency upper bound for each actor will never be violated. Evidently, this guarantee holds only for enforceable input images, i.e. those with \( i \leq 760 \) features for the SD actor and \( i \leq 1036 \) for SM (see the RRE tables in Figure 19). Finally, when analysing the overall energy consumption of all actors per frame, we obtain an overall

Table 4: Average dynamic energy consumption and savings per image for the SD and SM actors through distributed enforcement in dependence of requirement strictness defined as the minimum acceptable requirement satisfaction rate. The energy consumption without enforcement \( (n = 4, m = 20) \) serves as a baseline.

<table>
<thead>
<tr>
<th>requirement strictness</th>
<th>SD actor ( (UB_L = 80 \text{ ms}) )</th>
<th>SM actor ( (UB_L = 15 \text{ ms}) )</th>
<th>energy savings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( L(1, 1, m_{\text{max}}) ) [\mu J]</td>
<td>( L(1, 1, m_{\text{max}}) ) [\mu J]</td>
<td>SD+SM</td>
</tr>
<tr>
<td>norm. dist. index</td>
<td>avg. energy [mJ]</td>
<td>energy savings</td>
<td>avg. energy [mJ]</td>
</tr>
<tr>
<td>median</td>
<td>50%</td>
<td>385.4</td>
<td>41.2%</td>
</tr>
<tr>
<td>avg+1-σ</td>
<td>84.1%</td>
<td>392.0</td>
<td>40.4%</td>
</tr>
<tr>
<td>avg+2-σ</td>
<td>97.7%</td>
<td>402.3</td>
<td>39.9%</td>
</tr>
<tr>
<td>maximum</td>
<td>100%</td>
<td>420.7</td>
<td>37.6%</td>
</tr>
<tr>
<td>without enforcement</td>
<td>-</td>
<td>223.2</td>
<td>(baseline)</td>
</tr>
</tbody>
</table>
dynamic energy reduction of 33.8% in case of strict enforcement \((v = 1)\) and between 35.4% and 36.8% in case of loose enforcement \((v < 1)\) for the whole application, even though only two out of 9 actors were needed to be enforced. Noteworthy, the execution time and energy consumption of the RRE itself can be neglected when being implemented by simple table lookups.

### 2.7 Feedback-based Enforcement Strategies

The goal of enforcement strategies is to have the executions of a program on a given MPSoC platform satisfy a given set of non-functional requirements, even under an environmental input variation. For each discrete program execution \(k\), the input is modelled in the following by an environment feature vector \(i(k) \in I\), where \(I\) is called the environment space. As previously introduced, the number \(n\) of cores assigned to a program can be dynamically adapted as well as the voltage/frequency setting \(m\) per core for enforcement. Again, we call such a setting \((n, m)\) to execute a given program a configuration \(c\) and the set of possible system configurations to execute a program configuration space \(C\). In order to be able to consider and also react adequately to previously unseen and unexpected inputs, we consider in the following a more general class of RRE techniques than statically characterised pro-active RRE techniques as introduced in Sections 1.5 and 1.6. These are called Feedback-based RRE and react to a violation or satisfaction of a non-functional requirement \(\phi\) based on feedback from the system-under-control by adapting the configuration \(c(k + 1)\) for the \((k + 1)\)th execution accordingly. Figure 21 illustrates the concept of feedback-based RRE that will be explained in the following.

![Feedback-based RRE diagram](image)

Figure 21: Schematic illustration of feedback-based RRE. A system response vector \(r\) is mapped to a binary requirement response vector \(\phi\) on the basis of which the enforcement FSM \(F\) determines the next configuration \(c(k + 1) \in C\). Reprinted from [8].
2.7.1 Formal Definitions

As we are interested in non-functional execution qualities of a given program during its $k$th execution in configuration $c(k) \in C$ when experiencing an environmental input vector $i(k) \in I$ and whether a given set of requirements on these are met or not, the MPSoC platform, or system-under-control, is abstracted by a single function called system response function $r : I \times C \rightarrow \mathbb{R}^h$ [8]. The system response at execution $k$ is a vector of $h$ relevant execution qualities $r(i(k), c(k)) = (o_1(k), ..., o_h(k))$, e.g. corresponding to observed latency, power consumption, etc., during the $k$th execution.

Now, requirements on these execution qualities such as deadlines (e.g. in the form of latency constraints) must be guaranteed for each execution, where each quality $o_j, j = 1, ..., h$ is bounded according to Definition 1 using intervals from which two propositions $\varphi_j^{LB}$ and $\varphi_j^{UB}$ can be deduced

$$\varphi_j(o_j(k)) = \varphi_j^{LB}(o_j(k)) \land \varphi_j^{UB}(o_j(k))$$

$$= (LB_{o_j} \leq o_j(k)) \land (o_j(k) \leq UB_{o_j}) \quad (13)$$

where $LB_{o_j}$ and $UB_{o_j}$ denote the lower, respectively upper bound on the execution quality $o_j$.

The system response $r(i(k), c(k)) = (o_1(k), ..., o_h(k))$ is finally mapped to a requirement response using a requirement response function [8]:

$$\beta := \phi (o_1(k), ..., o_h(k)) = (\varphi^{LB}(o_1(k)), \varphi^{UB}(o_1(k)), ..., \varphi^{LB}(o_h(k)), \varphi^{UB}(o_h(k))) \in \{0, 1\}^{2h} \quad (14)$$

This binary vector $\beta$ serves as the input to the enforcement FSM $F$. Based on the requirement response, $F$ computes the next configuration $c(k + 1) \in C$ in reaction to enforce the desired non-functional properties for the next execution. Mathematically, an enforcement FSM is defined as follows.

**Definition 4.** ([8]) An enforcement FSM ($F$) is a deterministic finite state machine (Moore machine) that can be described by a 6-tuple $(Z, z_0, B, \delta, C, \gamma)$:

- $Z$ is a finite set of states.
- $z_0 \in Z$ is the initial state.
- $B$ is the input alphabet.
2.7 Feedback-based Enforcement Strategies

- $\delta$ is the transition relation: $\delta \subseteq B \times Z \times Z$ with $(\beta, z, z')$ representing a transition from state $z$ to state $z'$ under input $\beta$.

- $C$ is the output alphabet, also called configuration space.

- $\gamma$ is the output function, which maps each state to the output alphabet: $\gamma : Z \to C$.

Finally, for being able to verify conditional and temporal verification goals on a proper enforcement of requirements, we require a formal model of the environment that influences the system-under-control. In order not to be able to just analyse a RRE for individual traces, but rather to argue about the qualities of enforcers rather for families of traces, the environment is modelled by means of a discrete-time Markov Chain called environment FSM in the following.

Naively, we could associate a distinct environment state with each distinct input $i \in I$. However, this is not practical, because the number of possible input values can be very large, and hence the number of the environment states would be large as well, possibly leading to a state explosion. A possible solution is to partition the environment space $I$ into equivalence classes or partitions $p$ leading to the same requirement response $\phi$ [8].

**Definition 5.** ([8]) A partitioning $P$ of an environment space $I$ is a set of $|P|$ subsets $p_0, ..., p_{|P|-1}$ where

- $p_l \neq \emptyset$
- $p_{l_1} \cap p_{l_2} = \emptyset, \forall l_1 \neq l_2$
- $\bigcup_{l=0}^{|P|-1} p_l = I$

Obviously, any subset of instances $i \in I$ can be grouped into a single partition block $p$, if the system produces the same requirement response $\phi$ for all $i \in p$ in each configuration $c \in C$ [8]:

$$\exists \psi_c \in \{0, 1\}^{2^h} : \forall i \in p : \phi(r(i, c)) = \psi_c \quad \forall c \in C$$  \quad (15)

In summary, we can partition a large environment space of inputs $I$ into partitions $p$ that deliver the same binary requirement satisfaction vector for all $i \in p$ in each configuration $c \in C$. These partitions $p$ define a discrete state space of an environment FSM $E$ that can now be defined as follows.
Definition 6. ([8]) An environment finite state machine $E$ is a discrete-time Markov chain (DTMC) defined by a 3-tuple $(S, a, \Delta)$:

- $S$ is the finite set of states. Each $s \in S$ is assigned exactly one partition $p \in P$ of the environment space $I$, according to Eq. (15).
- $a : S \rightarrow [0, 1]$ is a function that assigns each state $s \in S$ its initial state probability $a(s)$.
- $\Delta \in [0, 1]^{[S] \times [S]}$ is a transition probability matrix.

In the following, it is shown how to construct an environment DTMC $E = (S, a, \Delta)$. In general, any distribution can be assumed to derive initial probabilities $a(s)$. In case the probabilities $\mathcal{P}(i), i \in I$ are known, we may obtain $a(s) = \mathcal{P}(s) = \sum_{i \in p} \mathcal{P}(i)$. Moreover, for a given trace $R = i(0), i(1), \ldots, i(|R| - 1)$, $a(s)$ can be simply determined from the number of occurrences of inputs $i \in p$ associated with state $s$ in trace $R$ divided by $|R|$ and the transition probability matrix $\Delta$ by assigning each pair $(s, s')$ of states the number of occurrences of transitions from $s$ to $s'$ divided by the number of all transitions leaving $s$, see [8] for details.

With environment states representing partitions of the environment space leading to the same requirement response, the system response function can be refined using environment states: $r' : S \times C \rightarrow \mathbb{R}^h$, see Figure 22. The figure shows the final and complete formal model for verifying the proper enforcement of requirements.

![Figure 22: Refined model of feedback-based RRE with environment modelled by an environment FSM $E$. Reprinted from [8].](image-url)
2.7.2 Verification Goals

Depending on the type of requirement, we distinguish different levels of strictness of requirement enforcement, see Section 2.2 and [26]. Accordingly, we define several verification goals of interest using temporal logic as follows:

- **For strict enforcement** of an execution quality $o_j$, $\varphi_j$ should always hold. This is indicated in CTL [5] by the abbreviation $AG(\varphi_j)$. For $h > 1$, such proposition leads to $AG\left(\bigwedge_{j=1}^{h} \varphi_j\right)$.

- **For loose enforcement**, the requirement $\varphi_j$ should eventually hold. This can be indicated in CTL [5] by the abbreviation $AF(\varphi_j)$. For loose enforcement, it is also interesting to check if a violating run is always followed by a next satisfying run. This could be expressed as: $AG\left(\neg\varphi_j \rightarrow X \varphi_j\right)$. For $h > 1$ requirements, this leads to $AG\left(\bigvee_{j=1}^{h} \neg\varphi_j \rightarrow X\bigwedge_{j=1}^{h} \varphi_j\right)$.

Finally, as we do not want to reason about or verify the satisfaction of verification goals during single runs or for single traces of input stimuli, but depending on a DTMC model of the typically uncertain environment (i.e. $E$), we can use PCTL [4], a probabilistic variant of CTL, to specify stochastic verification goals. In the following, let $P_{\varphi_j}$ denote the probability of satisfying a proposition $\varphi_j$. In our experiments, we use PRISM [17], a probabilistic model checker, to perform verification of a number of verification goals of interest on enforcers, see Section 2.7.4.

**Examples.** In the following, the previously introduced image processing application as shown in Figure 15 is revisited. Let the goal be to enforce a single-sided (upper bound) latency requirement per processed frame $k$. First, the construction of an environmental FSM $E = (S, a, \Delta)$ from an input trace $R = i(0), i(1), \ldots, i(|R| - 1)$ is explained. Here, $i(k) \in \mathbb{N}$ is just a scalar, denoting the image-content-dependent number of features inside an image frame $k$, thus a measure of workload to be processed in frame $k$. Subsequently, we introduce and analyse different RRE enforcement FSM variants. For simplicity of the example, let only a single single time-critical SD actor with a local latency upper bound $\bar{L} = UB_{o_L} = 80$ ms to be enforced. Such a one-sided requirement can be described using the proposition $\varphi_L(o_L) := \varphi_L^{UB}(o_L) = (o_L \leq UB_{o_L})$.

According to Section 2.7.1, we first partition the environment space $I$ into a set $P$ of partitions, by computing the system response $o_L = r(i, c)$ for all different inputs $i \in I$ for each $c \in C$, and then deriving the partitions $p$ of inputs $i \in I$.
that result in the same requirement response \( \phi(r(i, c)) \) for each \( c \in C \). Like in Section 2.5, we achieve this by computing the maximum number of features \( i_{\text{max}}(\bar{L}, c) \) that can be processed within a given deadline \( \bar{L} \) in configuration \( c \in C \). In case of the SD actor enforced in our running application, extracting the energy-minimal configurations delivers a total of \(|C| = 32\) partitions as shown in Table 5.

Table 5: Each partition \( p_i \) of inputs \( i \in I \) corresponds to an environment state \( s_i \) that causes the same requirement response \( \phi(r(i, c)) \forall i \in p \) in each \( c \in C \).

<table>
<thead>
<tr>
<th>( p_j )</th>
<th>( c_j )</th>
<th>( n_j )</th>
<th>( m_j )</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p_0 )</td>
<td>( c_0 )</td>
<td>1</td>
<td>1</td>
<td>0-33</td>
</tr>
<tr>
<td>( p_1 )</td>
<td>( c_1 )</td>
<td>1</td>
<td>2</td>
<td>34-67</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( p_9 )</td>
<td>( c_9 )</td>
<td>2</td>
<td>5</td>
<td>306-338</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( p_{31} )</td>
<td>( c_{31} )</td>
<td>4</td>
<td>20</td>
<td>2576-2708</td>
</tr>
</tbody>
</table>

Based on the above partitioning, an environment state \( s_j \) is defined for each partition \( p_j = \{ i_{\text{max}}(\bar{L}, c_{j-1}) + 1, \ldots, i_{\text{max}}(\bar{L}, c_j) \} \), where \( 0 \leq j < |C| \), thus \( s_0 \) is assigned the partition \( p_0 = \{ 0, \ldots, i_{\text{max}}(\bar{L}, c_0) \} \). For example, \( s_9 := p_9 = \{ i_{\text{max}}(\bar{L}, c_8) + 1, \ldots, i_{\text{max}}(\bar{L}, c_9) \} = \{306, \ldots, 338\} \).

In the following, we construct and analyse three variants of enforcement FSMs, each having as many states as configurations \(|C|\), i.e. \( Z = \{0, \ldots, 31\} \), a binary input \( \beta \in \{0, 1\} \) with \( \beta = \phi(r'(s, c)) = \phi_L(o_L) = (o_L \leq UB_{o_L}) \), an assumed initial state \( z(0) = z_0 = 17 \) and the configuration space \( C \) of cardinality \(|C| = 32\) according to Table 5.

- 1-step Enforcement FSM \( F_1 \): [7] proposes a technique called 1-step enforcement that decreases, resp. increases the current state (configuration) by exactly one step in case of a satisfaction (\( \phi_L \)), resp. violation (\( \phi_L \)) of a verification goal to be enforced. A corresponding enforcement FSM \( F_1 = (Z, z_0, B, \gamma, C, \delta_1) \) is shown in Figure 23. It has the following transition relation \( \delta_1 \):

\[
z(k + 1) = \begin{cases} 
0 \quad & \phi_L \land (z(k) = 0) \\
\beta L \land (z(k) \neq 0) \quad & \phi_L \land (z(k) \neq 0) \\
31 \quad & \phi_L \land (z(k) = 31)
\end{cases}
\]
2.7 Feedback-based Enforcement Strategies

- 8-step Enforcement FSM $F_2$: For comparison, we define an alternative 8-step Enforcement FSM $F_2 = (Z, z_0, B, \gamma, C, \delta_2)$, shown in Figure 24, with the following transition relation $\delta_2$:

$$ z(k + 1) = \begin{cases} 
0 & \varphi_L \land (z(k) = 0) \\
z(k) - 1 & \varphi_L \land (z(k) \neq 0) \\
z(k) + 8 & \varphi_L \land (z(k) \notin \{24, \ldots, 31\}) \\
31 & \overline{\varphi_L} \land (z(k) \in \{24, \ldots, 31\}) 
\end{cases} $$ (17)

- Clairvoyant Enforcement FSM $F_3$: represents the theoretically perfect case when the enforcement FSM is able to exactly forecast the next environment state $s(k+1)$, and determines based on it the next configuration $c(k+1)$ accordingly as introduced in Section 2.5 and Section 2.6 and selecting a configuration based on the currently observed input $i(k)$ according to Table 5.

2.7.3 Property Specification in PRISM

In the following, we distinguish and exploit two different types of verification goals:

- Non-probabilistic verification goals: written in CTL (computation tree logic) and LTL (linear temporal logic). The outcome of such verification goals is either true or false.
• Probabilistic verification goals: a probability operator $\mathcal{P}$ can be used to ask for the probability of a property to hold. It is formally specified in PCTL [14]. The outcome is a value between 0 and 1.

Both types of verification goals can be verified in PRISM [17], by default in the initial state of the model. However, we are interested in computing average probabilities of verification goals over all states. This is possible by using so-called filters in PRISM. Finally, LTL formulas have a bounded variant in PCTL [14], which allows to add an upper bound $\lambda$ on the number of successive steps or iterations in our model. In the following, we formulate several verification goals of interest for enforcement automata. The first is: $P=?[F^{\leq\lambda}L]$ which asks for the probability of $L$ to hold within $\lambda$ steps. The second being: $P=?[G^{\leq\lambda}(\neg L)]$ which denotes the probability of $L$ to never hold in any of $\lambda$ consecutive executions. Finally, with $P=?[\neg L \rightarrow F^{\leq\lambda}(L)]$, we ask for the probability of returning to a satisfying execution from a violating one in no more than $\lambda$ steps. Next, we provide verification results for the formulated verification goals.

2.7.4 Verification Results

We specified the previously introduced enforcement and environment FSMs for verification by the PRISM model checker using the PRISM modelling language [16]. In the following, we present verification results for the previously introduced single and one-sided latency requirement of the SD actor of the image processing application for two bounds of $\bar{L}=80$ ms and $\bar{L}=200$ ms.

Non-probabilistic verification goals: In case of strict enforcement, $AG(L)$ is false for enforcement FSMs $F_1, F_2$, and for both deadlines. Even that $F_2$ can potentially enforce the relaxed deadline $\bar{L}=200$ ms even for the highest possible input, the requirement $L$ doesn’t always hold for neither of them. The reason is that in a given state, there is always a possibility to receive a non-enforceable input next with non-zero probability that cannot be enforced by a single state transition. For $F_3$, $AG(L)$ holds for the deadline $\bar{L}=200$ ms, because $F_3$ predicts exactly the next environment state $s(k+1)$, and all possible inputs are enforceable for this deadline. However, it doesn’t hold for $\bar{L}=80$ ms, because not all inputs are enforceable here.

In case of loose enforcement, however, $AF(L)$ is true for all enforcement FSMs and both deadlines, which means that $L$ is eventually satisfied for all paths. For enforcement FSMs $F_1, F_2$, and both deadlines, $AG(\neg L \rightarrow X L)$ is false, which means that the system is not always able to satisfy the deadline for the next execution $(k+1)$ once missing it in a state during the $k$th execution.
For $F_3$, it holds for $\bar{L} = 200$ ms, because $F_3$ does not violate this deadline for all possible inputs. However, it doesn’t hold for $\bar{L} = 80$ ms, because not all inputs are enforceable here. For $\bar{L} = 200$, $AG[\neg \phi_L \rightarrow F^{\leq 4}(\phi_L)]$ is true for $F_2$, because we can enforce the highest observable input $i = 5,513$ in $R$ under the deadline $\bar{L} = 200$ ms using the enforcement state $z = 29$ or higher, which we can reach starting from any state through always four transitions at most (because $F_2$ transitions forward 8 steps when $\phi_L = 0$). Yet, for $F_1$, this verification goal does not hold, because $F_1$ would need to take 29 transitions to reach $z = 29$ in the worst case. For $\bar{L} = 80$, this verification goal doesn’t hold, neither for $F_1$ nor for $F_2$, because we cannot enforce all the possible inputs. For $F_3$, this verification goal holds for the deadline $\bar{L} = 200$ ms, because $F_3$ does not violate this deadline for all the possible inputs. However, it doesn’t hold for the case of $\bar{L} = 80$ ms, because not all inputs are enforceable for this deadline, and thus we might receive 5 or more consecutive violations (i.e. non-enforceable inputs).

**Probabilistic verification goals:** We know from before that $\phi_L$ doesn’t always hold. Still, we can check the probability of $\phi_L$ to never hold in any of $\lambda$ consecutive executions by computing $P_\lambda[\neg \phi_L] \rightarrow F^{\geq \lambda}(\phi_L)$. 

![Figure 25](image.png)

Figure 25: Probability of returning to a deadline-satisfying state in no more than $\lambda$ executions: $P_\lambda[\neg \phi_L \rightarrow F^{\geq \lambda}(\phi_L)]$ for a single latency bound $\phi_L = (o_L \leq \bar{L})$ where $\bar{L} \in \{80, 200\}$. The evaluation starts at $\lambda = 1$ as this verification goal involves an initial step for $\neg \phi_L$ at $\lambda = 0$, reproduced from [8].

Figure 25 shows $P_\lambda[\neg \phi_L \rightarrow F^{\geq \lambda}(\phi_L)]$, which is the probability of returning to a deadline-satisfying execution from a violating one in no more than $\lambda$
steps. Notice that this probability for enforcer \( F_3 \) in the case of \( \bar{L} = 200 \) ms is always 1, as \( F_3 \) never violates this deadline for the possible inputs. However, even if it predicts perfectly the next environment state \( s(k+1) \), it still has the possibility of receiving a non-enforceable input in the case of \( \bar{L} = 80 \) ms, and that is why it is not always 1. Still, it has a higher probability than \( F_1 \) and \( F_2 \) of returning to a deadline-satisfying execution from a violating one in no more than \( \lambda \) steps, as shown in Figure 25. Observe that enforcer \( F_2 \) always needs less or equal steps to reach a probability of \( \approx 1 \) than enforcer \( F_1 \). The result is plausible, since the 8-step Enforcement FSM \( F_2 \) jumps 8 configuration levels if \( \varphi_L = 0 \) (both FSMs going to the next lower state (configuration) if \( \varphi_L = 1 \)). We can also see that for all enforcement FSMs, the probability of returning back to a deadline-satisfying state in no more than \( \lambda \) steps is always higher or equally high for the relaxed deadline of \( \bar{L} = 200 \) ms than for the case of \( \bar{L} = 80 \) ms for each enforcement FSM. In [8], we also presented results on analysing other verification goals such as \( P = \gamma [ F^\infty \lambda (\varphi_L) ] \) and \( P = \gamma [ G^\infty \lambda (\neg \varphi_L) ] \).

The described verification works similar for a) two-sided requirements and b) \( h > 1 \) requirements. As an example, consider the 1-step Enforcement FSM \( F_4 = (Z, z_0, B, \gamma, C, \delta_4) \) shown in Figure 26. It has as many states as configurations \(|C|\), i.e. \( Z = \{0, \cdots, 31\} \) and \( \beta \in \{0, 1\}^2 \) with \( \beta = \varphi(r'(s, c)) = (L^B \leq o_L, o_L \leq U^B) \) as input, \( z(0) = z_0 = 17 \) as initial state, and the following transition relation \( \delta_4 \):

\[
\begin{align*}
z(k+1) &= \begin{cases} 
0 & (\varphi_L^{LB}, \varphi_L^{UB}) \land (z(k) = 0) \\
z(k) - 1 & (\varphi_L^{LB}, \varphi_L^{UB}) \land (z(k) \neq 0) \\
z(k) & (\varphi_L^{LB}, \varphi_L^{UB}) \land (z(k) \neq 0) \\
z(k) + 1 & (\varphi_L^{LB}, \varphi_L^{UB}) \land (z(k) \neq 31) \\
31 & (\varphi_L^{LB}, \varphi_L^{UB}) \land (z(k) = 31)
\end{cases}
\end{align*}
\] (18)

Figure 26: 1-step enforcement FSM for enforcing a two-sided requirement \( \varphi_L \). Annotated to each state is the output configuration \( c = (n, m) \), consisting of number \( n \) of powered cores and power mode \( m \).

In the following, we verify the previously introduced non-probabilistic and probabilistic verification goals based on the two-sided requirement. For the
non-probabilistic verification goals, $AG(\phi_L)$ is false for enforcement FSM $F_4$. The reason is that there is always a possibility to receive in a given state a non-enforceable input next with non-zero probability that cannot be enforced by a single state transition (i.e. $\phi^{UB}_L$), or receive in a given state an input next that violates the lower bound proposition (i.e. $\phi^{LB}_L$). For $F_4$, $AF(\phi_L)$ is true, $AG(\neg \phi_L \rightarrow X \phi_L)$ is false, and $AG[\neg \phi_L \rightarrow F^{\leq 5}(\phi_L)]$ is false, even though we can enforce the highest observable input $i = 5,513$ in $R$ under the deadline $\hat{L} = 200$ ms, but the lower bound proposition might be violated (i.e. $\phi^{LB}_L$).

Figure 27 plots the verification results of the probabilistic verification goals introduced afore for a two-sided requirement.

![Figure 27: Probability of different verification goals for different numbers of steps $\lambda$ for a two-sided requirement that has $UB_{oL} = 200$ ms and $LB_{oL} = 80$ ms for the enforcement FSM $F_4$, see [8].](image)

For the case of $h > 1$ requirements, we refer the reader to [9], where multi-requirements enforcement strategies for enforcing multiple non-functional requirements at a time, like latency and energy consumption, are proposed and verified. We finally present results on verification times.

All the above experiments were performed on an Intel i7-1065G7 CPU at 1.30–1.50 GHz under 64-bit Windows 10 and using PRISM version 4.7. The verification times for all the experiments conducted using enforcement FSMs for each verification goal and requirement are shown in Table 6. We can see that the verification times of enforcement FSMs $F_1$ and $F_2$ are roughly the same as they have a similar number of states. The experiments for $F_3$ showed
the lowest verification time (0.001 s), because of its low number of states and transitions. Moreover, increasing $\lambda$ in the probabilistic verification goals showed a light increase in verification time for all enforcers. Finally, notice that the verification time for the requirement $o_L \leq 80$ ms is higher than for $o_L \leq 200$ ms, because the number of states and transitions for $o_L \leq 80$ ms is higher than the case of $o_L \leq 200$ ms. Likewise, the verification times for the case of two-sided requirements are higher than the case of a single requirement bound.

### 2.8 Conclusions and Acknowledgement

In this chapter, we presented a formalisation, classification, and the practice of a class of techniques subsumed under the term of run-time requirement enforcement (RRE) that make the system management software of an MPSoC platform become the advocate of a parallel application program instead of both acting independently with the goal to provide means for the satisfaction of given non-functional requirements of parallel program execution such as performance (latency, throughput), power or energy consumption, or reliabil-
ity. The non-functional requirements can thereby be expressed by interval ranges and specified over the application program as a whole or for individual actors/tasks or threads, or even segments thereof. The goal of RRE is to enforce the satisfaction of these requirements at run time. For a distributed object detection application as an example, it has been shown that the variability of non-functional execution properties can be greatly reduced in dependence of the level of strictness that shall be fulfilled for each requirement, see also Chapter 15 for a developed live demonstrator of throughput enforcement for this application. Moreover, it has been shown that RRE techniques can be either implemented in a centralised or distributed manner.

We also investigated the problem of verification of techniques for run-time requirement enforcement of execution properties of applications on MPSoCs. We proposed an approach for reducing the environment (input) space into equivalent-response input partitions, which may lead to reductions of verification times by several orders of magnitude. Based on the stochastic environment DTMC, an enforcement strategy modelled by an enforcement FSM, a system response model and finally a given set of (one-sided or two-sided) interval-based requirements, probabilistic model checking can be performed to formally assess and compare different enforcement strategies. We also proposed a set of probabilistic as well as non-probabilistic verification goals of interest to assess enforcement strategies.

With reported verification times in the range of just a few ms for real-world examples, we conclude that our approach is feasible to analyse and compare the quality of enforcement strategies. Such techniques usually require the evaluation of several thousand candidates to determine Pareto-optimal solutions. For the presented case study, the proposed reduction of the state space of the environment DTMC would allow the evaluation of 100,000 candidate enforcer FSMs within just 10 minutes rather than 17 days in the non-partitioned case. In the future, we will investigate techniques to systematically explore, optimise, and synthesise enforcement automata automatically.

Finally, we would like to acknowledge the following list of persons for their very valuable contributions to the basics of invasive computing: Frank Hannig, Sascha Roloff, Gregor Snelting, Pouya Mahmoody, Maximilian Wagner, Andreas Weichslgartner, and Andreas Zwinkau.

References

2 Basics of Invasive Computing


Characterisation and Analysis of Invasive Algorithmic Patterns

Michael Bader, Stefan Wildermann, Michael Glaß, Alexander Pöppl, Behnaz Pourmohseni, Tobias Schwarzer, Jan Spieck, Mario Wille

Abstract The mapping of applications onto heterogeneous MPSoC architectures can have a strong influence on execution properties like execution time, energy consumption, or the number of deadline violations. In many application domains such as multimedia, HPC, signal and image processing, the execution properties are highly input dependent and have to fulfil real-time or performance requirements. A static mapping of an application to an MPSoC and a fixed scheme for executing the application tasks, therefore, may often result in an over-provisioning of resources for guaranteeing a given deadline or performance bound over all possible input variations. However, this may lead to sub-optimal utilisation of system resources, power and energy consumption.

This chapter first describes an actor-based programming model and demonstrates how to make use of it to provide an application execution scheme that can be adapted to input-dependent variations. Furthermore, it presents hybrid application mapping that performs dynamic run-time adaptations of resource claims upon observation of fundamental changes of environment and input data. We give insights on applying this scheme for energy-aware run-time management of soft real-time applications.

3.1 Introduction

Many applications and modern MPSoC systems exhibit varying execution phases and dynamically changing workloads. Recent observations give evidence of a rapid increase in the number of concurrent applications in embedded systems and HPC systems with different requirements. The system's workload, that is, the mix of concurrently executed applications tends to change over time. Such workload variations, including the activation and the termination of applications, often happen in reaction to external events whose arrival cannot be predicted. Moreover, applications can dynamically change between different execution phases or may have varying workload defined by their input data.
As the system utilisation as well as workload scenarios and execution phases of an application may vary strongly, statically mapping and executing them so that they perform best for the worst case or average case becomes inadequate with respect to resource utilisation. Particularly when their execution has to fulfil a set of non-functional requirements, static resource reservation and run-time requirement enforcement (RRE) techniques as introduced in Chapter 2 would either not be sufficient to cope with large variations, or require an undesirable over-provisioning of resources. In such situations, applications, algorithms and in particular also their mappings (resource claim) must be adapted to react to dynamic changes while still guaranteeing real-time and other non-functional requirements.

In this chapter, we summarise techniques for dynamically adapting the behaviour of applications as well as their resource claims. Whereas the first part focuses on patterns for algorithmic adaptations of applications (Section 3.2), the second part presents several techniques for application characterisation and dynamic mapping adaptations (Section 3.3).

### 3.2 Algorithmic Patterns

To evaluate the benefit of invasion-aware algorithmic patterns, we developed SWE-X10 [13], a proxy application for space- and time-adaptive tsunami simulations. SWE-X10 itself is built on the X10 extension ActorX10 [27] and uses an efficient finite volume solver for the shallow water equations. It follows an actor approach towards a more dynamic use of resources. We show how finite state machines, as part of an actor-oriented parallel programming model, can be used to identify parts of the computational domain that are not updated: following a lazy activation paradigm, respective compute resources could remain idle until needed. In addition, the actor approach allows Cartesian grid patches to propagate in time without global control. In the following, we first introduce the numerical model and the shallow water equations and secondly discuss implementation details of the SWE-X10 proxy application [13, 14] and its performance.

#### 3.2.1 The Shallow Water Equations

The simulation of tsunamis is not only a problem of high societal relevance, but also offers a multitude of computational challenges (see also Chapter 14). Our proxy applications adopt a numerical method by LeVeque et al. [11], who use Finite Volume discretisation to solve the shallow water equations, a system
of non-linear hyperbolic partial differential equations, which in their simplest form may be written as:

\[
\begin{bmatrix}
  h \\
  hu \\
  hv
\end{bmatrix}
+ \begin{bmatrix}
  hu^2 + \frac{1}{2} gh^2 \\
  hu \\
  hv^2 + \frac{1}{2} gh^2
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
= \begin{bmatrix}
  0 \\
  g h B_x \\
  g h B_y
\end{bmatrix},
\]

(19)

Here, \( h(x, y, t) \) is the water depth, and \( u(x, y, t) \) and \( v(x, y, t) \) are the respective water velocities at position \((x, y)\) and time \( t \). \( g \) is the gravitational constant.

Source terms on the right-hand side model the influence of the terrain below sea level, i.e. the elevation of the ocean floor—typically referred to as bathymetry data, here \( B(x, y) \).

As suggested by LeVeque et al. [11], SWE-X10 discretises the simulation domain on an equidistant Cartesian grid. Each cell \( i, j \) stores the cell-averaged quantities \( h_{i,j}^{(n)} \), \( (hu)_{i,j}^{(n)} \), \( (hv)_{i,j}^{(n)} \), \( b_{i,j}^{(n)} \) at time step \( t_n \). For each time step numerical fluxes are computed between any two adjacent cells. From the balance of these fluxes—essentially the in- and outflow of mass and momentum across cell edges—the updated quantities \( h_{i,j}^{(n+1)} \), \( (hu)_{i,j}^{(n+1)} \), \( (hv)_{i,j}^{(n+1)} \), \( b_{i,j}^{(n+1)} \) for the next time step are computed.

The resulting update scheme follows a wide-spread pattern (“stencil computation”) in scientific computing, but also poses interesting challenges for resource-aware computing:

**Local time stepping:** The time step size \( \Delta t_n \) has to be chosen such that a wave cannot travel farther than one grid cell per time step. As the wave speed depends on the water depth \( (c_{\text{wave}} = \sqrt{gh}) \), choosing different time step size in different parts of the domain can substantially reduce computation time, but also creates a complicated time-stepping scheme and non-balanced computational load.

**Wetting and drying:** When a tsunami inundates the coast, the wet \((h > 0)\) and dry \((h = 0)\) areas dynamically grow or shrink. Here, the more complicated numerical schemes that correctly treat wetting&drying are more expensive, whereas dry regions are not simulated at all.

**Ocean at rest:** In regions where the tsunami has not “entered”, yet, the numerical scheme will essentially recompute the steady state in every time step. This computational effort could be saved by lazy evaluation, i.e. by only updating grid cells that are currently changing in time.
Taken together, we are facing complicated algorithmic patterns that require complex algorithmic schemes and lead to dynamic-load-balancing challenges. The actor model allows us to formalise these schemes and at the same time expose entities of the code to design space exploration, etc.

3.2.2 SWE-X10

In SWE-X10, the ActorX10 extension is used to decouple the application logic from the parallelisation scheme. This facilitates more high-level optimisations, such as lazy activation, where actors are only enabled and start computing when they are actually reached by the tsunami wave. Furthermore, the event queue semantics of ActorX10 enables an overlap of communication and computation without intervention from the application developer, as long as there is a sufficient amount of actors per node. Each actor is assigned to a Cartesian patch of the computational grid. Using the actor’s finite state machine to control patch updates, we realise lazy activation of patches, only when a propagating wave enters the respective patch. Overlapping of communication and computation in the fully non-central actor-based control, as well as careful optimisation (especially vectorisation) of kernels lead to high performance and parallel efficiency in shared and distributed memory.

Actor Model  The actor model is a computational model first described by Hewitt et al. [7] in the context of artificial intelligence and later formalised by Agha [1]. Essentially, an actor may be described as a thread of execution with a state attached to it. Actors execute their code independently and without explicit synchronisation. Collaboration between actors may be attained by explicit messages through statically defined communication channels, or at the start of the program execution. This structure allows for a very low amount of coupling. As communication is handled by the actor framework, actors only need to specify their communication interfaces, and how to react to data being available to them. The concrete implementation of the actors is encapsulated from the other actors in the actor graph. SWE-X10 uses the actor model where control is realised by means of finite state machines to implement a distributed and locally coordinated time stepping scheme.

The basic building blocks of the actor-based computational model are channels, ports, and actors. These components are arranged in the actor graph. An actor graph is a directed graph \( G_a = (A, C) \) containing a set of actors \( A \) and a set of channels \( C \subseteq A.O \times A.I \) connecting actor output ports \( A.O \) with actor input ports \( A.I \). Each channel has a buffer size determined by \( n : C \to \mathbb{N}^+ \),
and a possibly empty sequence of initial tokens denoted by \( d : C \rightarrow D^* \) where \( D^* \) denotes the set of all possible finite sequences of tokens.

Actors may only communicate externally using their ports. A port may only process tokens of a specific type. The model distinguishes two types of ports, input ports as endpoints for receiving data, and output ports for sending data to other actors. Ports are connected through \textit{channels}. Channels are buffers of finite size that may contain up to \( n(c), c \in C \) tokens of type \( D \). Communication between actors connected in this way occurs in \textit{first in, first out (FIFO)} order, i.e. the ordering between messages in the same channel is always preserved. This is the natural mode of operation for many applications, and facilitates the use of widely used patterns in scientific computing, such as a halo layer exchange, as the application may rely on the fact that the data is always read in sequence. The sole means of communication between actors is through channels. Implicit communication of any kind, for example through shared data structures or shared memory locations, is prohibited. An actor is therefore constrained to only consume data items (\textit{tokens}) from channels connected to its input ports and to produce tokens on channels connected to its output ports. It is also possible to place initial tokens in \textit{FIFO} channels, e.g. \( d(c) = \langle 1, 2 \rangle \) to denote that two initial tokens with the values 1 and 2 are on channel \( c \).

\section*{Actor-Based Coordination}

SWE-X10 relies on a decomposition of the simulation domain into rectangular patches. In order to update cells to a new state \( d_{i,j}^{(n+1)} \), it is required to know the values of the neighbouring cells quantities at the current time step \( n \). This, in turn, necessitates the transfer of the values of the outermost layer of the adjacent patches. These are typically stored in an extra row or column outside the actual patch, in the \textit{ghost layer}, and they have to be available before a new time step can be computed. In SWE-X10, the task of transferring the ghost layers, and the coordination of the computation falls to the actors. To this end, a simulation actor is allocated on each patch. Each actor is connected to the actors controlling the patches adjacent to its own. An example for the resulting domain decomposition is depicted in Figure 28. It shows a grid subdivided into \( 3 \times 3 \) patches, and \( 3 \times 3 \) actors coordinating them. Here, the connection between every two neighbouring actors is denoted by a single edge that stands for their bilateral communication. This communication is implemented through four channels, two in each direction (as depicted in Figure 29). In each direction, there is one channel for control information, such as termination messages, and one for the cell quantities of the ghost layer. The token capacity for each channel is set to two for all channel types in order to avoid deadlocks.
3 Characterisation and Analysis of Invasive Algorithmic Patterns

Figure 28: Decomposition of the simulation domain into $3 \times 3$ patches. Each patch is assigned an actor $A_{i,j}$ that is connected with its direct neighbours [13].

Figure 29: Actor Graph with $3 \times 3$ actors. Each actor $A$ (orange) is connected to its neighbours through four channels, one channel $D$ for ghost layer data exchange (depicted in blue) and one channel $C$ for control messages (depicted in green). Token capacity is set to two for all channels [13].

The actor graph is distributed onto the available X10 places using a block distribution. For the actors, there are different levels of functionality that may be added successively. The basic level assumes a fixed time step size, and full activity across the entire simulation domain. In a second step, lazy activation of actors is added, so that only actors that have been reached by the wave perform computations. In the basic model, actors trigger their neighbours through the ghost layer exchange. Whenever an actor is done with an iteration, it sends its ghost layer data to its neighbours. At the same time, each actor may compute an iteration once it has received ghost layer data from all its neighbours.
Lazy Activation of Actors  The basic actor model already introduces local coordination to the computation. Actors only depend on their direct neighbours, and coordinate amongst themselves. In the basic implementation of SWE-X10, an iteration is performed regardless of whether it is actually needed. In subsection 3.2.1 the hyperbolicity of the shallow water equations was mentioned. This type of partial differential equations exhibits wave-like behaviour, i.e. there is a finite propagation speed of the tsunami wave across the simulation domain. Cells that have not yet been reached by the wave will be at rest, i.e. there is no change in the water height or momenta. An actor may find out whether it is initially needed from the scenario: if its patch contains part of the initial perturbation, it should start as active, otherwise start as dormant. The set of active actors propels the simulation forward, and as soon as the wave reaches the boundary of a dormant actor’s patch, it is activated and joins the computation. In many cases, the initial perturbation will be limited to just a few patches. The remaining patches will only become active as the simulation progresses.

To enable this behaviour, the set of actions are adapted and the actors’ finite state machine are changed, as shown in Figure 30 to model the new behaviour. Initially, each actor determines its activity state, and sends it to its neighbours. Actors that contain part of the initial wave set their state to propagating wave, while those that do not set their state to lake at rest. Active actors perform simulation steps as described above. Unlike previously, ghost layer data is only needed from active neighbours now. The actor tests the activation conditions using the aggregate guard recvActive(). If an actor does not have any active neighbours, it may just keep computing updates until the wave reaches one of the boundaries of its patch by triggering itself at the end of its actions. During the computation of the update, the actor determines for each of its copy layers whether the update actually changes any values. If there are changes, or the neighbour is already active, the updated data will be sent at the end of the iteration. If the neighbour is still inactive and there is a non-zero update, the actor will also send a control message stating that it shall henceforth provide updates (sendActivation()). Once this message is received (recvActivation()), the neighbour in lake at rest state will change its state to propagating wave, and propagate its new status to all its neighbours, so that its other active neighbours know to expect updates from it, and may in turn send their updates. Once an actor reaches its termination condition, it sends a termination signal to all its neighbours, and terminates. The same happens if a termination signal is received (recvTerm()).
Characterisation and Analysis of Invasive Algorithmic Patterns

Evaluation

In the following, the benefits of lazy activation are demonstrated. The actors are distributed to their places at the start of the application. The CPU time used by the SWE-X10 application is measured by summing up the time used by each CPU to compute the scenario. This metric assumes that CPUs of inactive actors otherwise remain idle, and thus provides a rough approximation of the required energy to compute the solution. As a test scenario, a modified radial dam break scenario is chosen. In this scenario, the initial water elevation is placed in the lower left part of the simulation domain. The simulation time was set to 90 simulated seconds, a sufficient time for the wave to reach the entire domain. The grid resolution was set to 8192×8192 grid cells, distributed onto actors containing patches of 512×512 grid cells each. Actors were distributed onto eight CPUs on four nodes according to the scheme displayed in Figure 31a. The application recorded the activation time of the first actor for each X10 place. The time from the first activation to the end of the simulation was then used for each actor to compute the overall CPU time. For the configuration without lazy activation, the overall execution time times the number of CPUs in the computation yields the CPU time, as all CPUs are active and computing updates from the start.

For the run without lazy activation, an overall execution time of 1,433 seconds was measured. All actors are active from the start of the simulation, therefore, the overall CPU time utilised may be summed up to 12,264 CPU seconds, or 3.41 CPU hours. In comparison, the version with lazy activation takes 1,203...
3.3 Hybrid Application Mapping

To take advantage of the enormous computing power of multicore platforms, actor-based programming models as presented before and other parallel models are increasingly used. Each application is divided into several actors (or tasks) that can be executed simultaneously on different cores and exchange data with each other. Optimising application mapping, i.e. finding an appropriate assignment of application tasks to platform resources, is a particular challenge. This is due to the fact that the number of possible mappings for an application grows exponentially with the application size (number of tasks) and the platform size (number of cores).

Figure 31: Setup to test lazy activation of actors. The red circle marks the initial water displacement. Throughout the simulation, it will collapse and radiate outwards as a propagating wave. Each little square denotes an actor; the initial assignment of actors to places (i.e. CPUs), is marked in magenta [13, 14].

seconds to complete. Summing over the measured CPU activity (see Figure 32) yields a total number of 6,741 CPU seconds, or 1.87 CPU hours. Hence, in terms of CPU time spent, lazy activation enables significant savings in terms of used system resources. Naturally, the actual resource utilisation benefits of lazy activation heavily depend on the initial scenario. In cases where the initial disturbance is large and located in the centre of the simulation domain, the gains will be significantly smaller. However, typical tsunami simulation setups usually contain a relatively localised initial disturbance. Large parts of the simulation domain will be at rest initially, especially if propagation across the entire ocean is considered.
Application mapping methodologies for multicore/manycore systems can be classified into two categories [31]. In design-time (static) mapping approaches, all mapping decisions are conducted statically at design time (offline). These approaches employ compute-intensive optimisation and verification techniques to find an optimal mapping for each application which is also verified to satisfy the application’s requirements. As mappings are statically determined and these approaches are not adequate for the design of systems with a high dynamics and variation in application mix and application workloads. In run-time (dynamic) mapping approaches, all mapping decisions are made dynamically at run time (online). These approaches can take into account the current run-time situation in their mapping decisions and dynamically adapt to it. However, they are limited to lightweight mapping heuristics to find an acceptable mapping at a low computational effort.

Hybrid Application Mapping (HAM) (see Figure 33) combines static and dynamic mapping approaches to exploit the individual strengths of each [6, 16, 38, 39, 40, 43]. During a phase of Design Space Exploration (DSE), a set of Pareto-optimal mappings is computed for each application at design time where compute-intensive mapping optimisation and non-functional verification techniques are affordable [30]. These mappings are then used for run-time requirement enforcement (RRE), see proper definitions in Chapter 2, to dynamically execute the application on demand according to one of the precomputed mappings which fits best to the current system workload, application state, and resource availability [4, 29, 41, 42]. For a more theoretical background on speed-scaling for RRE, we refer to Chapter 4. In the following, we give a brief overview of techniques investigated in invasive computing.
3.3 Hybrid Application Mapping

Design Space Exploration (DSE)

symmetry-eliminating mapping exploration (binding, routing)

resource usage, ...

isolation-aware timing analysis

thermal-safety analysis

reliability, energy, ...

isolation-scheme exploration (allocation, scheduling)

mapping distillation

distilled set of mappings

mapping reconfiguration analysis

resource management

thermal-safety admission checks

reconfiguration admission checks

platform status

launch/termination request

set of mapping alternatives with pairwise reconfig. data

Figure 33: Design flow of Hybrid Application Mapping (HAM) with concept highlighted that enable a predictable (red), thermally safe (green), scalable (orange), and adaptable (blue) deployment of applications with timing requirements in heterogeneous multicore systems according to [15].

(illustrated in Figure 33) that enable a strictly predictable (red), thermally safe (green), scalable (orange), and adaptable (blue) deployment of applications with timing requirements in heterogeneous manycore systems.

3.3.1 Complexity Reduction

With the ample availability of resources on manycore systems and likewise growing application workloads in terms of number of parallel tasks and messages, the amount of possible mappings becomes immense. In particular, the large number of options for mapping tasks to resources contributes significantly to the size of the search space. Finding the best or even only an optimised mapping out of this huge search space is therefore a complex and time-consuming task. For large-scale manycore systems, there will be an increasing number of resources of equivalent resource types on chip, distrib-
Characterisation and Analysis of Invasive Algorithmic Patterns

uted in recurring patterns throughout the architecture. This means that the search space contains several mappings with equivalent resource requirements and non-functional properties. Schwarzer et al. [30] proposed to capture these symmetries by SAT formulations and to couple SAT solvers with Evolutionary Algorithms to perform the DSE. Richthammer et al. [25] additionally proposed a decomposition of the target architecture to significantly reduce the number of mapping possibilities so that speed-ups and quality improvements can be achieved.

Moreover, the offline-performed DSE in HAM approaches often considers many design objectives (quality objectives such as latency, energy, reliability and resource-related objectives [38, 40]). In effect, an immense number of Pareto-optimal mappings may result. To comply with restricted timing and storage of a run-time requirement enforcer, Pourmohseni et al. [17] proposed techniques for distillation (truncation) of the mappings set.

3.3.2 Composability

Resource contention between concurrent applications for accessing shared resources renders the timing behaviour of each application dependent not only on the arbitration policy of shared resources but also on the behaviour of other applications. Composability ensures that concurrent applications are separated and, therefore, cannot affect the (worst-case) timing behaviour of one another [2, 10]. In a timing-composable system, resources (or resource budgets) are exclusively assigned per running application so that concurrent applications are temporally and/or spatially isolated from each other. This enables that application mapping can be performed at design time for each application individually, i.e. without requiring knowledge about the concrete workload scenarios in which the application will be executed at run time. Invasive computing establishes dynamic isolation between concurrently executed applications to achieve timing composability by means of explicit resource reservation per application.

Pourmohseni et al. [19] demonstrated that a fixed isolation scheme often leads to sub-optimal mappings which suffer under-utilisation of resources. As a remedy, they proposed an isolation-aware application characterisation approach which introduces isolation-scheme exploration and isolation-aware timing analysis into the offline DSE in HAM (red boxes in Figure 33).

However, even when applications are temporally or spatially isolated, their power profiles may still lead to interdependencies due to transient temperature effects (e.g. creating hotspots affecting multiple applications). Pourmohseni
et al. [18] introduced thermal composability into HAM (green boxes in Figure 33) which performs a thermal-safety analysis as part of the DSE and then applies a thermal-safety admission check at run time to enforce thermal safety proactively. For more information on thermal management for invasive computing, we refer to Chapter 7.

### 3.3.3 Mapping Reconfiguration and Task Migration

Dynamically switching between different application mappings is possible in invasive computing using a fourth language primitive apart from invade - infect - retreat as introduced in Chapter 1 called re-invade: Here, each task has to be migrated from the resources of the source claim to those of the target claim. We have investigated the overhead of mapping reconfiguration and task migration in [20, 21, 22, 23] (blue boxes in Figure 33) which allows us to bound the time for migrating between two mappings, and thus enables us to give real-time guarantees even when such adaptations are dynamically applied at run time.

### 3.3.4 Scenario-aware HAM

For streaming applications, much of the run-time dynamism does not only arise from concurrently executed applications but also the input-dependent execution characteristics of the application. For example, the computational workload of object detection in images may depend on the input image and the features it contains as exemplified in Chapter 2. Though, due to the sheer number of possible input data, determining specific mappings for each input data at design time is typically infeasible. In invasive computing, we solve this problem by grouping input data into a fixed number of so-called data scenarios, which comprise input data with similar execution characteristics. Here, scenarios do not merely distinguish between the execution properties of the entire application, but also encapsulate subtle differences in execution properties at the task level. Therefore, two input data with the same total workload but different distribution of work among the tasks of the application may be divided into two different scenarios.

For each of these scenarios, mappings can now be computed that are tailored to the common structure of the data they comprise. We have developed an automated scenario-based hybrid application mapping displayed in Figure 34. Our methodology is designed to meet the application’s (soft) real-time requirements while minimising secondary objectives such as energy consumption. The approach is particularly targeted at streaming applications that receive
new input data periodically and the mapping can be adapted after each iteration. We specifically support black-box applications so that our approach can be generically applied to any application and is independent of any application-specific knowledge and code analysis. To achieve this, the approach is fully data-driven by making mapping decisions entirely based on observed non-functional execution properties as execution time and energy consumption. Therefore, all models and optimisations make use of machine learning and are performed on the basis of training data sequences that model the runtime input stream. The non-functional execution properties are gathered by profiling the applications in stand-alone mode for different inputs and mappings without needing knowledge about the code. Further, the system can be tuned to the specific needs of the application mix and architecture by a set of hyperparameters.

At run time, a run-time manager monitors the non-functional execution properties during the processing of input data. This feedback provides the necessary information to deduce the current data scenario and to determine the best-suited mapping for this scenario regarding the design objectives, e.g. the number of deadline misses and the energy consumption. If this mapping differs from the currently active mapping, the run-time manager triggers a mapping reconfiguration. In the next section, we present our scenario-aware DSE that determines the mappings and scenarios at design time by employing
3.3 Hybrid Application Mapping

Figure 35: Structure of the scenario-aware DSE. The optimisation cycle is entered with an initial partitioning of the training data set into scenarios. As long as no termination criterion is met, which we can assess over the validation data set, a DSE explores a set of mappings optimised for the current scenarios in each iteration. After distilling the mappings, they are used to refine the scenarios.

an iterative optimisation loop consisting of alternating mapping exploration and scenario clustering. Based on the output of the DSE, corresponding scenario selection and mapping selection models are generated automatically by applying data-driven techniques that make use of training data and machine learning.

Scenario-Aware Design Space Exploration Data scenarios can be determined by clustering input data based on observed non-functional execution characteristics such as latency and energy consumption. We can measure these characteristics by profiling the applications in stand-alone mode for different inputs and each considered mapping in a given mapping set. At the same time, the identified scenarios define the objective space for optimising this mapping set. Thus, exploring scenarios and optimising mappings form two interdependent optimisation problems. As a remedy, we have designed a scenario-aware DSE [33] that simultaneously optimises a partitioning of input data into scenarios while exploring scenario-optimised mappings that minimise latency and energy across these scenarios, as illustrated in Figure 35.

In order to tackle the interdependence, we define scenario and mapping optimisation as two separate subsequent tasks that are repeated iteratively. To enter the optimisation loop, we have to either create an initial scenario partitioning or an initial mapping set. Both are valid starting points. In this work, we generate an initial mapping set by random assignments since no further mapping information is given. The DSE has separated objectives for each
defined scenario, e.g. the average execution time and energy consumption in that scenario. We further incorporate the needed resources per resource type for the mappings as scenario-independent resource-related objectives. This is done as it enables the run-time manager to react to different resource occupations at run time. Instead of having a single DSE with separate objectives per scenario, it is also possible to perform a separate DSE per scenario and merge the results after the explorations have finished. Our intention of performing a single DSE is that this stimulates the exploration of mappings that cover multiple scenarios at once. Such mappings are advantageous since they provide a good trade-off across multiple scenarios and as such may reduce the cost of mapping reconfiguration.

As a reduced number of mappings is advantageous for exploring scenario partitionings, we apply a distillation procedure after DSE, which reduces the number of mappings in the mapping set. Two mappings that are differing only slightly can be substituted by a single mapping without sacrificing much coverage. Therefore, given $n$ training data samples, a mapping can be described by an $n$-dimensional vector storing the execution properties over the $n$ data samples. Mappings are clustered according to their vectors, and afterwards a representative mapping is chosen per cluster while the rest of the mappings is discarded.

The final step to conclude a cycle of the iterative optimisation is the calculation of the data scenarios — a process that we denote as scenario discovery. Scenario discovery requires a measure of similarities between input data instances for clustering the input data. Given $m$ distilled mappings and $\ell$ different non-functional characteristics obtained per mapping (e.g. end-to-end latency and energy consumption), we construct an $(m \times \ell)$-dimensional vector per training data that contains the $\ell$ non-functional characteristics of processing said data under each of the $m$ distilled mappings. By clustering these vectors using unsupervised learning, e.g. Gaussian mixture models [24] or $k$-means, we create partitions of the data set. These partitions represent the data scenarios that are used as the starting point of the next iteration loop.

Finally, we define termination criteria that end the optimisation cycle. Potential candidates are a fixed number of iterations or the convergence of some objective function over the validation data set. The validation data set is disjoint from the training data set. As a consequence, we can use the validation set to monitor the performance of the exploration on an unknown data set which is useful for avoiding overfitting the optimised models to the training data set. The scenario-aware design space exploration contributes to HAM and is therefore evaluated in the complete HAM context (see Figure 36).
**Scenario Selection**  Scenario selection is the process of deciding on the best-suited data scenario for processing subsequent input data based on the detected scenarios of the previously processed input data. Hence, this comprises two components: First, it consists of a scenario identification model that deduces the scenario of processed input data based on the evoked execution data characteristics. Second, it comprises a model that predicts a suitable scenario for future data based on these classifications.

The *scenario identification* problem can be modelled as a classification problem. We apply neural networks as they offer fast and accurate classification even for a small neural network architecture. We use the non-functional execution properties per task measured in the current iteration as input vector. The input layer thus contains one neuron for each entry in the non-functional execution vector. Each scenario is represented by a neuron on the output layer. Training uses input data samples labelled from the scenario-aware DSE. To facilitate training, useless input information can be omitted from this vector, e.g. execution time information of negligible tasks. Furthermore, neural network pruning mechanisms can be applied during or after training [9]. In our experiments, a few hidden layers with merely up to 64 neurons were sufficient to provide high accuracy.

The purpose of *scenario prediction* is to predict the scenario of the input data in the next application iteration. The most simple implementation of a scenario selection component is to always switch the identified scenario. However, this may be an insufficient model in several use cases. First, a scenario switch usually also triggers a mapping reconfiguration which may induce a significant overhead. Consequently, mappings should normally stay active for a couple of input data frames to mitigate the initial reconfiguration cost. Note that this is heavily dependent on the regarded application. For long-running applications, more frequent mapping reconfigurations may be feasible than for short-running applications with high reconfiguration overhead. Additionally, depending on the accuracy of the scenario identification model, erroneously identified scenarios may also trigger erroneous scenario switches. Scenario selection is therefore crucial to avoid erroneous scenario switches and thus stabilise the scenario selection process. This can be achieved by only switching to a new scenario if multiple of the previously processed input data indicate a scenario switch. As such, there is no correct scenario selection procedure that works best for every possible application. Instead, the best-suited scenario switching strategy is explored by our system automatically that minimises the design objectives as number of deadline misses and energy consumption.
We evaluated several design alternatives for scenario selection models [34]. Decision trees created the lowest overhead but also lacked in terms of scenario selection quality. Genetic programming has a low overhead and achieved the best performance. Neural networks also supplied top performance but imply a higher overhead for the run-time manager. Finally, reinforcement learning models offered a high overhead and the worst performance. As the scenario-aware design space exploration, scenario selection is evaluated in the context of the complete HAM flow (see Figure 36).

**Mapping Selection**  Mapping selection is the process of choosing a mapping out of the set of Pareto-optimal mappings given the scenario predicted by scenario selection with the goal to minimise deadline misses and energy consumption for the processing of subsequent input data. A data-driven mapping selection approach has been proposed by Spieck et al. [32].

Mapping selection is based on the measured per-task execution characteristics of the \( w \) previous iterations and the predicted scenario. In this context, the size \( w \) of this history window can be configured as a hyperparameter. In our experiments, we achieved best results with a window length of two to three. We assessed several techniques for mapping selection [32] and had the most consistent selection quality with a neural-network-based solution. The training data for the neural network is prepared by sliding a window of the defined length \( w \) over an available training data sequences. For each window segment, we determine the per-task execution characteristics of the input data of the previous \( w \) iterations. Furthermore, scenario selection predicts the best-suited scenario for the next iteration. Execution data and predicted scenario serve as the input of the neural network. The mapping which can process the current input within the deadline with minimal energy consumption is then determined to be classified as output for this input. The best interplay between the scenario and mapping selection model is reached when the window size for both models is identical. A mappings selection methodology for multiple concurrently running applications following this scheme is presented in [36].

For evaluation, we used two streaming applications with soft real-time deadlines: ray tracing (consisting of 11 tasks) and stitching (comprising 31 tasks). The experiments were performed on a heterogeneous MPSoC architecture with 4×4 tiles. Its resources are arranged in a 2D network-on-chip (NoC) similar to existing tile-based architectures like the Tilera TILE-Gx [37], Kalray MPPA-256 [3], and Intel SCC [8]. On each tile, a dual-core processor of one of the three distinct processor types PowerPC, AMD-S, and AMD-B is mounted that differ in computing power and energy consumption. The execution of
3.3 Hybrid Application Mapping

these applications is simulated with a heterogeneous MPSoC simulator [26, 28] targeting NoC-based heterogeneous manycore architectures. We compared the following approaches: *Avg* uses a fixed scenario-agnostic mapping optimisation approach that applies a static mapping optimised for the average case over all input data [38]. *Fix* uses our scenario selection module to select the best-suited scenario per input data and then chooses a fixed mapping per scenario. *NN* is our proposed solution that combines scenario selection and mapping selection so that different mappings can be chosen per scenario. Finally, *Opt* is the theoretical optimum that can only be achieved by a run-time manager with perfect information about the present and future data. We give *Opt* unlimited optimisation budget to assess the execution characteristics of all mapping alternatives and then pick the best-suited one. Obviously, the overhead of calculating the best-suited mapping is not included into the latency of *Opt* while the overhead of the selection algorithms is included for the rest of the approaches.

The resulting Pareto fronts for the ray-tracing application are shown in Figure 36 and the stitching application in Figure 37. For both applications, our proposed solution *NN* outperforms the other baselines *Avg*, *Fix*, and *NN* with regard to deadline misses and energy consumption. *NN* even comes close to *Opt* for some scenarios. But even *Fix* that relinquishes the mapping selection aspect presents a valuable improvement over the scenario-agnostic *Avg*.

**Domain-Adaptive Hybrid Application Mapping** Because of the sheer abundance of input data, only a small subset of possible input data can be used for the design-time training of the scenario-aware models. As a consequence, a shift in the data distribution between training-time data (source domain) and the data actually observed at run time (target domain) may occur and lead to a suboptimal execution. This shift may arise in particular when using the models in another environment, e.g. image processing taking place during the day (source) compared to at night (target), or on a sunny day (source) compared to a rainy day (target). To retain the desired execution qualities, one might perform a complete re-design of the system with the new data. Though, a lack of labelled data or the high cost of generating these labels often constrains this re-design. For that reason, we developed a system using unsupervised domain adaptation (DA) techniques to retrain the models by using the original labelled data from the source domain and unlabelled data from the target domain. This enables our system to significantly improve upon the mapping selection quality in the target domain without the requirement of labelled target data or a holistic re-optimisation for the target domain, which would introduce a high optimisation overhead.
Figure 36: Pareto fronts for the proposed mapping selection approaches compared to the baseline Avg and theoretical optimum Opt for the ray-tracing application under different deadlines.

Figure 37: Pareto fronts for the different mapping selection approaches compared to the baseline Avg and theoretical optimum Opt for the stitching application under different deadlines.
For our scenario and mapping selection models, we evaluated multiple different variants of unsupervised domain adaptation. The two best-suited models DANN [5] and CDAN [12] follow the principle of adversarial domain adaptation. Adversarial domain adaptation networks operate by transforming the representation of the input such that features emerge that are domain-invariant but discriminative with regard to the scenario and mapping selection. The corresponding model consists of three subnetworks as shown in Figure 38: The purpose of the feature extractor is to take the per-task execution characteristics of processing the current input and transform it into a domain-invariant and discriminative feature vector. The main network uses this feature vector to predict the best-suited scenario or mapping for the processed input data. The domain classifier discriminates whether the incoming data belongs to the source or target domain. This model is now trained in a fashion that the domain classifier is unable to distinguish between feature vectors of the source and target domain. This means that the output of the feature extractor is domain-invariant and hence the main network is trained on characteristics of the data that are valid both in the source and target domain. For further implementation details, we refer to Spieck et al. [35].

Since we have separate models for scenario selection and mapping selection, we use two domain adaptation networks. For scenario selection, the main network is constituted by a feed-forward classification neural network. Its output layer contains as many neurons as scenarios. The probabilities of the input data belonging to each scenario can then be assessed by a softmax function. For mapping selection, a classification network could be used where each output neuron represents one potential mapping candidate. However, a better network structure is given by a regression network that tries to predict the mapping index in a list of mapping candidates ordered by their average
execution time. When having soft real-time deadlines, it is decisive to select mappings that keep the execution time under the given deadline. With a regression network, the distance between the execution time of the selected mapping to the deadline can be expressed directly in the loss function. This facilitates training and makes the output more robust.

We evaluated our proposed domain-adaptive HAM methodology Proposed against three baselines. Static is a mapping methodology that explores a fixed mapping at design time that is laid out for the average input data [33]. Source is the state-of-the-art scenario-aware HAM approach [32]. Target is the theoretical optimum which is a complete redesign by performing the scenario-aware HAM approach exclusively on the data from the target domain.

As applications, we used the same two streaming applications as described afore. Beside the previously used 4×4 MPSoC, we also evaluated our domain-adaptive HAM on an 8×8 MPSoC to assess the scalability of our approach. The resulting Pareto fronts for minimising the deadline misses and energy consumption of processing an input stream of 10,000 test data items from the target domain are displayed in Figure 39.

![Figure 39: Pareto fronts regarding deadline misses and energy consumption for the evaluated mapping selection approaches.](image)
3.4 Conclusions and Acknowledgements

Clearly, Static is outperformed by each of the scenario-aware procedures. This is a testament to the importance of a scenario-aware HAM approach. Since we test our setup with data from the target domain which exhibits a significant domain shift compared to the source domain used at design time, Target is consistently better than Source for all test setups. Proposed does an excellent job in closing the gap between the two approaches, e.g. for a deadline miss percentage of 6% for the stitching application, the energy consumption of Proposed is more than 10% lower than of Source for both MPSoC sizes. Nevertheless, there is still some room for improvement compared to Target. Though, this is to be expected since Target is a complete redesign to the target data, which inter alia comprises novel mappings tailored to the target domain explored by a new DSE execution. A complete redesign, however, entails a very high optimisation time and further requires labelled data. The target domain DSE necessitated more than a week for convergence with an Intel i5-6600K CPU. Contrarily, the proposed domain adaptation approach was completed in a couple of hours for both applications. No scalability issues can be detected when comparing the results for the 4×4 and 8×8 architectures.

This chapter presented techniques to adapt the execution scheme and mapping of applications at run time to optimise execution properties and, at the same time, comply with a set of non-functional requirements. We first presented an actor-based programming model. We demonstrated how to use it to provide an application execution scheme that can be adapted to input-dependent variations. For the chosen proxy application from the HPC domain, we significantly reduced the overall execution time and CPU time by adopting an input-dependent execution scheme called lazy activation.

Furthermore, we introduced recent results on hybrid application mapping (HAM) that divides application mapping into a design-time and a run-time part. HAM enables dynamic run-time modifications of the resource claim to react to changes in environment and input data and thus enable run-time requirement enforcement (RRE). We gave insights on applying this scheme for loose RRE of soft real-time applications, namely image stitching and ray tracing. Our evaluation shows that it is possible to significantly reduce the energy consumption as well as the percentage of misses of the soft deadline by using the proposed techniques.

At this point, we would like to thank Jürgen Teich and Andreas Weichslgartner for their many valuable contributions, particularly to the topics of application
characterisation and hybrid application mapping. Indeed, several publications, as listed in the following, resulted from tight cooperation between the author teams of Chapter 2 and Chapter 3.

References


4 Scheduling Invasive Multicore Programs Under Uncertainty
Nicole Megow, Alexander Lindermayr, Bertrand Simon

Abstract In this chapter, we present algorithmic and complexity theoretic results for coping with different types of uncertainty when scheduling on multicore systems. We consider challenges arising due to unknown task execution times, uncertain task sets as well as uncertain processor availability or speed. Coping with such uncertainties is a key challenge when allocating and scheduling resources using invasive programming concepts. We design scheduling algorithms that obtain under uncertainty provable performance guarantees. We also overcome barriers from traditional worst-case analysis by exploiting structural properties of typical practical input instances and by integrating (possibly erroneous) predictions, e.g. provided by machine-learning algorithms, into decision making. Further, we quantify the performance of dynamic speed scaling mechanisms as a tool for run-time requirement enforcement. We complement our theoretical findings by empirical experiments.

4.1 Introduction
Coping with uncertainties is a key challenge when scheduling and allocating resources in invasive computing environments. Indeed, uncertain inputs as well as dynamically changing machine or environmental properties are an omnipresent issue when executing programs on multicore systems: task sets or their dependencies might be uncertain, arrival times are unknown, execution times are often over- or underestimated, resource availabilities or processing speed may not be known, etc. It is of major importance to the system performance to design algorithms that perform well even under uncertainty.

The concept of invasive computing offers programs a temporally exclusive claim of resources. This admits a better control on the resource usage and an analysis of non-functional program properties without interference from other programs. However, this comes at a price: applications dynamically claim (invade) and release (retreat) resources in a decentralised way, which causes new uncertainties. The truly available number of processors and their
characteristics, such as processing speed, may not be known at design time as it depends on the entire system state and other applications being executed. Further, there can still be observed large fluctuations in execution time, power consumption and other properties of programs on multicore systems which are due to variations of workload (e.g. program input), as well as machine (e.g. power manager) and environmental states; see the introduction in Chapter 1.

In this chapter, we highlight results from several lines of research addressing different types of uncertainties in scheduling programs on multicore systems. We consider the challenges arising due to uncertain execution times (Section 4.2), uncertain task graphs (Section 4.3) and uncertain processor availability and speed (Section 4.4). Further, we use dynamic speed scaling as a tool for run-time requirement enforcement (see the discussion also in Chapter 2) in Section 4.5. Our chapter contains provable guarantees on the performance of algorithms coping with uncertainty as well as complexity theoretic results justifying certain assumptions or restrictions in the system design. From a conceptual point of view, we address online decision making at run time, robust decision making at design time and we integrate predictions of unknown quality (e.g. provided by machine-learning algorithms) about unknown input into decision making. With the latter we follow a very recent and vibrant line of research and, therefore, give more emphasis to it (in Section 4.2).

### 4.2 Uncertain Execution Times

Processing requirements of jobs are often fluctuating or unknown. *Non-clairvoyant* scheduling refers to scheduling algorithms that are designed in such a way that they require no knowledge about the processing times of jobs. In contrast, *clairvoyant* algorithms have access to the precise processing requirements. Non-clairvoyant scheduling is a fundamental task and has been studied extensively in many variations [8, 18, 19, 22, 32].

We consider non-clairvoyant scheduling with the objective of minimising the sum of weighted completion times. Here, the completion time of a job is the first point in time when it received an amount of processing equal to its total processing requirement. More precisely, we are given a set of $n$ jobs, each job arriving online at its release date, with individual weights and with unknown processing requirements. These jobs must be scheduled preemptively on a single or identical parallel machines. We also consider non-clairvoyant scheduling on most general heterogeneous machines, typically called unrelated machines, where jobs may have very different processing
requirements on each of the machines, but a machine-dependent processing rate is given.

We assess the performance of an online algorithm by the worst-case framework of competitive analysis. An online algorithm is $\rho$-competitive if, for all instances $I$, the objective value of the algorithm $\text{Alg}(I)$ for instance $I$ satisfies $\text{Alg}(I) \leq \rho \cdot \text{Opt}(I)$, where $\text{Opt}(I)$ is the objective value of an optimal solution for $I$.

Non-clairvoyant algorithms typically assign processing rates to jobs that sum up to at most one per machine and per job. They describe the relative amount of processing a job receives in every sequential time interval of a certain length. The most prominent strategy is the Round-Robin (RR) algorithm, which assigns equal rates to all alive jobs. While in practice the length of a time interval of a round is an implementation detail that correlates with the computational effort, in theory strong lower bounds rule out constant competitive algorithms unless these intervals are infinitesimally small. Under this assumption, which is called time sharing, RR is 2-competitive for the simplest problem variant (single machine, no release dates, equal weights) which is best possible [32]. The same guarantee is possible using a natural generalisation of RR to weighted jobs [22] and/or to identical machines [8, 32]. Scheduling on unrelated machines is much harder and requires careful migration between machines [15]. Nevertheless, it is possible to compute rates proportional to job properties and machine constraints and obtain $O(1)$-competitive algorithms [18, 19].

### 4.2.1 Learning-Augmented Algorithm Design

Nowadays, the assumption of non-clairvoyance seems too strong for many applications. While the exact processing time might be unknown, often some estimate is available, e.g. extracted information from past data and/or predictions obtained by machine-learning algorithms which have advanced tremendously in the past years. One can expect that such predictions are often of high quality but there is no guarantee and they might be arbitrarily wrong. Therefore, solely relying on such information brings large risk.

The recently emerging line of research on learning-augmented algorithms proposes to design algorithms that have access to (possibly erroneous) predictions on uncertain input. The goal of these algorithms is to achieve an improved performance if the prediction is accurate while performing not much worse than algorithms without access to predictions, if the predictions are completely wrong. Ideally, the performance of a learning-augmented
algorithm is a function of the quality of the prediction for some well-defined error measure. Here, finding a useful prediction and defining an appropriate error measure is a key task. Given a definition for the prediction error $\eta \geq 0$ that quantifies the quality of the prediction for a given instance, the goal is to express the competitive ratio of the algorithm by a monotone function $f(\eta)$. A learning-augmented algorithm is called $f(0)$-consistent, in case of perfect prediction, and $\beta$-robust, if $f(\eta) \leq \beta$ for all possible errors $\eta$. Starting with the work by Lykouris and Vassilvitskii [26, 27], algorithms with predictions became a very active research area over the last couple of years; see [31] for a first survey.

First results on non-clairvoyant scheduling with predictions [20, 33, 37] concern the single-machine problem (without weights and release dates) with predicted processing requirements $\{y_j\}_{j \in J}$ for a job set $J$, which we call length predictions. In their seminal paper [33], Kumar et al. propose an algorithm that is controlled by a parameter $\lambda \in (0, 1)$, which can be seen as an indicator of the algorithm’s trust in the accuracy of the prediction. Measuring the quality of a prediction $\{y_j\}_{j \in J}$ w.r.t. the actual processing requirements $\{p_j\}_{j \in J}$ by the $\ell_1$ metric ($\ell_1 = \sum_{j \in J}|p_j - y_j|$), their algorithm has a competitive ratio of at most $(1/(1-\lambda))(1 + n\ell_1/\text{Opt})$ while also maintaining a robustness factor of $2/\lambda$. Note that $\ell_1$ and $\text{Opt}$ depend on a concrete instance in the competitive ratio, which we omit in the notation. However, the $\ell_1$-metric does not seem to distinguish well between “good” and “bad” predictions, as has been noted recently by Im et al. [20]. They argue that, intuitively, the linear error measure $\ell_1$ is incompatible with the sum of (weighted) completion time objective and using $n \cdot \ell_1$ as upper bound, as Kumar et al. do, may overestimate the “actual” error, substantially.

Im et al. [20] propose a different error measure $\nu$ that satisfies certain desired properties and is based on the optimal solution of artificial instances mixing $y_j$ and $p_j$. It satisfies $\ell_1 \leq \nu \leq n\ell_1$. Using this error, they design a learning-augmented randomised algorithm with guarantees depending on $\nu$. Their algorithm is quite sophisticated, requires large constants to diverge from RR, and it seems very challenging to generalise it to scheduling settings with release dates, weights or even heterogeneous machines. Further, the error measure $\nu$ is still sensitive to changes in the predicted job lengths which would not affect an optimal schedule at all which seems an undesired property.

We contribute to the line of research on learning-augmented online algorithms for non-clairvoyant scheduling problems in all three relevant aspects: We propose a novel prediction model for scheduling problems, introduce a meaningful error measure for it and finally design actual algorithms for scheduling
weighted jobs on heterogeneous machines. We give an overview in the following subsections and refer to the full version [24] for details.

### 4.2.2 Prediction Model and Error Definition

We introduce a novel prediction model for scheduling problems, called *permutation prediction model*. Intuitively, it provides a permutation of jobs suggesting that an algorithm should use it as a priority order for scheduling. The idea is that, instead of predicting job lengths, we take structural properties of an input instance into account that an optimal algorithm may exploit. For minimizing the sum of weighted completion time, the *Weighted Shortest Remaining Processing Time (WSPT)* order, i.e. jobs in order of weight over processing time ratios, has proven to be useful in various settings. Indeed, for the non-clairvoyant version of the single-machine problem without release dates, knowing the WSPT order of jobs would be sufficient to determine an optimal schedule [35]. While this knowledge is not sufficient for optimally scheduling with release dates and/or on multiple machines, it still admits strategies with good approximations on an optimal solution [2, 16, 29].

Formally, for a scheduling instance with job set \([n] := \{1, ..., n\}\) and a single or multiple identical machines, our prediction is a permutation \(\hat{\sigma} : [n] \rightarrow [n]\) of all jobs. Due to the aforementioned power of the WSPT order, we call the associated permutation of jobs, \(\sigma\), *perfect prediction*. On unrelated machines, the job-to-machine assignment crucially matters. Therefore, we add such an assignment to our prediction. In this most general model with \(m\) machines, our prediction is defined as \(\hat{\sigma} = \{\hat{\sigma}_i\}_{i \in [m]}\), where \(\hat{\sigma}_i\) is the permutation of jobs assigned to machine \(i\). Given a scheduling instance without release dates and the optimal job-to-machine allocation, we speak of a perfect prediction \(\sigma = \{\sigma_i\}_{i \in [m]}\), if \(\sigma_i\) involves exactly those jobs that are scheduled in an optimal solution on machine \(i\) and orders them in WSPT order, for each \(i \in [m]\).

In the permutation prediction model, jobs still arrive online and, at any time, an algorithm has access only to predictions on jobs that have been released already. At any release date, the permutation is updated consistently, that is, the relative order of previously known jobs remains fixed.

The definition of the prediction error is a crucial element in the design of algorithms with predictions. Intuitively, the error measure shall quantify the impact that an imperfect prediction has on an (optimal) scheduling algorithm. It is not unnatural to express the error as \(|\text{Opt}(\hat{\sigma}) - \text{Opt}(\sigma)|\), as has been done for other problems [5, 13, 25]. The disadvantage of this approach is that
this error could be even negligible whereas the impact of running an optimal algorithm with the wrong prediction could be significant.

In more detail, our error measure shall capture the change in the cost that an optimal schedule must face when two jobs \( j \) and \( j' \), with (unknown) processing times \( p_j, p_{j'} \) and weights \( w_j, w_{j'} \), are inverted in a prediction \( \hat{\sigma} \) with respect to \( \sigma \). For example, on a single machine without release dates, if \( j \) and its successor \( j' \) in \( \hat{\sigma} \) are swapped in \( \sigma \), the schedule that follows \( \hat{\sigma} \) pays an additional cost of \( w_{j'} p_j \) but saves \( w_j p_{j'} \) compared to the schedule that follows \( \sigma \), where \( w_j \) denotes the weight of job \( j \). Our error measure \( \eta^S \) for identical machines captures exactly this increase of the objective value due to inverted job pairs.

In presence of release dates and/or on unrelated machines, just knowing the order may not allow us to express the change in the exact optimal cost. Therefore, we approximate the optimal cost by the change in the sum of weighted completion times when preemptively scheduling jobs in the given priority order, \( \hat{\sigma} \) resp. \( \sigma \). This idea leads to our general error \( \eta^R \). We remark that \( \eta^S \) can be seen as a special case of \( \eta^R \) where no job-to-machine assignments are given and there are no individual release dates.

Our defined error measure satisfies desired properties such as (i) monotonicity, (ii) Lipschitzness, (iii) theoretical learnability, and (iv) practical learnability. Im et al. [20] advocate particularly the first two properties.

Monotonicity requires, in the length prediction model, that the error grows as more predictions become incorrect. In our setting, adding an inversion \((j', j)\) increases the error by \( w_{j'} p_j - w_j p_{j'} \), since \( \sigma(j') < \sigma(j) \) implies \( w_{j'} / p_{j'} \geq w_j / p_j \). Thus, the error grows, and our definition satisfies monotonicity.

Lipschitzness requires the error to be sensitive to the cost of the optimal solutions for the actual and predicted instance. Our error measures immediately satisfy Lipschitzness for our prediction setup, as it precisely measures the cost between a solution that follows \( \hat{\sigma} \) and one that follows \( \sigma \), when scheduling the actual instance preemptively according to the given order.

Our prediction model is theoretically learnable: we show that permutations are efficiently PAC-learnable in the agnostic sense w.r.t. our error definition. This essentially says that there exists an efficient algorithm which, given a polynomial number of samples on the quantity to predict, outputs with high probability a prediction which is close, regarding our error definition, to the best possible prediction for the underlying distribution. However, this theoretic result gives a rather large bound on the required number of samples to get a low prediction error. Therefore, we further demonstrate that our
predictions are *learnable and useful in practice*. We show that even a few number of seen samples lead to very good performance of our algorithms.

In general, it is difficult to compare different prediction and error models. However, we can convert a given length prediction into a permutation prediction by computing the WSPT order based on the predicted processing requirements \( \{y_j\}_{j \in J} \) for an instance \( J \). This allows us to compare our error to the previously proposed measures \( \nu \) and \( \ell_1 \) for the unweighted single-machine problem.

Firstly, our error \( \eta^S \) is less vulnerable than \( \nu \) and \( \ell_1 \) to changes in the predicted instance which do not affect the *structure* of an optimal solution. To see this, consider an instance with \( p_j = j \) and a predicted instance with \( y_j = j - 1 \) for all \( j \in [n] \). The optimal solutions of both instances have the same structure, giving \( \eta^S = 0 \), while \( \nu = \text{Opt}(\{\max\{p_j, y_j\}\}) - \text{Opt}(\{\min\{p_j, y_j\}\}) = n(n + 1)/2 - (n - 1)n/2 = n \) as well as \( \ell_1 = n \). This shows that our prediction and error seem to capture well the relevant characteristics of an input-prediction in terms of derived actions, while \( \nu \) and \( \ell_1 \) also track insignificant numerical differences between the actual and predicted instances.

In contrast, there are other instances where \( \nu \) and \( \ell_1 \) underestimate the actual difficulty that is caused by the inaccuracy of the prediction given to an (optimal) algorithm. Im et al. [20] give such an example with \( p_1 = y_1 = \ldots = p_{n-1} = y_{n-1} = 1 \) and \( p_n = n^2 \) but \( y_n = 0 \). While the structural difference of the optimal solutions for predicted and true values is large (\( \Omega(n^3) = \eta^S \)) the other error definitions only measure \( \nu = n^2 + n \) and \( \ell_1 = n^2 \).

### 4.2.3 The Preferential Time Sharing Framework

We describe a framework for designing learning-augmented algorithms for non-clairvoyant scheduling with untrusted predictions, and give rigorous performance guarantees for several concrete scheduling settings.

In their seminal paper, Kumar et al. [33] propose a single-machine time sharing algorithm for preemptively combining two algorithms, a clairvoyant algorithm that follows predicted processing times and a non-clairvoyant algorithm. The rate, at which each of these algorithms is executed, is controlled by the confidence parameter \( \lambda \in (0, 1) \). Intuitively, a small value for \( \lambda \) represents high trust in the prediction, while a large value refers to the case that the predictions might be incorrect and should not be trusted. We extend this idea to a general framework for scheduling jobs with non-uniform weights and arbitrary release dates on unrelated machines. Here, we require (as Kumar et al. [33]) that both algorithms are *monotone*. An algorithm is monotone if,
given two instances with identical inputs but actual job processing requirements \( \{p_1, \ldots, p_n\} \) and \( \{p'_1, \ldots, p'_n\} \) such that \( p_j \leq p'_j \) for all \( j \in [n] \), the objective value of the algorithm for the first instance is at most its objective value for the second one.

Given two monotone algorithms \( \mathcal{A} \) and \( \mathcal{B} \) and a confidence parameter \( \lambda \in (0, 1) \), we define a new preemptive algorithm that runs in every infinitesimal time interval in the first \((1 - \lambda)\)-fraction algorithm \( \mathcal{A} \) on all machines and in the remaining \( \lambda \)-fraction of the interval algorithm \( \mathcal{B} \), on all machines. It also hides arrived jobs until they are released in the simulated, i.e. slowed down, schedule of \( \mathcal{A} \) resp. \( \mathcal{B} \). As a result, assuming that \( \mathcal{A} \) is \( \rho_{\mathcal{A}} \)-competitive and \( \mathcal{B} \) is \( \rho_{\mathcal{B}} \)-competitive, the new algorithm has a competitive ratio of at most \( \min\left\{ \frac{\rho_{\mathcal{A}}}{1 - \lambda}, \frac{\rho_{\mathcal{B}}}{\lambda} \right\} \) for the same problem.

Our Preferential Time Sharing framework crucially builds on this technique and takes as input two monotone algorithms, a clairvoyant algorithm \( \mathcal{A}^C \) with a competitive ratio of at most \( \rho_C \) and a non-clairvoyant algorithm \( \mathcal{A}^N \) with a competitive ratio of at most \( \rho_N \). Intuitively, the non-clairvoyant algorithm will ensure robustness, while the clairvoyant algorithm, following a given prediction, gives a good consistency if the prediction is correct. As \( \mathcal{A}^C \) will have access to predictions while being oblivious of true processing requirements, we call it prediction-clairvoyant. Our framework then gives, with \( \mathcal{A} = \mathcal{A}^C \) and \( \mathcal{B} = \mathcal{A}^N \), a time sharing algorithm with consistency \( \rho_C / (1 - \lambda) \) and robustness \( \rho_N / \lambda \). When aiming for error-sensitive guarantees, we require an error-dependent performance guarantee. We say that a prediction-clairvoyant algorithm is \( \eta \)-error-dependent for an error measure \( \eta \) if its objective value is bounded by \( \rho_C \cdot \text{Opt}(J) + \eta(J, \hat{\sigma}) \) for any instance \( J \) and prediction \( \hat{\sigma} \).

We obtain the following results when applying the Preferential Time Sharing framework to different concrete scheduling problems.

**Theorem 1.** Preferential Time Sharing is, for every \( \lambda \in (0, 1) \), a learning-augmented non-clairvoyant online algorithm for minimising the total weighted completion time on

1. a single machine without release dates with a competitive ratio of at most

\[
\min \left\{ \frac{1}{1 - \lambda} \left( 1 + \frac{\eta^S}{\text{Opt}} \right), \frac{2}{\lambda} \right\},
\]
4.2 Uncertain Execution Times

2. \( m \) identical machines with release dates with a competitive ratio of at most

\[
\min \left\{ \frac{1}{1 - \lambda} \left( 2 + \frac{\eta^S}{m \cdot \text{Opt}} \right), \frac{3}{\lambda} \right\}, \text{ and }
\]

3. unrelated machines with release dates with a competitive ratio of at most

\[
\min \left\{ \frac{1}{1 - \lambda} \left( 5.8284 + \frac{\eta^R}{\text{Opt}} \right), \frac{128}{\lambda} \right\}.
\]

These results can be interpreted as follows. Consider the first result for a single machine. In the case of good predictions, i.e. a small error \( \eta^S \), our algorithm is guaranteed to achieve a performance of roughly \( 1/(1 - \lambda) \), which can be arbitrarily close to an optimal solution for small values of \( \lambda \). This bound then gets worse for a growing prediction error \( \eta^S \), but at some point, it hits the robustness bound \( 2/\lambda \), which then guarantees constant performance, even for arbitrarily bad predictions. The confidence parameter \( \lambda \) essentially controls how fast, in terms of growth of the prediction error, the robustness bound \( 2/\lambda \) is dominating in the minimum, and therefore how much the consistency factor \( 1/(1 - \lambda) \) differs from the robustness bound.

4.2.4 Experimental Evaluation

We validate our theoretical findings for a single and parallel identical machines in empirical experiments.

Dataset We generate synthetic instances, where each instance is composed of 1000 jobs. This size gives the algorithms enough jobs to work properly while it does not incur too much computational effort. The processing requirements for the jobs are individually sampled from a Pareto distribution with scale 1 and shape 1.1. This distribution was used in the seminal work on learning-augmented scheduling [33] and is (similar to the related Zipf distribution) generally considered to model scheduling applications very well [1, 6, 9, 11, 17, 21, 23]. Intuitively, it gives many tiny jobs and few very large jobs.

Types of experiments We perform two types of experiments. In sensitivity experiments, we generate length predictions by adding Gaussian noise to the processing requirements with an increasing standard deviation \( \omega \) for a fixed instance. Kumar et al. [33] also performed this kind of experiment, which became common in the area of learning-augmented algorithms [3, 4, 25, 27].
In online learning experiments we first fix a synthetic instance, called base instance. Then, we consider 10 rounds, where in every round \( t \) an instance \( J_t \) arrives, which is generated by adding independently sampled Gaussian noise to the base instance. For a true processing requirement \( p \), this noise has a standard deviation equal to \( \gamma \cdot \sqrt{p} \) for a parameter \( \gamma \geq 0 \). The prediction for round \( t \) is then the instance of average processing times of the set of previous instances \( \{J_0, \ldots, J_{t-1}\} \). In the initial round we use a random instance as prediction. This type of experiment was also performed by Dinitz et al. [10] to demonstrate the speedup of predictions for the bipartite matching problem.

**Results for a single machine**  We first present implementation details of the considered algorithms. As online benchmark (without predictions), we use the best-possible non-clairvoyant algorithm Round-Robin (RR) [32].

**TwoStage** (algorithm by Wei and Zhang [37]) executes RR until a certain time point depending on the predicted processing requirements and the confidence parameter \( \lambda \in [0, 1] \). Then, it schedules the jobs in non-decreasing order of their predicted processing requirements. If at any time a job finishes before or after its predicted length, it finishes the remaining instance with RR.

**MultiStage** (algorithm by Im et al. [20]) processes jobs in subsequent phases and decides whether to follow the prediction or to execute RR by tracking quality of the prediction. This is done by processing and computing the error of small random samples, which is then projected to the whole set of remaining jobs. We implemented a basic variant of this algorithm, which is \( O(1) \)-robust and \( (1 + \epsilon) \)-consistent for any \( \epsilon > 0 \) with high probability under some assumptions (Corollary 34 in [20]). Our implementation uses base two for unspecified logarithms. A consequence of this choice is that if \( \epsilon < 0.215 \), MultiStage executes solely RR on our instances. Therefore, we performed the experiments with rather large \( \epsilon = 0.25 \) and \( \epsilon = 10.0 \).

On a single machine with uniform job weights, the Preferential Time Shar-
ing (PTS) algorithm essentially matches the Preferential-Round-Robin (PRR) algorithm of Kumar et al. [33]. The only difference between both is the theo-
etically different prediction model. However, since the other algorithms use the length prediction model, we compute permutations based on predicted processing times. Thus, PTS and PRR behave equally in our experiments.

For every parameter setting we perform 10 runs and report empirical competitive ratios with corresponding error bars that denote the 95% confidence interval of the runs. The empirical competitive ratio is the average objective value of an algorithm over all runs divided by the optimal objective value.
We first discuss results of the sensitivity experiment, which are visualised in Figure 40. In the case of having access to a perfect prediction ($\omega = 0$) the algorithms achieve their best performance, as expected. However, even for very small noise ($\omega = 0.1$), the performance of TwoStage and MultiStage drastically drops compared to having access to precise predictions. This is by design: these algorithms switch their execution to the robust fallback procedure RR when detecting incorrect predictions. While TwoStage stays in this mode until the instance completes, MultiStage still estimates medians and errors, incurring an additional overhead. While the performance of PTS smoothly degrades for larger noise depending on $\lambda$, it still outperforms RR until $\omega \approx 20$. For very large noise, the performance of TwoStage and MultiStage stays unchanged, while PTS with $\lambda = 0.1$ still grows. For larger values of $\lambda$, e.g. $\lambda = 0.66$ as in the figure, PTS shows a constantly superior performance than MultiStage and, w.r.t. TwoStage, a smoother performance with substantially better consistency and only slightly larger robustness.

In the online learning experiment (Figure 41), TwoStage and MultiStage do not improve their performance over RR by using predictions. We suspect that this is again due to the fact that the prediction is still too erroneous over the first ten rounds to activate their trustful subroutines. We performed these experiments also with 100 rounds, but did not observe a significant difference. While in the initial round with the random prediction PTS performs slightly worse than the other algorithms, it improves over RR already after seeing one sample. This shows that in our setup one sample is enough to approximately distinguish small jobs from large jobs, and this classification is enough to prevent large jobs from delaying the completion of many small jobs. This also demonstrates that permutation predictions capture the relevant information of practical instances.
Results for multiple machines  We generate 10 synthetic instances with 1000 jobs each. Processing requirements are again sampled from a Pareto distribution with shape 1.1 and scale 1, weights and release dates from a Pareto distribution with shape 2 and scale 1. We implement PTS with the clairvoyant P-WSPT algorithm [29] and the non-clairvoyant WDEQ algorithm [8]. To compute empirical competitive ratios and error bars, we use the objective value of the 2-competitive P-WSPT algorithm as baseline. The results of the sensitivity experiment for five machines (Figure 42) show that for small noise PTS outperforms WDEQ. For growing noise the performance of PTS slowly degrades, but still improves upon WDEQ until \(\omega \approx 35\). For large values of \(\omega\), the empirical competitive ratio of PTS with \(\lambda = 0.1\) continues growing, while for \(\lambda = 0.5\) and \(\lambda = 0.8\) the ratios quickly converge to their robustness bounds.
4.3 Uncertain Task Graphs

A directed acyclic graphs (DAG) is a popular representation for parallel task processing, where each node represents the execution of a sub-task and each edge formulates precedence constraints between sub-tasks. The DAG model assumes a fixed structure capturing only straight-line code. While this captures the intra-parallelism of tasks, it does not capture the typical conditional nature of control flow instructions, such as if-then-else statements. The presence of such conditional constructs within the code modelled by the task may mean that different activations of the task cause different parts of the code to be executed. The conditional DAG model generalises the DAG model by allowing additional conditional nodes [7, 30].

Clearly, conditional nodes impose an uncertainty on the actual set of tasks and their dependencies. This increases intuitively the difficulty in answering questions on, e.g. the worst-case execution time and schedulability of a task set, at design time. We investigate how to determine the worst-case execution time (maximum makespan) for a conditional DAG task under list scheduling. We show several hardness results concerning the complexity of the optimisation problem on multiple processors, even if the conditional DAG has a well-nested structure. For general conditional DAG tasks without such structure, the problem is intractable even on a single processor. Despite these negative results, certain DAG structures, which are very common in practical applications, are very well tractable. We give an overview on the model and results; details can be found in our full version [28].

**Conditional DAG model** Let $\tau$ be a conditional parallel task (cp-task) processed on $m$ identical processors. The task $\tau$ is characterised by a conditional DAG $G = (V, E, C)$ where $V$ is a set of nodes, $E \subseteq V \times V$ is a set of directed
edges (arcs) and \( C \subseteq V \times V \) is a set of distinguished node pairs, the conditional pairs. Each node \( j \in V \) represents a sequential computation unit (sub-task, job) with an individual execution time \( p_j \). Slightly abusing notation we refer to jobs and nodes equivalently. The arcs describe the dependencies between sub-tasks as follows: if \((v_1, v_2) \in E\), then \( v_2 \) can only start processing if \( v_1 \) has completed, unless \( v_2 \) is an endpoint of a conditional pair.

A conditional pair \((c_1, c_2) \in C\) of nodes denotes the beginning and ending of a conditional construct such as an if-then-else statement. In sub-task \( c_1 \), there is a conditional expression being evaluated and, depending on the outcome, exactly one out of many possible subsequent successors must be chosen. The examples in Figures 43a and 43b show conditional DAGs with conditional nodes depicted by a square and all other nodes as circles.

Notice that Figure 43a shows nested conditional constructs where there is no interaction between a node within a conditional branch and any other node outside this branch. Assuming such well-nested structures as input is very natural when modelling the execution flow of a structured programming language [30]. Below we give additional complexity theoretic arguments supporting this assumption. In contrast, Figure 43b shows a cp-task without this assumption; we refer to it as conditional DAG with shared nodes.

**List scheduling** We consider list scheduling of conditional DAGs. Let \( \tau \) be a cp-task with \( G = (V, E, C) \) and with execution times \( p_j \), for each \( j \in V \), to be executed on \( m \) parallel identical processors. Let \( < \) be a given fixed-priority order (FP-order) over \( V \). A (non-preemptive) fixed-priority schedule (FP-schedule) starts executing the job with the highest priority according to \( < \) among the available jobs whenever a processor is idle and processes it until completion. A job is available if all predecessors have been completed.

Consider a set of jobs \( J \subseteq V \) obtained by fully executing the jobs of the conditional DAG \( G = (V, E, C) \) taking into account the outcome of conditional pairs. We call \( J \) and the induced subgraph \( G_j \) of \( G \) realisation of \( G \). In the resulting FP-schedule induced by \( < \) for the realisation \( G_j \) the latest completion time of any job is called makespan for realisation \( G_j \). Taking the maximum makespan over all realisations, we obtain \( M(G, <) \), the worst-case makespan of \( \tau \) for list scheduling according to the FP-order \( < \).

The worst-case makespan problem (CDAG-MAX) is to compute \( M(G, <) \) for a given a cp-task with a conditional DAG \( G \), execution times \( p_j \), a number \( m \) of parallel identical processors and an FP-order \( < \).
Conditional DAGs with Shared Nodes  
Firstly, we consider the general conditional DAG model that allows shared nodes, that is, interdependencies between nodes in different conditional constructs. We show by reduction from a Boolean satisfiability problem that CDAG-MAX is highly intractable. More precisely, we show that CDAG-MAX for conditional DAGs with shared nodes is strongly coNP-complete, even on a single processor.

This is in stark contrast to CDAG-MAX without shared nodes, which can be solved on a single processor in polynomial time [30]. The hardness result shows that it is crucial (from a complexity theoretic perspective) to restrict to such nested conditional DAGs and we do so in the following. Notice also that such a restriction is justified by practical relevance [30].

Conditional DAGs without Shared Nodes  
We restrict our attention to conditions DAGs with independent (nested) conditional constructs. We develop a general reduction framework that exploits a non-obvious relation between the problem of computing the maximum makespan for a conditional DAG and minimising the maximum makespan for a DAG, a very classical scheduling problem. Our reduction framework uses an intermediate problem and we show that there is an approximation preserving reduction from the intermediate problem to CDAG-MAX that also preserves some structural properties of input instances. The reduction framework allows us to show several hardness and inapproximability results for CDAG-MAX by proving corresponding results for the intermediate problem.

The intermediate problem is LS-MAX, where we are given a precedence constraint DAG $G = (V, E)$, execution times $p_j$ for each job $j \in V$, $m$ identical parallel processors and a deadline $D$. The goal is to decide whether $\bar{c}_{\text{max}} > D$, where $\bar{c}_{\text{max}}$ is the maximum makespan that can be achieved by any list scheduling (FP) order. We give an approximation preserving polynomial time reduction from LS-MAX to CDAG-MAX.

The main idea is to use the same graph for the constructed CDAG-MAX instance as in the input LS-MAX instance but replacing each original job $j$
with a conditional construct defined by a conditional pair \( c_j = (c^1_j, c^2_j) \). Each branch of the conditional construct consists of a single copy of job \( j \) (see Figure 44) such that each realisation of the constructed conditional DAG will execute exactly one copy of each original job using precedence constraints analogous to the original instance. Essentially, each realisation of the constructed conditional DAG then is a copy of the input LS-MAX graph (with conditional dummy nodes) as illustrated in Figure 45. The key ingredient is the definition of the fixed priority order of the constructed conditional DAG task such that each possible list scheduling order of the LS-MAX instance is used by at least one realisation of the constructed conditional DAG. This implies that the makespan of each FP schedule of the LS-MAX instance corresponds to the makespan of a realisation of the CDAG-MAX instance and vice versa.

![Figure 45: a) Input LS-MAX instance. b) Conditional DAG as constructed by the reduction. Each \( G_i \) represents a conditional construct as illustrated in Figure 44. c) Realisation of the constructed conditional DAG. Figure adapted from [28].](image)

Based on this reduction framework, we show several hardness and inapproximability results for CDAG-MAX by proving the corresponding results for LS-MAX. As a main result, we show that CDAG-MAX is strongly coNP-complete for both, non-preemptive (run-to-completion) and preemptive scheduling. Further, we prove that CDAG-MAX is strongly coNP-complete even if each realisation of the conditional DAG is a tree or a two-terminal series-parallel graph, and it is weakly coNP-complete if each realisation of the conditional DAG consists of four chains to be processed on two processors. Since our reduction is approximation preserving, it enables us to derive inapproximability results. We show that it is NP-hard to determine the worst-case makespan for non-preemptive and preemptive FP-scheduling of a given conditional DAG task on multiple processors within a factor 7/5 and 6/5, respectively.

On the positive side, it is already known that the worst-case makespan of a conditional DAG task can be approximated within a factor of 2 [30]. We show that if the conditional DAG (without shared nodes) has in each realisation bounded width, then the worst-case makespan for non-preemptive FP-scheduling can be computed in pseudo-polynomial time via a dynamic program. A realisation
has a bounded width, if the size of the longest antichain of the realisation is bounded by a constant. We also show that this algorithm can be turned into a fully polynomial-time approximation scheme (FPTAS, i.e. arbitrarily small approximation factor) if a certain monotonicity property holds for the job completion times under FP-scheduling. We prove that it does hold, e.g. for a bounded number of chains. For general DAGs the monotonicity property does not hold as a classical example known as Graham anomaly [14] shows.

4.4 Uncertain Processor Availability or Speed

In this section, we focus on a type of uncertainty that has received little attention in the literature: uncertainty about machine states and their characteristics. The objective is to design scheduling strategies that can be robust if some machines are not available or turn out to run at lower speeds. The unavailability of machines may come from failure or from other invasive programs having claimed their resources exclusively. The speed reduction (or generally change in speed) may come from a global decision to save energy or to avoid overheating, or can be due to run-time requirement enforcements.

Specifically, we are given a set of jobs to split into $m$ subsets (or bags) so that they can be executed on $m$ machines. But, after these bags are definitely decided, we learn which machines are available and at what speed. We now have to schedule the $m$ bags of jobs onto the machines minimising the makespan, this time fully knowing their state. The objective is to be provably close to the optimal schedule an algorithm could obtain if it knew the machine states in advance. The metric we use to measure this is the competitive ratio, i.e. the maximum ratio between the makespan achieved and the optimal makespan, over all adversarial machine states.

The core of the decision relies on the first phase of the problem, the creation of the bags. The second phase, scheduling the bags, is not restricted in complexity, following standard online algorithms practice. This means that the lower bounds we provide hold for any algorithm in the second phase, but the algorithms we design will not abuse this.

As this problem is too difficult to be solved in its full generality, we consider two variants on the allowed machine states and three variants on the allowed job sizes. Combining these options therefore leads to six variants of this problem.

- The three settings related to job sizes are named Sand, Bricks and Rocks. Sand means that jobs have an infinitesimal size. This setting is fundamental from a theoretic perspective as it allows to understand the best balance of
bags without considering packing issues of jobs into fixed-capacity bins. Bricks mean that all jobs have a unit size and this setting allows to see how the perfect balance in the Sand setting is affected by this simple discretisation. Rocks means that jobs can have an arbitrary size, and is therefore more difficult to solve as it requires not only to understand what the algorithm is able to achieve, but also what the optimal algorithm is not able to achieve, which gives more flexibility.

- With respect to machine states, we consider the setting with arbitrary speeds and the restricted variant with speeds being either 0 or 1. The latter means that some machines fail, but the remaining machines all run at the same speed; this variant has already been studied with Sand and Rocks by Stein and Zhong [36].

Table 7: Summary of lower and upper bound results on speed-robust scheduling.

<table>
<thead>
<tr>
<th>General speeds</th>
<th>Speeds from {0, 1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.B</td>
<td>U.B</td>
</tr>
<tr>
<td>Discrete jobs (Rocks)</td>
<td>$\bar{\rho}(m)$</td>
</tr>
<tr>
<td>Equal-size jobs (Bricks)</td>
<td>$\bar{\rho}(m)$</td>
</tr>
<tr>
<td>Infinitesimal jobs (Sand)</td>
<td>$\bar{\rho}(m)$</td>
</tr>
</tbody>
</table>

The current knowledge about the competitive ratio lower and upper bounds of all six settings is summarised in Table 7. We do not detail here the exact definition of $\bar{\rho}_{01}(m)$ and $\bar{\rho}(m)$. For more details we refer to our full version [12]. Previously, the results known were the lower bounds for the \{0, 1\}-speed setting and the 5/3 upper bound. We establish two tight upper bounds in this setting. For Sand, we show that the lower bound which uses only two scenarios of failing machines is surprisingly tight by exposing a configuration of bags carefully imbalanced so that roughly one third of the bags is as full as possible (i.e. not too big if no machine fails) and the remaining bag sizes decrease linearly every other bag. The bag configuration for 20 machines and how to schedule them if 6 machines fail is illustrated in Figure 46. The novelty in this configuration is that the first bag is not as big as possible, which allows to have a faster increase of the bag sizes. The previous study proposed a weaker solution with larger small bags but a slower nonlinear increase in bag sizes.

The knowledge of this optimal solution for Sand is also an important building block to achieve a 4/3-competitive algorithm for bricks which matches the
lower bound. Intuitively, if the number of bricks is large enough, then the Sand solution can be approximated sufficiently to obtain a 4/3-algorithm. The correctness of this algorithm is completed by an intricate case analysis for fewer jobs involving high-level arguments to eliminate redundant cases and a computer program to show that the remaining base cases also lead to a 4/3-competitive algorithm.

In the general speed settings, we also obtain matching upper and lower bounds for Sand, which are substantially higher than for the restricted speed setting. The arguments are entirely different as the configuration has to be designed to be robust against $m$ relevant adversarial strategies, whereas for restricted speeds, only two adversarial strategies were actually relevant for Sand. We show that a classic scheduling strategy named LPT (Largest Processing Time first) is able to achieve a competitive ratio of $2 - \frac{1}{m}$ for Rocks. While this result is straightforward with restricted speeds, it is highly nontrivial in the general case. We also show that this ratio of 2 can be improved for all $m$ on Bricks: we design a 1.8-competitive algorithm, but we are not able to match the lower bound for Bricks, contrarily to the restricted speeds.

There is still quite a lot to understand on both speed settings and rocks, as the gaps between the lower bound and the algorithms is quite large and both should be improvable: no lower bound uses different job sizes, and no algorithm analysis achieves to precisely make use of the restriction on the optimal solution when the algorithm leads to a high makespan.
4.5 Speed-Scaling for Run-Time Requirement Enforcement and Power Management

Dynamic voltage and frequency scaling (DVFS) on modern processors is a means to actively control the power and energy consumption of a multi-processor system on chip (MPSoC). In particular it can be used for run-time requirement enforcement (RRE); cf. Chapter 2. We study how DVFS can be used to minimise the energy consumption of an application running on an MPSoC under various scenarios. We bring old and new results together to provide a complete spectrum of possible solutions, and evaluate their quality and running times on a real-world dataset. Details can be found in the full version [34].

An application is modelled by a set of tasks whose nominal execution times are known and which obey precedence constraints given by a direct acyclic graph $G$ (DAG). A task cannot be started before its predecessors are terminated. Each core can execute one task at any time and its speed can be chosen within a range of accessible speeds, and changed between each task execution. We distinguish two models depending on whether we allow any speed in an interval (mainly for theoretical analysis) or if we have to choose a speed among a set of possible values (depending on the manufacturer). The actual execution time of a task is then divided by the speed of the corresponding core. The counterpart is of course that the power consumption of a core depends on its speed. We assume as usual that the energy consumed by a core running at speed $s$ for one unit of time equals $s^\alpha$ for some small constant $\alpha > 1$.

We consider two types of problems:

- **SpeedScaling**: We are given the mapping of each task to its core and the order in which each core executes the tasks mapped to it (encoded in $G$). The problem is then equivalent to minimising the critical path of the graph $G$. That is, find speeds such that the total execution time of the longest path is minimised.

- **Speeds&Scheduling**: In addition to selecting the speeds at which each task should be executed, we provide a schedule for the tasks, i.e. we determine the core and the starting time for each task.

The knowledge of some properties on the precedence graph $G$ is very useful to determine whether efficient solutions can be designed. For instance, Series-Parallel Graphs (SP-graphs) are known to admit a linear-time optimal algorithm for SpeedScaling and continuous speeds. An SP-graph, see Figure 47, is defined as either a single task, the series composition of two SP-graphs
4.5 Speed-Scaling for Run-Time Requirement Enforcement and Power Management

or the parallel composition of two SP-graphs. A parallel composition adds no precedence constraints whereas in a series composition, one graph must be entirely terminated before starting any task of the second one.

Therefore, for the SpeedScaling problem, we propose using a fast linear algorithm to solve the problem for continuous speeds on a SP-graph (although not always respecting the speed bounds, which was not an issue on our experiments), or to resort to a convex program otherwise. Regarding discrete speeds, a rounding approach allows to find efficiently a $r^{\alpha - 1}$-approximation on SP-graphs where $r$ equals the maximum ratio between two consecutive allowed speeds, and an ILP computes the solution in the general case. The algorithms are summarised in Table 8 and the computational results are presented in Figure 48. We use the E3S dataset composed of graphs containing around 10 tasks and the GENOME dataset from the Pegasus library composed of graphs ranging from 50 to 1000 tasks. All these datasets were composed exclusively of SP-graphs. Discrete speeds are chosen as 20 equally distributed options. One of the unexpected conclusions of this experiment is that, on SP-graphs and discrete speeds, the rounding approach offers a quality of solution only a few percents away from the optimal ILP in a computational time two orders of magnitude shorter.

Table 8: Summary of results for SpeedScaling (colours link to Figure 48).

<table>
<thead>
<tr>
<th></th>
<th>Approximate Solution</th>
<th>Exact Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous speeds</td>
<td>SP-graph (exact solution)</td>
<td>Convex Program</td>
</tr>
<tr>
<td>Discrete speeds</td>
<td>SP-G + rounding ($r^{\alpha - 1}$-approx)</td>
<td>ILP</td>
</tr>
</tbody>
</table>

The SpeedsScheduling problem is more complex to solve. First, exact solutions cannot be computed in a reasonable time. There is no clear formulation for continuous speeds, and the ILP for discrete speeds is prohibitively expensive. This means that approximate solutions are the best choice here. The solution we propose for continuous speeds is a $2^{\alpha - 1}$-approximation if
the optimal solution uses speeds of at most half of the maximum possible speed. This algorithm uses a convex program to decide the speeds on which each task should be run and a list scheduling algorithm to decide when and on which core they are run. A rounding procedure similar to the one used for the SPEEDSCALING problem also allows to achieve an approximation ratio for discrete speeds. The algorithms are summarised in Table 9. In Figure 49, we illustrate the running times and solution quality achieved by the latter approach on the same dataset as for the SPEEDSCALING problem. In particular, the algorithm was able to handle a graph of 100 tasks in 25ms, 500 tasks in 75ms and 1000 tasks in 140ms.

Table 9: Summary of algorithms for the SPEEDS&SCHEDULING problem.

<table>
<thead>
<tr>
<th></th>
<th>Approximate Solution</th>
<th>Exact Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous speeds</td>
<td>Convex Program + List Scheduling $(2^{\alpha-1}$-approx)</td>
<td>Prohibitive ILP</td>
</tr>
<tr>
<td>Discrete speeds</td>
<td>Convex Program + Rounding + List Scheduling ($(2r)^{\alpha-1}$-approx)</td>
<td></td>
</tr>
</tbody>
</table>
4.6 Conclusion

In this chapter, we addressed several types of uncertainty that arise in allocating and scheduling resources in heterogeneous systems using invasive programming concepts. We presented algorithmic and complexity theoretic results. In particular, we designed and verified, both theoretically and empirically, methods for scheduling problems with uncertainties in executions times of tasks, dependencies between tasks, and processor availability and speed. As a fundamental tool to enforce run-time requirements of invasive applications, we investigated methods for dynamic voltage frequency scaling with strong guarantees on running time and performance.

While we investigated these types of uncertainty independently, it remains as open question how to handle combinations thereof. Further, it would be interesting to continue the line of research going beyond worst-case guarantees, e.g. by employing techniques such as data-driven algorithm design.

Figure 49: Consumed energy of the $5 \times 5$ benchmark graphs for solutions found by the Convex Program & rounding approach (squares) subject to a fixed number of available (free) cores ranging from 1 to 128. The results are normalised: a value of 1 corresponds to the case with optimum discrete speeds and infinitely many cores. The required solver time to find these solutions ranges from 7 ms (■) to 150 ms (■) according to the given colour key. The crosses denote optimal-energy solutions as determined by the prohibitive ILP approach on small instances. Figure adapted from [34].
References


References


4 Scheduling Invasive Multicore Programs Under Uncertainty


B: Architectural Research
5 Adaptive Application-Specific Invasive Micro-Architectures

Lars Bauer, Jürgen Becker, Jörg Henkel, Fabian Lesniak, Hassan Nassar

Abstract Adaptivity is one of the key features of a computing system to improve performance. In this chapter, we present the i-Core, a runtime-reconfigurable processor. Its instruction set is extended by Special Instructions (SIs) that are executed on a runtime-reconfigurable hardware accelerator framework. By this means, applications can be processed much faster by using their specific SIs. We also enabled additional standard processors in the i-Core tile to make use of the accelerator framework to improve the overall performance even further. In addition, we discuss hardware extensions to allow more versatile use of the invasive micro-architecture ($\mu$Arch). An adaptive L1-cache allowing to assign cache resources during runtime and an interface to use the accelerator framework as a transparent near-memory accelerator are presented. We also present a security analysis with counter measures and an in-depth analysis on the worst-case execution time including the delay for reconfiguration during runtime.

5.1 Introduction

In this chapter, we develop the i-Core and investigate mechanisms that provide runtime adaptivity in its micro-architecture ($\mu$Arch). The primary focus is on advancing state-of-the-art reconfigurable processors while investigating their applicability to invasive computing. Such reconfigurable processors leverage runtime-reconfigurable hardware (i.e. FPGAs) to provide application-specific accelerators. This allows to adapt the instruction set to implement so-called Special Instructions (SIs) dynamically at runtime to gain a significant speedup during execution. Our efforts led to the creation of the i-Core, a runtime-reconfigurable processor that is binary compatible with the LEON3 processor used in the invasive computing prototype and that makes reconfigurable hardware easily usable for invasive applications.

After introducing the basic i-Core architecture and its components in the following, we will present selected topics in the remainder of this chapter. At first, we will show manually designed accelerators and SIs for applications used throughout the Invasive Computing project, and in Section 5.2 we present an
approach to generate SIs dynamically at runtime. In Section 5.3, the benefits of adaptivity are examined further in the form of an adaptive cache architecture. We developed dynamic intra-tile cache reallocation, leading to performance improvements by adapting the amount of available cache according to the needs of the application. We then analyse security aspects of adaptive caches and reconfigurable accelerators in Section 5.4. To increase the efficient utilisation of the reconfigurable hardware, we have developed an intra-tile multicore support, allowing regular cores to use the reconfigurable fabric of an i-Core (see Section 5.5), and we have also developed a near memory computing extension (see Section 5.6). In Section 5.7, we present task scheduling for runtime-reconfigurable processors that hides the reconfiguration latency by scheduling decisions. And last but not least, we also developed performance models for a save and tight worst-case execution-time (WCET) analysis of the i-Core in Section 5.8. This includes strategies to analyse and enforce WCET for hard real-time applications, while considering the impact of the reconfigurations during runtime.

5.1.1 Overview of the i-Core Architecture

Figure 50: i-Core consisting of adaptive microarchitecture (μArch) and reconfigurable fabric.

Figure 50 provides an overview of the i-Core architecture [4, 10, 16] in an i-Core tile. The LEON3 pipeline was extended with several components to enable the use of application-specific SIs that are implemented by runtime-reconfigurable accelerators which are provided by the application developer along with the application binary. The accelerators are loaded into reconfigurable containers that correspond to embedded FPGAs (eFPGA) inside the InvasIC system, i.e. fine-grained reconfigurable regions that can be recon-
figured at runtime without disrupting the rest of the system. For prototyping purposes, instead of eFPGAs inside an ASIC, we use the partial runtime reconfiguration feature of Xilinx FPGAs to obtain the same functionality. They are connected to an interconnect infrastructure that establishes communication between the accelerators, the i-Core μArch, and the on-chip memory. The main architectural components and extensions of an i-Core tile are:

**i-Core Tile-Local Memory** – Initial requirement analysis showed that most SIs are memory-bound. Thus, to utilise the reconfigurable fabric efficiently, a high-bandwidth memory connection is required. The tile-local memory (TLM) on i-Core tiles provides two additional 128-bit ports to the i-Core reconfigurable fabric in addition to the 32-bit AHB port.

**Adaptive Cache** – The L1 data cache can adaptively change architectural parameters at runtime, like the associativity and size.

**Memory Arbiter** – The memory arbiter connects the memory stage of the pipeline and the load/store units (LSUs) of the reconfigurable fabric with the adaptive cache and the TLM. It automatically manages the different bit width, e.g. if an LSU performs a 128-bit access to an address that is not in the TLM address range, then the memory arbiter will perform four 32-bit accesses to the cache instead.

**CoRQ** – We developed the Command-based Reconfiguration Queue (CoRQ) to load accelerators into the reconfigurable fabric at runtime by performing partial reconfiguration of the fabric [7]. The focus of CoRQ is to enable the CPU to issue sequences of reconfiguration requests, and relieve the CPU from managing accelerator availability. To guarantee the reconfiguration delays, CoRQ uses an internal buffer to store the accelerator bitstreams, which allows to utilise the full bandwidth of the reconfiguration port. CoRQ also allows to preempt and resume a reconfiguration, which ensures that reconfiguration requests from a high-priority task can not be delayed by reconfigurations from low-priority tasks, while at the same time ensuring progress for the low-priority reconfigurations (instead of aborting and restarting them) [23].

**SI Execution Controller** – SI execution is coordinated by the SI Execution Controller that uses SI-specific μPrograms to control the behaviour of SIs, e.g. they control memory access, communication between reconfigurable containers, etc. The μOps are stored in the remote- or primary SI storage (R-SS or P-SS), depending on whether the SI shall be executed by the i-Core or one of the other cores in the tile.
Near Memory Configuration (NMC) Interface – This module allows to configure memory regions to be used for transparent near-memory computing. It monitors memory accesses to such memory regions and triggers the reconfigurable fabric using the COREFAB interface.

COREFAB Merger and FAM – The COREFAB extension allows that not only the $i$-Core can make use of the reconfigurable fabric, but that also the other cores in the tile and the NMC can access the reconfigurable fabric. The Fabric Access Manager (FAM) grants at most one of them access to the reconfigurable fabric at any given time. And if the $i$-Core executes an SI in parallel to that, then the Merger aims at merging the $\mu$Ops coming from the R-SS and P-SS, i.e. if the two $\mu$Ops use distinct resources, then they are executed in parallel (otherwise only the $\mu$Op from the $i$-Core is executed).

$i$-Core Management Interface – This module provides access to SI-specific state information. It specifies for each SI whether the SI can be executed in hardware (otherwise an ‘unimplemented instruction’-trap will be issued) and which $\mu$Program shall be used. This information can be updated by software and by CoRQ (e.g. after a reconfiguration finishes) and it is queried by the $i$-Core pipeline whenever an SI shall be executed.

5.1.2 Prototype and Integration

We have implemented the $i$-Core with the reconfigurable fabric on both an Xilinx XUP V5 board and an Xilinx VC707 board (uses the same FPGA family as the InvasIC proFPGA prototyping platform). A floor plan of this implementation is shown in Figure 51. The blue area is the baseline LEON3 pipeline, the five red areas show the reconfigurable containers, the purple area is the reconfiguration controller, the yellow area is the on-chip memory and its interfaces, and the green area is the infrastructure to connect everything. In addition to this standalone implementation, we have integrated it together with additional standard LEON3 cores into the $i$-Core tile. A demonstrator of the $i$-Core capabilities using a video encoder demo (see Figure 52) was presented at several locations. In collaboration with the other architecture projects, the $i$-Core was integrated into the CHIPit and proFPGA demonstrators. We also provided a tool flow to obtain partial bitstreams and perform runtime reconfiguration for the invasive hardware prototype.

5.2 Generating and Using Special Instructions (SIs)

We will now discuss the reconfigurable fabric, hardware accelerators, and Special Instructions (SIs). SIs serve as the interface for the applications to
the accelerators on the fabric. An SI is an assembly instruction and multiple accelerators might be needed for its implementation. Internally, the SI uses a \( \mu \)Program to control the execution of the accelerators to implement the functionality of the SI [11]. The accelerators can be reconfigured at runtime and if not all of them are reconfigured when an \( i \)-let executes an SI, then an ‘unimplemented instruction’-trap is issued and the functionality of the SI is executed in a software implementation, i.e. the trap handler. In [16] we have provided an overview of the fabric, the SIs, as well as an application case study of an H.264 Video Encoder utilising accelerators in the fabric.

### 5.2.1 Examples for SIs and Accelerators

We have identified compute-intensive parts for the applications developed in Chapters 3, 13, and 14 that greatly benefit from execution on the \( i \)-Core. We have worked together with the compiler project (see Chapter 11) to ensure that our SIs can be used in InvadeX10 and ActorX10 \( i \)-lets. As examples we chose the SIFT feature matching (used in object recognition) and floating-point matrix multiplication (used in high-performance computing). These computationally intensive parts have been implemented as \( i \)-Core SIs (sift and fmm), for which we have developed pipelined floating-point accelerators.
We designed a pipelined and flexible floating point multiply-accumulate-vector (FMAV) accelerator [2, 21] that – among others – is used to implement sift and fmm. Figure 53 shows its block-level architecture that we integrated into the i-Core. The FMAV accelerator performs single precision floating point operations and supports 12 operation modes. This allows flexible combinations of multiplication, addition and subtraction with optional storing of the result in a register. Its resource utilisation is 1,051 LUTs (132 CLBs). Alternatively, using DSP blocks to implement FMAV can reduce the resource usage to 562 LUTs (71 CLBs) and 5 DSPs.

Figure 54(a) shows the execution times for the kernels using i-Core SIs in comparison to using the LEON3 with its commercial high-performance floating-point unit (FPU-HP), the non-pipelined FPU-Lite, and a version compiled with the soft-float library. The area overhead of the i-Core and the two LEON3s with hardware FPUs compared to a LEON3 without FPU (soft-float) is shown in Figure 54(b). The resource numbers for the i-Core include all the i-Core support hardware except the i-Core TLM, as it can be used by LEON3 cores as well. Figure 54(c) shows the efficiency of the i-Core, i.e. the achieved speedup per area (relative to the LEON3 with soft-float) in comparison to the two LEON3 hardware FPUs.

We also used the FMAV accelerator (together with other accelerators) for the fWave (54 floating-point operations) and HLLE (116 FP operations) Riemann solvers that are heavily used in the ‘Shallow Water Equations’ (SWE) application as a comprehensive example to demonstrate the benefits of invasive computing. Compared to the LEON3 with its commercial high-performance floating-point unit, the i-Core executes the fWave kernel 7.54× faster (141× compared to soft-float), which leads to a speedup of 4.82× for the entire SWE.
5.2 Generating and Using Special Instructions (SIs)

These application-specific optimisations are possible due to the large flexibility that is offered by the reconfigurable fabric. Different accelerators (e.g. for cryptographic operations, audio/video coding, image processing) can be loaded at runtime without the need to modify the i-Core.

5.2.2 Automatic Generation of SIs During Runtime

For many computationally intensive tasks, the critical section consists of one or more loops. If the processing time of such loops can be reduced by using an accelerated SI, the processing time can be greatly reduced. However, if no SIs for the i-Core are provided during compile-time (offline SI), then no acceleration is possible during runtime. Thus, we propose to automatically detect loop kernels during execution and then translate them into an equivalent SI. Several steps are necessary to accelerate a loop transparently, dynamically, and automatically: (i) monitor instructions, (ii) prepare configuration, (iii) configure hardware accelerator, and (iv) use accelerator on i-Core fabric. Such an automatically generated SI (Auto-SI) is detected and configured during the first run of the inner loop in software. Preparing a configuration for this inner loop is only possible if Atom implementations for all necessary algorithmic operations are available. Implementations for generic floating-point and vector operations are included by default and their bitstreams can be loaded in the next step. After reconfiguration, the Auto-SI is available from the 2nd run of the outer loop and the appropriate instruction is inserted dynamically.

Figure 55 shows the evaluation for an inner loop of the basic maths square root algorithm. 50,000 iterations are run on the CPU in 700,000 cycles (left). First, a realistic scenario, in which the CPU frequency is 6× of the FPGA frequency is assumed (centre). Second, the invasive LEON3 scenario is used, in which the CPU runs at twice the frequency of the FPGA fabric (right). In both scenarios, we achieve a significant speedup of 2× and 4.67×, respectively. Further analysis shows a speedup up to the factor 9.5× on sequences of ADD instructions with an additional overhead of
5.3 Run-Time Adaptive Cache Architectures

5.3.1 Intra-Tile Cache Re-Allocation

To be able to cope with the agility of dynamically adaptive systems like invasive computing systems, we introduced an adaptive cache architecture [26]. Classically, cache memory is statically allocated at design time. This leads to a distribution of memory resources that is optimised for an average use case. We propose an adaptive cache architecture that enables parametrisation and resource allocation of cache memory resources between cores during runtime. The hardware representation shown in Figure 56 makes it possible to deeply integrate the adaptive cache into an existing processor \( \mu \text{Arch} \). The reallocation is done with small overhead such that different algorithm classes can be executed more efficiently on the manycore platform [24].

The cache model is a hardware realisation that allows for dynamic runtime adaptations of various cache parameters and settings. Different design trade-offs are weighed against each other and a modular implementation is presented. This hardware representation makes it possible to deeply integrate the adaptive cache into the i-Core \( \mu \text{Arch} \). A core feature of the adaptive cache is the intra-tile cache adaptation, i.e. the available cache memory resources can be re-allocated between the cores within a tile. To reduce the cost of such a cache architecture, each core is able to acquire half of the cache resources of each neighbouring core. Thus, each core could double its cache capacity while only marginally increas-

Table 10: Relative runtime after cache re-allocation.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Relative Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 0 ADPCM (MiBench)</td>
<td>-0.5%</td>
</tr>
<tr>
<td>CPU 1 Coremark</td>
<td>+2.8%</td>
</tr>
</tbody>
</table>

22% resource consumption. The relation shown between overhead and speed-up of these first results is very promising [15].
ing the multiplexers and wires. To give a simple example demonstrating the potential of the cache re-allocation, we consider two benchmarks (ADPCM and Coremark): In the basic setup, Coremark executes with four cache blocks on \( CPU_0 \) while ADPCM is running less cache-efficient with four cache blocks on \( CPU_1 \). As shown in Table 10, we can see a relative speedup of nearly 3% including 880 ns adaptation time after reallocating two cache blocks from \( CPU_0 \) to \( CPU_1 \).

The design is carefully weighted with as little overhead as possible to maintain the performance of the cache architecture to be applicable for the strict timing constraints of an L1 cache. The L1 cache design is integrated into the CPU \( \mu \)Arch and is evaluated on reconfigurable hardware. In our evaluation we showed that already a slight hardware overhead of less than 10% compared to the standard LEON3 cache subsystem enables our dynamic cache architecture [25]. In addition, we showed that it is possible to fit the cache memory optimally onto the FPGA block RAM memory primitives.

### 5.3.2 Associativity and Cache Size Impact

In the following, detailed tests regarding cache miss rates and associativity are discussed. For the first test, the software application is run five times on previously defined hardware scenarios. Each time the application runs, a different cache configuration is applied by the software. The configuration parameter to be changed is the amount of exclusive cache blocks available as cache memory for the processor. A cache block is the smallest instance of the cache architecture consisting of multiple lines with multiple words each, i.e. a set, with a fixed size of e.g. 512 bytes. By changing the amount of cache blocks that the cache can utilise, not only the cache size will change depending on the size of each cache block, but also the cache associativity will change. For each scenario, we vary the amount of exclusive cache blocks from 4 to 8 in steps of 1. We measure the cache miss rate during the execution of the application as performance metric.

The left graph in Figure 57 shows the behaviour of the cache misses over the total cache size. We can notice a non-linear behaviour and a performance gain that can be achieved when running the benchmark with a different cache configuration. For instance, a cache with a total size of 3.5 kB (built using 7 cache blocks of 512 bytes per cache block) would lead to the lowest miss rate in this case, hence the best cache performance. In this test, we always selected the same amount of cache blocks for each hardware scenario (cache block size of 256/512/1024 bytes), which lead to different total cache sizes for each case.
5 Adaptive Application-Specific Invasive Micro-Architectures

Figure 57: Miss rate for the Coremark benchmark with 2000 iterations on three scenarios of the adaptive cache architecture with either same associativity or cache size.

For the second test (right graph in Figure 57), the same three hardware scenarios previously presented are being used. However, this time the total cache size will be varied from 2 to 6 kB in steps of 1 kB by adjusting the amount of cache blocks respectively for each hardware scenario. This has the advantage that the effect of the cache block size and amount of cache blocks can be directly seen for all scenarios for each cache size. The hardware designs have been expanded to contain more available cache blocks as needed, however, the amount of cache blocks to be used in each case is still configured at runtime by the software and varies depending on the hardware scenario. For example, to build a 5 kB cache in all three scenarios we would need 20 cache blocks of 256 bytes, 10 cache blocks of 512 bytes, but only 5 cache blocks of 1024 bytes.

The right graph in Figure 57 shows the different miss rates measured for each cache configuration on each hardware scenario. We see the miss rates staying in the range of 5% to 8% with variations in the miss rate in a range of about 2%. The data shows that for this particular application a minimum miss rate can be achieved at a total cache size of 3.5 kB with 7 cache blocks of 512 bytes per cache block. The same analysis for other applications would provide different results, thus showing the importance of an adaptive cache structure that can match the individual application needs and that can be controlled by the software.

Our cache architecture exploits fine-grain runtime adaptation, which enables performance gains while keeping the hardware implementation overhead to a minimum. It is especially suitable for multicore architectures, where a common cache resource can be distributed among multiple cores.
5.4 Security Aspects of the i-Core

This section focuses on the security analysis performed on the i-Core and its adaptive μArch. First, methods to detect information leakage within the reallocatable cache are discussed. Afterwards, we provide insight on voltage-based attacks on the reconfigurable fabric and possible countermeasures.

5.4.1 Information Leakage Protection

Our improved cache architecture provides reallocatable cache blocks that can be remapped on demand, according to the individual cache requirements of the applications on each processor. Originally, the cache architecture was designed with functional goals in mind, while not considering security concepts in the first place. Therefore, we have investigated possible information leakage between individual cache blocks. Information leakage can happen on both, the hardware and software level. Since the cache is purely implemented in hardware, we focus on methods to detect and prohibit information leakage on the hardware level.

The cache architecture offers isolation between cache blocks of individual processors, thus guaranteeing that no information stored in the cache will be visible on another processor without sharing it explicitly or through the main memory. However, vulnerability to information leakage can not be ruled out completely at design time. Possible weaknesses include untested or unexpected hardware behaviour, which may result from incorrect implementation or even degradation of the integrated circuit. To detect leakage, we propose measuring the bit rate of data written or read by each CPU to each of its

Figure 58: Cache bit-rate reference compared to the leakage case.
cache blocks. The observed bit rate at each cache block is then compared to a previously determined reference.

Figure 58 shows a comparison between the reference bit rate and the deviating measurement in case of information leakage. In this case, information is leaked by a secondary processor that has unexpected read access to cached values of the primary CPU. However, this mechanism is only feasible for data flow-centric, repetitive tasks due to the dependency on a reference bit rate measurement. Tasks with non-uniform memory access patterns can deviate from the reference even without leakage.

5.4.2 Isolation Modes and Mitigation of Voltage-based Attacks

As explained in Section 5.1.1, the application developer has the freedom of creating new SIs and accelerators. However, if the application developer is not trusted, then these new SIs can be maliciously used to harm the system or to manipulate data. Several successful attacks exist in the literature. The attacker might perform fault attacks from inside the FPGA by causing a strong voltage drop by using so-called power-wasting circuits (e.g. ring oscillators). In Figure 59, we show four power-wasting circuit types that cause high voltage drops and subsequent crashes, timing faults, or side-channel leakage [17]. The first three power wasters, Figure 59(a), (b) and (c), all exhibit a self-oscillating behaviour [18]. The fourth attack in Figure 59(d) is based on dual-port RAM [1]. It creates short circuits rather than self-oscillation, which is achieved by writing data via one port and simultaneously writing the inverse data to the same address via the other port. These attacks can lead to system crashes after only a few microseconds [20].

Hence, isolation is needed to countermeasure these attacks. The first component that we need for a countermeasure against a malicious accelerator is the fast and guaranteed reconfiguration latency of CoRQ (see Section 5.1). The idea is to use the blanking bitstream provided by the FPGA vendor tools to overwrite the suspected accelerator. However, even with CoRQ’s performance, this countermeasure would be too slow to prevent an attack. Hence there is a need to build custom bitstreams that can be reconfigured within the time limits before an attack can lead to a successful crash.
To build the custom bitstreams, the structure of the normal bitstream has to be understood. The explanation here focuses on the 7-series architecture from Xilinx, however, it also holds true for other Xilinx FPGAs. Figure 60 shows the bitstream structure of (a) the 7-series, (b) and (c) the custom bitstreams that we build. For the 7-series, the first part of the bitstream selects the reconfigurable container that shall be reconfigured on the reconfigurable fabric. This is done by writing data to the frames (i.e. the smallest addressable entity of the configuration data). This selection always requires a fixed size in the bitstream, no matter how large the container is or how the design actually looks like. The next part of the bitstream deals with shutting down the selected container, which is done by the SHUTDOWN command that disconnects the interface between the static logic and the container. That means that the logic inside the region actually continues executing until it is overwritten by the design payload, i.e. the final part of the bitstream. The design payload has a variable size, as it scales with the size of the container. This means that the blanking bitstream for the 7-series architecture has to overwrite a large portion of the container before a potential attack is stopped. This comes at long and unpredictable timing costs.

Instead, our approach ignores the payload and focuses on using the AGHIGH command. This command changes all accelerator-internal interconnects into ‘Z’ state, thus breaking the loops of all self-oscillating structures. To be able to do that in a flexible way, we separate two parts of the bitstream (select, shutdown; see Figure 60) into separate bitstreams. Hence selection, shutdown, and payload can be configured individually when needed, which allows us to precisely control the configuration status of a potentially malicious container. The shutdown bitstream is also modified to use the AGHIGH command instead of the SHUTDOWN command. The final result is that using our ‘LoopBreaker’ bitstream is \(20\times\) faster than using the blanking bitstream provided by the FPGA vendor.

Identifying the suspect accelerator is not a trivial task. If performed wrong, the system will be affected by disabling a legitimate accelerator, while the
malicious accelerator might still execute and harm the system. To perform the selection, previous works like [18] performed netlist analysis and searched for combinational loops. However, we demonstrate the attacks are not limited to the simple structures from Figure 59. For example, a Reed-Solomon encoder can easily become malicious by simply removing the registers on the path between each multiply accumulate stage. This complicates the detection problem and can bypass the state-of-the-art detection tools.

To overcome this problem, we developed our own detection tool. The detection tool is a lightweight software that can be run either as part of the software that controls CoRQ, or through a trusted remote entity to evaluate the bitstream similar to attestation. It needs to performed only once per accelerator bitstream. We first produced a dataset of a collection of bitstreams between pure attacks, attacks hidden in benign logic, and benign logic. Then using feature engineering, we extract five features from the bitstreams and one from the design metadata. The one based on the metadata is the estimated power by Vivado which can be signed and sent with the bitstream. The other features detect the repetition within the bitstream, the resource usage within the container and the randomness of the design implemented by the bitstream. Based on these features, the bitstream is classified by a machine learning approach into one of three categories. The first category is green, which is the completely harmless designs. The second category is yellow, which are suspicious designs. They might be weak attacks or high-power or resource-consuming designs that can be a catalyst for attackers in neighbouring reconfigurable containers, as we demonstrated. This category is the one that our LoopBreaker countermeasure can target at runtime when they actually start an attack. The final category is red, which are the designs that are clearly malicious and should not be allowed to be used as accelerators on the i-Core.

5.5 Intra-Tile Multicore Extension for the i-Core

As we have shown in Section 5.2, a significant speedup can be achieved by using the reconfigurable fabric of the i-Core, e.g. 10× for SIFT matching (used in the object recognition application) compared to a LEON3 with an FPU. We have observed that even when implementing complex Special Instructions (SIs) on the reconfigurable fabric, parts of the fabric may remain unused if only a single i-let executes on the i-Core. Additionally, the performance improvement was available only to i-lets running on the i-Core. Therefore, we investigated the possibility of granting other cores in an i-Core tile access to the reconfigurable fabric. Hence allowing concurrent execution of multiple SIs (issued by different cores on the tile) through the use of idle fabric resources.
As a result we proposed the COcurrent REconfigurable FABric utilisation (COREFAB) technique in [12]. COREFAB describes hardware components (see Figure 50 in Section 5.1) and a protocol to allow regular cores in a reconfigurable shared-memory multicore (which corresponds to a tile in the invasive architecture) accessing the fabric of a reconfigurable core. Additionally, it introduces the concept of on-the-fly merging of fabric accesses from different cores at runtime, to allow execution of multiple kernels concurrently.

In the following, we will name SIs issued by the i-Core as ‘primary SIs’ and those from regular cores as ‘remote SIs’. From the application developer’s view, primary SIs and remote SIs appear identical, but the hardware interface for remote SIs is slower (i.e. the latency to issue a remote SI is higher). Remote SIs use a dedicated ‘Remote SI Storage (R-SS)’ (to store their \( \mu \)Programs) that can be accessed in parallel to the ‘Primary SI Storage (P-SS)’ for the primary SIs (see Figure 50). If a primary SI and a remote SI execute in parallel, then the SI ‘Merger’ (see Figure 50) analyses in every control step whether or not the \( \mu \)Ops of the two SIs collide in their resource usage. If not, then they are merged on-the-fly to allow parallel execution. Otherwise, the remote SI is stalled until the next control step. To reduce the likelihood that both SIs collide, we developed an online binding that generates the \( \mu \)Programs of the primary/remote SIs in such a way that they use distinct computation and communication resources if possible [14].

In our evaluation, we compare COREFAB with state-of-the-art reconfigurable multicore systems. They only allow for exclusive access to the reconfigurable fabric, either by time-multiplexing it (SIs cannot execute in parallel) [5] or by spatially partitioning the fabric into private regions (an SI can never use the entire reconfigurable fabric) [27]. Compared to [5], COREFAB improves the performance of the LEON3 cores in an i-Core tile by 1.3\( \times \) on average, without reducing the performance of the i-Core itself. Ref. [27] leads to a 2% better performance of the LEON3 cores compared to COREFAB, however, that would come at a cost of more than 3\( \times \) lower performance of the i-Core.

### 5.6 Near Memory Computing Extensions

To improve the utilisation of the i-Core accelerator fabric, we focused on improving the accessibility of the i-Core, its accelerators and the tile-local memory (TLM). As a part of this proposition, the COREFAB mechanism (see Section 5.5) allows all generic processors in a tile to use the i-Core Special Instructions (SIs) remotely. To go one step further, the i-Core shall be made available as a generic near-memory accelerator operating on its TLM. The
goal is to allow processors to write data to the $i$-Core TLM in a fire-and-forget fashion, which is then picked up by an $i$-Core accelerator for further processing [19].

Compared to remote SIs, operations on data are not explicitly executed with an instruction, but rather run implicitly by memory access. Thereby, the $i$-Core can run asynchronously with respect to the issuing generic processor, allowing both to run in parallel. Additionally, the $i$-Core benefits from the advantages of near-memory computing: The dedicated connection to the $i$-Core TLM has particularly high bandwidth. By not using the main AHB bus, $i$-Core operations are not slowing down the generic cores in the tile.

### 5.6.1 Region-based Near-Memory Computing

The reconfigurable framework is primarily controlled by software, as it requires an SI to operate. Starting a task on a remote core is usually done using an $i$-let, which basically is a remote procedure call (RPC), requiring a communication channel to be set up before. To simplify and reduce communication effort between a remote processor core (client) and accelerator, the accelerator framework can now work transparently to memory access: The memory, which the accelerator operates on, is fully accessible via the bus and thus accessible to all clients. A client can define regions within that memory, consisting of a starting address, size of the region and an operation. By watching the memory bus, the accelerator can identify access to such a region and trigger the associated operation. This trigger can decide whether enough data is available, depending on the selected operation, and start executing that operation automatically. Afterwards, it is sufficient to stream input data to the respective address in memory and read output data after processing is done.

As in our case, the $i$-Core contains both a CPU core and an accelerator framework, which yields two possibilities to trigger near-memory operations:

**Interrupt-based trigger** The processor core receives an interrupt after a region received enough data. The $i$-Core starts processing in software and can use the hardware accelerator if suitable. We call this the software-assisted mode, as algorithms can be partly accelerated by hardware and run the remaining operations in software.

**Direct hardware trigger** The hardware accelerator is directly started without software intervention. As no software overhead occurs, this approach allows very low latencies. The $i$-Core processor can execute other code in parallel, as the hardware accelerator does not interfere with the pro-
5.6 Near Memory Computing Extensions

Processor resources. However, the respective operation needs to be fully implemented in hardware.

Platform Modifications  The implementation of the region-based near-memory accelerator (NMA) concept requires to extend the platform by modifying the processor itself and providing a new module called near-memory computing (NMC) Controller as shown in Figure 61.

![Figure 61: Implementation overview for the near-memory operations, using the accelerator framework and processor core of the i-Core (based on [19]).](image)

Processor modifications  Due to its ability to run a broad range of tasks the general purpose processor core is a key element to implement a very flexible NMA architecture along with the reconfigurable framework.

As a bus master, it can access not only the i-Core TLM but also other system components to load data if necessary. Therefore, clients do not have to provide all the data required to process the task if the missing values are accessible to the NMA internal processor.

To notify the processor about pending tasks in the software-assisted mode, it is directly connected to the NMC Controller. As our processor design allows customisable hardware traps, this feature is used to assert an interrupt each time the processor needs to start working on a region. Moreover, this approach allows the processor to execute unrelated tasks in parallel, which get interrupted automatically whenever the accelerator itself requires action. In our reference implementation, the operating system treats the reconfigurable
5.6.2 Near-Memory Computing (NMC) Controller

The main component of our implementation is the NMC Controller. It is placed between system bus and i-Core TLM to snoop bus transactions on the TLM. During execution, the NMC Controller can deny or delay write access to the TLM to prevent the NMC result from being affected by inconsistent values in memory. Moreover, in case of encryption algorithms it also allows to ensure that clients are not able to read the plaintext. When delaying a memory access, bus splits are used to avoid locking up the system bus. Figure 62 shows the architecture of the NMC Controller implementation.

![Figure 62: Schematic of the NMC Controller.](image)

When a write transaction to the SPM matches the processing condition of an enabled region, the corresponding action is triggered. Either an interrupt can be used in case of the software-assisted mode or the reconfigurable framework can be accessed directly using the COREFAB interface (Section 5.5). This interface allows to bypass the processor and directly execute a task in the accelerator framework. Input parameters required by the accelerator framework such as operands and the SI to run have to be configured as part of the region configuration prior to execution. Nevertheless, running special instructions (SIs) on the processor in parallel is still possible since the implementation of the extended framework comes with an arbiter to switch between local and NMC instructions.

5.6.3 Evaluation: AES Encryption Performance

To demonstrate both the flexibility and performance of the software-assisted approach, an AES encryption benchmark is used. The implementation is based on the MiBench suite. As the AES algorithm involves a diverse control
flow as well as preliminary operations like key derivation, the software-assisted operation mode is used. Consequently, a finished write to the input data region triggers an interrupt and preparations are done in software. The operations for a single encryption round are implemented in hardware. The hardware accelerator is therefore called for each encryption round (AES requires 10, 12 or 14 rounds, depending on key length) except for the last one. As a slightly different implementation is necessary in the last encryption round, we fall back to the original software implementation for that. This highlights a benefit of the processor-based approach: instead of having to implement the full AES algorithm in hardware, less demanding operations like preparation or finalisation can still be handled in software. Although only parts of the algorithm are accelerated, we still see significant performance gains as shown in Figure 63. An intermediate result is also included to distinguish memory bandwidth and hardware acceleration gains. In the intermediate case, the AES algorithm does not use hardware acceleration but still profits from increased memory performance of the NMA processor.

It can be seen that the software performance increases by 37% when using the high-bandwidth memory interface only. Using the LUT-based hardware acceleration, execution time improves by approx. 53% compared to the software implementation in conjunction with the high-bandwidth memory interface.

### 5.7 Task Scheduling for Runtime-Reconfigurable Processors

Reconfiguring accelerators requires a considerable amount of time. When an i-let enters a computational kernel, potentially executing multiple different SIs per iteration, then it will trigger reconfigurations for the accelerators of the needed SIs before the kernel starts. While reconfiguration is an asynchronous operation, and thus potentially concurrent to program execution, SIs have to be emulated in software until the required accelerators are reconfigured. This reconfiguration latency can be efficiently hidden through a task
scheduling approach if the scheduler is aware of the underlying reconfigurable architecture. We have developed the Performance Aware Task Scheduling (PATS) strategy [3] that aims to maximise efficiency of all running i-lets. We introduce the notion of efficiency for an i-let executing on the i-Core as \( \frac{\text{# completed reconfiguration requests}}{\text{# reconfiguration requests}} \). Note that an i-let can request multiple reconfigurations, depending on the SIs that it will use, but some of the accelerators may already be reconfigured due to previous requests. An i-let that has just started reconfiguring its accelerators will have a low efficiency, while an i-let that finished reconfigurations will have a high efficiency. As a special case, i-lets that do not demand any reconfigurations are defined to always have an efficiency of '1'.

PATS targets a set of periodic soft-deadline tasks and aims to reduce the system tardiness, i.e. the sum of the total time by which deadlines were missed over all i-lets. The priority to schedule an i-let depends on its efficiency, weighted by the remaining time to its soft deadline. Tasks with a low efficiency and sufficient slack wait until their efficiency increases or their slack becomes tight. Figure 64 shows the comparison of PATS with typical scheduling algorithms that do not consider the changing efficiency of i-lets. We performed the evaluation for different types of deadlines that differ in how tight they are relative to the minimum required time to finish the i-let. Overall, PATS achieves a lower tardiness compared to them.

We have also developed a Makespan Optimizer for Reconfigurable Processors (MORP). In addition to the scheduling decision, it also decides which i-lets shall obtain which share of the reconfigurable fabric at which point in time [13].

5.8 Worst Case Execution Time Analysis and Optimisations for Reconfigurable Processors

To make the remarkable average-case performance improvements of the i-Core accessible to i-lets with real-time requirements, a worst case execution time
(WCET) analysis is required for reconfigurable processors. The main challenge in comparison with non-reconfigurable processors is to analyse the reconfiguration process. A simple technique to perform it is to *stall* the i-Core during reconfigurations. But this would affect the performance significantly, as the reconfiguration time of an SI is in the range of milliseconds. Instead, reconfigurable processors typically continue execution in parallel to an ongoing reconfiguration and use a conditional branch or a trap mechanism to execute an SI in software, if its hardware is not reconfigured yet. While this provides very good execution times, it is hard to guarantee an upper bound for the WCET, because it is no longer trivially clear which SI executions can use fast hardware implementations and which are emulated in software.

Figure 65: Timing analysis of a kernel with parallel reconfiguration of an SI (based on [8]).

Figure 65 shows the iterations of an application kernel. The reconfigurations were started right before the kernel and due to our analysable CoRQ reconfiguration controller [7] (see Section 5.1) and compile-time known bitstream sizes, we know when all reconfigurations are completed. A safe WCET can be obtained by assuming that every SI invocation in every kernel iteration will be executed in software (see Figure 65 (a)), but that will lead to a high execution-time overestimation. Knowing the WCET of a single kernel iteration, we can determine how many iterations were performed *at least* until all reconfigurations are completed (see Figure 65 (b)). But it is not safe to assume that all subsequent iterations execute their SIs in hardware, as done in Figure 65 (b). If the first iterations execute *faster* than their worst case, then it can happen that more iterations use slow software execution for their SIs (see Figure 65 (c)), which leads to an *increased* execution time for the entire loop. We were the first to identified this *timing anomaly* for reconfigurable processors and developed models to precisely bound it to determine a safe WCET [8].

**WCET Enforcement for Opportunistic Runtime Reconfiguration:** In addition to the WCET analysis for the i-Core, we have developed a method to automatically choose WCET-optimising sets of SIs for computational ker-
nels [6]. Our approach selects subsets from a bigger set of possible SI implementations for a constrained reconfigurable area, with the aim of optimising the task’s WCET bound. However, this so-called WCET configuration (WCET-Cfg) has a noticeably reduced average-case execution time (ACET) compared to a configuration that optimises for the average case (ACET-Cfg, which—on the other side—has a significantly higher WCET). We now present how to combine the best of both, i.e. we provide the improved performance of the ACET-Cfg, while still being able to guarantee the minimised WCET of the WCET-Cfg.

The main idea is to utilise the slack, i.e. when an application executes, then it will often execute faster than the guaranteed WCET. The reason is that the guaranteed WCET is an upper bound of the generally unknown actual WCET. We decompose the guaranteed WCET of a job (e.g. encoding a video frame) into the WCET for one iteration of its outer-most loop, e.g. encoding a single macro-block of the frame. After each iteration of the kernel (i.e. the outer-most loop), we determine how much slack we accumulated so far by using a performance counter. We start executing with the WCET-Cfg, but after some time we typically have accumulated enough slack to reconfigure to the ACET-Cfg. During reconfiguration, the accumulated slack will reduce drastically, but afterwards the ACET-Cfg will most likely increase the slack faster than the WCET-Cfg did. However, it could happen, that the ACET-Cfg executes slower than the WCET-Cfg, as it has a higher WCET. Eventually, it could even happen that we violate the WCET that we guaranteed for encoding the video frame. To ensure that this never happens, we enforce the WCET, by reconfiguring back to the WCET-Cfg early enough to guarantee that the WCET can never be violated. Therefore, we need to determine the amount of slack that we need to accumulate before safely switching to the ACET-Cfg, such that we have enough slack left to reconfigure back to the WCET-Cfg if we need to.

To evaluate our approach, we benchmarked the main kernel of an H.264 video encoder (the EncodeMacroBlock kernel). It executes SIs to encode macro-blocks with inter-frame prediction (I-MBs) or intra-frame prediction (P-MBs). The SIs used to accelerate I-MBs and P-MBs differ partially and thus the WCET- and ACET-Cfg differ as well. The ACET-Cfg focuses more on those SIs for I-MBs, as they dominate the performance for most videos (based on profiling), whereas the WCET-Cfg must not use any profiling information but has to consider the worst case.

Figure 66 shows execution time results of the EncodeMacroBlock kernel when our approach is applied for different I-MB/P-MB ratios. A kernel iteration
that encodes a P-MB takes up to 3744 cycles longer when using the ACET-Cfg compared to the WCET-Cfg. The reconfiguration bandwidth was assumed to be 800 MB/s (as supported by Xilinx UltraScale+ FPGAs) and switching between WCET- and ACET-Cfg takes 45658 cycles. Our approach is beneficial for frames that contain at least 50% I-MBs (i.e. at least moderate motion in the video). The maximum execution time reduction is 23.0%. For frames that contain less than 50% I-MBs, the execution time can be slowed down (as expected) by up to 11%. The reason is that such scenarios were rare and thus the ACET-Cfg did not optimise for them. But even if they happened, the statically guaranteed WCET bounds were never violated and actually it was never required to reconfiguring back to the WCET-Cfg. When considering the typical execution profiles with at least 40% I-MBs, then the average execution time reduction is 10.2% compared to continuously executing the WCET-Cfg. Note that these execution time reductions were achieved on top of an already highly optimised system. Just using the i-Core accelerators already reduces the guaranteeable WCET by more than $10\times$ compared to executing software only. On top of that, our WCET-optimised configuration reduces the WCET by more than 29%. And on top of that, our opportunistic runtime reconfiguration reduces the average-case execution time by 10.2% while still ensuring the guaranteed optimised WCET (by enforcement) and by negligible overheads (adding slack monitoring using a single performance counter). More details about this work are available in [9].

5.9 Conclusion

This chapter summarised the contributions regarding adaptive microarchitectures in invasive computing systems. The main contribution in this field is the i-Core, an adaptive, runtime-reconfigurable processor. We showed the

Figure 66: Optimised execution time of EncodeMacroBlock for different execution profiles (based on [9]).
benefits of reconfigurable processors in various domains, where the use of SIs greatly improved the processing time for different algorithms. In addition, we conducted detailed analysis regarding applicability to real-time environments and with respect to the security aspects of the i-Core. We also presented improvements to the hardware surrounding the i-Core, such as the near-memory extensions and COREFAB which make it easier for other processors to use the accelerator framework. By introducing intra-tile cache reallocation, the hardware can be adapted to the cache requirements of different applications. Overall, the improvements shown in this chapter prove the benefits of adaptive application-specific hardware.

We would like to acknowledge the valuable contributions of Marvin Damschen, Artjom Grudnitsky, Tanja Harbaum, Michael Hübner and Carsten Tradowsky.

References


5 Adaptive Application-Specific Invasive Micro-Architectures


6 Invasive Tightly-Coupled Processor Arrays

Jürgen Teich, Marcel Brand, Frank Hannig, Christian Heidorn, Dominik Walter, Michael Witterauf

Abstract Tightly-Coupled Processor Arrays (TCPAs) denote a class of massively parallel arrays of locally interconnected processing elements (PEs). They distinguish themselves from coarse-grain reconfigurable arrays (CGRAs) by the fact that the PEs are programmable using small instruction memories. In consequence, TCPAs allow for the parallel execution of multiple rather than just innermost loop computations of many computationally intensive applications that may be expressed by nested loops from diverse areas of applications such as scientific computing and image and signal processing. Moreover, in distinction to GPUs, the local communication of data between PEs leads to a highly energy-efficient and, at the same time, timing-predictable acceleration of loop nests. In this chapter, invasive TCPAs are introduced. Through local propagation of binary invade request signals, complete regions including linear and 2D sub-arrays of PEs can be claimed in just a few clock cycles. Moreover, safety concepts such as dual (TMR) and triple modular redundancy (TMR) are shown to be portable to invasive parallel TCPAs by invading dual and triply-replicated regions on demand and thus enabling what we call safe loop processing. Newest research results include techniques for run-time enforcement of non-functional execution qualities for loop programs executed on invasive TCPAs. In order to enforce a given set of non-functional requirements of a loop nest, resources (the invaded TCPA region) shall not be overprovisioned. As a solution, techniques summarised as self-invasion of claim sizes of latency-bound programs, self-power adjustment and self-selection of redundancy scheme are presented. Finally, as TCPAs execute a loop application according to a cycle-accurate synchronous loop schedule, any stall of the processor array due to lack of input data or lack of buffer space to store output data should be avoided. As a solution, techniques for real-time scheduling of I/O data transfers from external sources such as DRAMs into a TCPA and back to memory are presented.
Figure 67: Example of an invasive TCPA (right) and the claims of two concurrently executed loop nests having invaded a claim of a $3 \times 3$ (orange), and a $2 \times 5$ sub-array (green), respectively. The TCPA is embedded into a heterogeneous MPSoC architecture using a network-on-chip (left). The abbreviations AG, GC, and NA stand for address generator, global controller, and network adapter, respectively. Figure adapted from [13].

### 6.1 Introduction

TCPAs [13] have been shown to provide a highly energy-efficient and, at the same time, timing-predictable acceleration for many computationally intensive applications that may be expressed by nested loops from diverse areas of applications such as scientific computing and image and signal processing [11, 26, 27]. An example of a TCPA is shown in Figure 67.

In invasive TCPAs, regions of processing elements (PEs) can be claimed based on a hardware-controlled propagation of invasion control signals between neighbouring PEs. Not only may such decentralised parallel invasion strategies reduce the invasion overhead by orders of magnitude w.r.t. a centralised approach, even bounds on the invasion time of invading $N$ processing elements in $O(N)$ clock cycles have been shown to be achievable through full invasion control in hardware in our initial research [17]. In Figure 67 (right), two concurrently executed loop nest applications are shown and their invaded sub-array claims are highlighted in different colours. Moreover, the invasive TCPA is embedded into a heterogeneous multi-tile MPSoC architecture with NoC-based inter-tile communication.

In the context of predictable multicore computing [25, 26], we have shown in [27] that the execution time (latency) and throughput of an invasive loop
program on an invasive TCPA can be determined and optimised statically. Schedules can be described by parametric expressions depending only on the problem size and claimed array size as run-time parameters (thanks to recent results on *symbolic loop scheduling*, see Chapter 11). Major architectural innovations to support guarantees for multiple non-functional properties such as *fault tolerance* and *energy consumption* have driven our research next. Here, in order to provide fault tolerance as a requirement of parallel execution, we were able to lift concepts of dual (DMR) and triple modular redundancy (TMR) known for the single processor case to the level of *replicated array processing* [23]. In this realm, a programmer can either invade a single (no protection), double (DMR; enabling error detection capabilities), or triple array replicated processing (TMR; single error correctability). In order to support also the required voting efficiently, special hardware structures are introduced that allow to switch on and off a replication scheme based on the *requirements* of an application or upon observation of a certain failure behaviour, see Section 6.2.

Code efficiency (density) is another important goal driven by tight memory constraints within processing elements of a TCPA and also affects the time needed to infect a claimed array with proper instruction sequences in parallel. In this area, a processing element architecture called *orthogonal instruction processor (OIP)* [3, 4] is introduced in Section 6.3. By providing an individual instruction memory and decoder for each functional unit inside a processing element, the problem of code explosion as is known for VLIW processors can be greatly avoided, resulting in much higher code density, energy efficiency, and in consequence, also much lower configuration times for loop applications on TCPAs.

Next, the problem of run-time enforcement of non-functional execution qualities for parallel loop programs when executed on TCPAs is investigated. In order to enforce a given set of non-functional requirements of a loop nest when executed in parallel on an invasive TCPA, an overprovisioning of resources (the invaded region of TCPA processors) shall be avoided. Here, novel techniques summarised as *self-invasion* of claim sizes of latency-bound programs, *self-power adjustment* and *self-selection of redundancy scheme* (see Section 6.4) are introduced.

Subsequently, TCPA architectures with floating-point processing elements allowing to dynamically choose/adapt the accuracy of mantissa computations per instruction, so-called *anytime instruction processors* (AIP, see Section 6.5), are described.
Finally, techniques developed for real-time scheduling of I/O data transfers from external sources such as DRAMs into a TCPA and back to memory are presented in Section 6.6.

6.2 Safe(r) Loops – Fault-tolerant Parallel Loop Processing

The high integration density of future multicore systems will inevitably lead to an increased vulnerability of the circuits, e.g. a malfunction due to thermal effects, circuitry wear-outs, or cosmic radiation. However, instead of analysing error and fault effects on single cores, for the first time well-known fault tolerance schemes such as dual (DMR) and triple modular redundancy (TMR) were lifted to the level of loop programs and their parallel processing on multicores. In particular, approaches were investigated how, based on application requirements on reliable execution, an invasive loop program may request to switch on and off fault tolerance schemes for error detection or correction of certain parts or a parallel loop application as a whole [31]. Without creating any inefficiency by including error detection circuits in our hardware, the regular structure of TCPAs offers an application to claim (a) a non-redundant, (b) a dual-replicated (see Figure 68(a)), or even (c) a triple-replicated array instance for computing the parallel program in lock-step mode (see Figure 68(b)–(d)).

In [19], this idea and how to adaptively provide (a) signal replication of input and output signal streams and (b) voting (hard-wired vs. dedicated functional units) are demonstrated. Kindly consult Chapter 11 for concepts for the compilation of loop nests to TCPAs. Figure 68 illustrates the essential ideas of our approach: Rather than claiming a single array of processors, we claim double or triple times the number of processors in a contiguous region to allow for the detection or correction, respectively, of soft errors, e.g. single event upsets (SEUs) automatically. Our proposed approach for providing on-demand DMR or TMR on such processor arrays works as follows: Using the principles of hardware/software co-design [24], a safety-critical loop program—or individual variables therein—such as found in digital media, linear algebra and signal processing applications, is transformed first so to execute a lock-step parallel schedule of two (for DMR) or three (for TMR) identical copies of a parallel loop [23]. Based on a compile-time quantitative analysis of the execution time and reliability gains in terms of failure probability as a function of the observed soft error rate during operation, it is shown in [20] how a suitable redundancy scheme may be selected at runtime to enforce a desired safety level.
6.2 Safe(r) Loops – Fault-tolerant Parallel Loop Processing

In order to realize such a capability, a so-called error handling unit (EHU) was designed and integrated within each PE that is capable of making majority votes (or comparison) on values within PE’s register file (see Figure 69(a)). An EHU generates three outputs: the result of a vote/comparison operation, a one-bit error signal denoting whether a non-maskable error is detected, and an error diagnosis signal that describes the source of the detected non-maskable error, e.g. multiple errors in different input replicas or an error in the EHU, etc. As shown in Figure 69(b), the error and diagnosis outputs from different PEs are aggregated at the level of the whole processor array to notify the on-tile control CPU (RISC processor) about any error event. Apart from this event, an error location index denoting the PE that has detected a non-maskable error, and an error diagnosis vector per PE row, describing the cause of the detected error, is generated. In case of a detected error, an interrupt is generated that leads to executing an error handling service routine as part of the TCPA driver code running on an on-tile RISC CPU. The error code (type and location of the detected error) is then communicated to the application level as a return value of the infect request, which provides the programmer with the means to specify application-specific exception handling routines, e.g. to adapt the employed fault tolerance mechanism depending on the error code.
Figure 69: Implementation of comparison/voting instructions for DMR/TMR loop execution on TCPAs. Apart from a multi-cycle software implementation, a dedicated functional unit called error handling unit (EHU) is shown (left). (a) An EHU may be integrated into each processing element as an additional functional unit. It votes (or compares) the content of multiple registers (input/output ports or internal registers of the PE) and stores the result on its output port. Upon the detection of a non-maskable error, this unit generates an error signal as well as an error diagnosis value that describes the cause and location (coordinate) of the PE where the error has occurred. The corresponding instruction is implemented by a single cycle operation. The figure in (b) shows how these signals are aggregated at the PE array level, constructing error and diagnosis vectors per PE row. In the case of detecting an error, an interrupt is raised that results in running an error handling service routine (part of the TCPA driver code) on a tile-local CPU (RISC processor).

6.3 Orthogonal instruction processing

In general, system-on-chip architectures such as TCPAs have only access to a limited amount of memory, which stands in contrast to the characteristic of VLIW compilers that typically produce lengthy code, especially when employing software pipelining in loop programs. In [3], a new processor architecture concept called Orthogonal Instruction Processing (OIP) was proposed that tackles this problem. Its principle is shown in Figure 70. Different to a conventional VLIW processor that loads each cycle a very long instruction word from a central instruction memory, the major idea of OIP is to assign each FU its own instruction memory and branch unit. However, the register files and the flag distribution are still shared among all FUs. According to [4], a considerable amount of instruction memory can be saved for many compute-intensive loop applications when migrating from a VLIW architecture to an
OIP architecture, and these savings easily outweigh a potential small hardware overhead for individual instruction pointers.

6.4 Run-Time Requirement Enforcement (RRE) on TCPAs

Run-time requirement enforcement (RRE, see Chapter 2) denotes a class of techniques to steer a set of non-functional properties on program executions, such as the latency $L$ or power $P$, within given corridors, also known as requirements. Because of their inherent uncertainty—after all, their loop bounds and the number of available PEs are unknown at compile time, only becoming known during runtime—loops pose a unique challenge for implementing RRE, a challenge we set out to conquer in Chapter 11. At the same time, this uncertainty offers unique possibilities for optimisation, e.g. by autonomously choosing the region of PEs to be claimed such that energy consumption is minimised, instead of requiring the application developer to specify the exact claim size. We call this concept self-invasion for latency-constrained programs.

In more detail: With self-invasion, the claim size of an invasion shall be determined at runtime, such as to neither overprovision nor underutilise TCPA resources while satisfying given requirements such as fulfilling latency, throughout and/or power constraints. On the first view, this problem seems

Figure 70: Processing element (PE) with an orthogonal instruction processing (OIP) architecture including three functional units (FU). Each FU has a dedicated instruction memory, an instruction decoder, a program counter, and a branch unit. The FUs share input and output interfaces as well as the register file. Additionally, each FU can access the flags of all FUs. The figure depicts the data and control flows inside a PE by blue and red connections, respectively. The flag logic is depicted by yellow connections. Figure adapted from [3].
to have ties to the challenge to statically determine latency-optimal loop schedules as presented in Chapter 11. Yet now, we do not want the application programmer to specify resource requirements, but rather specify latency bounds corridors and thus shift the determination of a proper claim size to satisfy these latency bounds to runtime. This is necessary, particularly if there is a lot of variance in the problem size of the given loop programs, e.g. unknown loop bounds.

However, as described in Chapter 11, determining such a claim size is a multi-objective optimisation problem whose execution time needs to be considered during the enforcement itself. It is, therefore, useful to compare various strategies. In the following, we assume the goal is to find a minimal claim size that satisfies a set of given requirements such as on latency and power consumption by proposing and comparing three different search strategies. Each strategy iterates over all available claim sizes \( x \times y \) (with \( X \times Y \) being the maximum available claim size), but in a different order: ascending column (\( x \) is iterated in ascending order over first, then \( y \)), ascending size (the claim sizes are ordered by \( x \times y \) in ascending order), and a pseudo-binary search starting from \([x/2], [y/2] \). These search strategies are summarised in Table 11.

When only comparing their worst-cast time complexity, the pseudo-binary search seems to be the clear winner. However, since \( X \times Y \) is usually relatively small (below 1000), the difference in the execution times is not as clear-cut,

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Ascending Column</th>
<th>Ascending Size</th>
<th>Binary</th>
</tr>
</thead>
</table>
| Pseudo code | \( \text{for } 1 \leq x \leq X \text{ do} \)
| | \( \text{for } 1 \leq y \leq Y \text{ do} \)
| | ... |
| | \( \text{while } (x_l \leq x_h \land y_l \leq y_h) \text{ do} \)
| | ... |
| | \( x_l = \left\lfloor \frac{(x_h + x_l)}{2} \right\rfloor + 1 \text{ or} \)
| | \( x_h = \left\lfloor \frac{(x_h + x_l)}{2} \right\rfloor - 1 \text{ or} \)
| | \( y_l = \left\lfloor \frac{(y_h + y_l)}{2} \right\rfloor + 1 \text{ or} \)
| | \( y_h = \left\lfloor \frac{(y_h + y_l)}{2} \right\rfloor - 1 \) |

| Search order | (1,1);(1,2);(1,3)... |
| | (1,1);(1,2);(2,1); |
| | (1,3);(3,1);(1,4);... |
| | (2,4) |
| | (2,6) |
| | (4,4) |
| | (6,2) |
| | (6,4) |
| | (6,6) |

<table>
<thead>
<tr>
<th>Performance (worst-case)</th>
<th>( O(XY) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( O(X^2Y) )</td>
</tr>
<tr>
<td></td>
<td>( O(\log(XY)) )</td>
</tr>
</tbody>
</table>
6.4 Run-Time Requirement Enforcement (RRE) on TCPAs

Table 12: Execution time for finding a minimal TCPA claim size $x \times y$ that satisfies a given latency requirement of a given 3-nested loop program.

<table>
<thead>
<tr>
<th>Loop Bounds</th>
<th>Latency Req.</th>
<th>Ascending Column</th>
<th>Ascending Size</th>
<th>Pseudo-Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>${10, 10, 10}$</td>
<td>$[170; \ 200]$</td>
<td>6.76 ms (12)</td>
<td>6.86 ms (18)</td>
<td>6.64 ms (2)</td>
</tr>
<tr>
<td>${100, 100, 100}$</td>
<td>$[170; \ 2 \cdot 10^4]$</td>
<td>7.47 ms (64)</td>
<td>7.41 ms (64)</td>
<td>6.69 ms (7)</td>
</tr>
<tr>
<td>${1000, 1000, 1000}$</td>
<td>$[170; \ 2 \cdot 10^7]$</td>
<td>7.42 ms (56)</td>
<td>7.44 ms (62)</td>
<td>6.71 ms (6)</td>
</tr>
</tbody>
</table>

as shown in Table 12. The table shows the execution times for finding the minimal claim size for a given latency requirement and loop bounds for a 3-nested loop with different bounds on a dual-core ARM Cortex-A9 with 667 MHz (the number of considered claim sizes before finding the minimum is annotated in parentheses). While the pseudo-binary search is the fastest, the difference to the other two strategies is at most only about 10%.

Therefore, a more important metric is the following: How long does finding the claim size (which we call “enforcement” in the following) take in relation to the execution time of the loop itself? This relation is highlighted in Table 13 (assuming an FPGA-based TCPA prototype with an emulation clock frequency of just 50 MHz). As can be seen, for lower input sizes (lower bounds), the enforcement decision may take more time than the loop execution itself (2000× as long). However, for larger input sizes (higher bounds), the enforcement takes much less time than the loop execution itself (enforcement is 50× faster). Still, for ASIC clock frequencies of 0.5–1 GHz for the TCPA, the overhead for claim optimisation cannot be neglected, even for larger loop bounds.

In summary, the time needed for self-invasion needs to be taken into account during runtime enforcement of execution-time related requirements on TCPAs. The required execution time to derive a minimal claim size and invade the resulting sub-region on a TCPA must be balanced with the execution time of

Table 13: Relating the execution time of enforcement with the execution time of the given loop nest itself on an FPGA-based TCPA accelerator prototype running at a clock frequency of 50 MHz. (The average of the execution times of the three search strategies was taken as the execution time of enforcement.)

<table>
<thead>
<tr>
<th>Loop Bounds</th>
<th>Execution time (TCPA)</th>
<th>Execution time (enforcement)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>${10, 10, 10}$</td>
<td>0.0034 ms</td>
<td>6.8 ms</td>
<td>2000</td>
</tr>
<tr>
<td>${100, 100, 100}$</td>
<td>0.36 ms</td>
<td>7.2 ms</td>
<td>20</td>
</tr>
<tr>
<td>${1000, 1000, 1000}$</td>
<td>360 ms</td>
<td>7.2 ms</td>
<td>0.02</td>
</tr>
</tbody>
</table>
the loop nest itself. For adequately large loop bounds or repetitive executions of a loop nest of equal loop bounds on a TCPA accelerator, this enforcement overhead, however, may easily pay off [30]. Finally, alternatively or in combination with the above runtime computation of a required minimal claim size to be invaded for satisfying a latency or execution time requirement of a loop nest, also concepts for auto-adjustment of the power to stay in a desired power corridor or to minimise the needed power have been elaborated, e.g. by adjusting the voltage/frequency of a claimed processor region at runtime.

6.5 Anytime Instruction Processors (AIP)

In many computationally intensive applications, floating-point arithmetic is favoured over fixed-point arithmetic because of its high range, coming at the cost of high latency and power consumption for arithmetic operations. However, especially in domains like digital signal or image processing, accurate results are not always necessary and small deviations can be tolerated. This fact is leveraged by approaches such as approximate computing [9, 12, 22, 32] and mixed-precision computing [2, 8, 21] to trade accuracy off against other non-functional execution properties like power and latency.

However, different applications may show different degrees of tolerance to inaccurate results. For floating-point operations, the concept of anytime instructions and related processors called anytime instruction processors (AIPs) were introduced in [5]. In such an instruction, the number of most-significant mantissa bits to be accurately computed is encoded in the instruction word itself. Using anytime instructions, execution time and energy consumption trade-offs can be realised in an application-specific way. Either the programmer or the compiler may adjust the accuracy of computations according to non-functional requirements on latency and/or energy.

In the following, the basic concept of anytime instructions is presented more closely. Then, implementations of anytime floating-point additions and multiplications are presented. Contrary to divisions for which proposed algorithms naturally operate most-significant bit first [6], novel algorithms and implementations were needed to allow most-significant bit first computations also for additions and multiplications. One approach proposed in [5] is based on on-line arithmetic, a technique to transform least significant bit first (LSB) operations into most significant bit first (MSB) (Section 6.5.1). In Section 6.5.2, an alternative approach called bitmasking is proposed.

As motivated above, anytime instructions provide a programmable accuracy for floating-point operations by specifying a number $a \in \mathbb{N}$ of most significant
mantissa result bits to be accurately computed within the instruction word as follows: \(^1\) \(\text{op} \ a \ \text{target operand1 operand2}\)

**Example 6.1.** Given the anytime instruction `add 3 r2 r1 r0` with two register operands \(r_1\) and \(r_0\) and a target register \(r_2\), and let \(r_1\) and \(r_0\) carry the values 5.25 and 2.5, respectively. The field \(a = 3\) of the above instruction indicates that only the three most significant bits (MSBs) of the result’s mantissa shall be accurately computed, whereas the exponent and the sign are always to be computed accurately to minimise the overall error. Inside the registers \(r_0\) and \(r_1\), values are represented as normalised floating-point numbers (IEEE 754 standard) each, with a sign bit \(s\), \(E\) exponent bits \(e_i\), \(0 \leq i < E\), and \(M + 1\) mantissa bits \(m_j\), where \(0 \leq j \leq M\) and \(m_M = 1\) being implicit and the only bit with value 1 before the decimal point. The bit representation of a normalised floating-point number is:

\[
s \ e_{E-1} \ldots e_0 \ (m_M \ldots m_0)\tag{20}
\]

For the simplicity of this introductory example, let \(M = 4\) and \(E = 3\) with a bias \(B\) of \(B = 3\). With \(5.25 = -1^0 \cdot 2^{5-3} \cdot 1.3125 \equiv 0\ 101 \ (1.)0101\) and \(2.5 = -1^0 \cdot 2^{4-3} \cdot 1.25 \equiv 0\ 100 \ (1.)0100\), the bit representations of the two floating point numbers in \(r_1\) and \(r_0\) are obtained as:

\[
\begin{align*}
r_1 &= 0 101 \ (1.)0101, \\
r_0 &= 0 100 \ (1.)0100.
\end{align*}
\]

Then, the addition of the two mantissae is performed after equalising the two exponents, which is achieved in this example by shifting the mantissa of the addend in \(r_0\) one position to the right. Below, the correct result mantissa (school method) and the result of the given anytime instruction is shown.

\[
\begin{array}{c}
1.0101 \\
+ \ 0.1010 \\
\hline
1.1111 \equiv 1.9375
\end{array}
\]

The result of the anytime instruction obviously deviates from the accurate result by \(~10\%)\, but we show next that latency and/or energy trade-offs can be nicely exploited thereby.

\(^1\) Note that anytime instructions work on standard precision number representations, i.e. IEEE standard floating-point formats (IEEE 754 [34]), thus not changing the precision, respectively number format.
6 Invasive Tightly-Coupled Processor Arrays

6.5.1 On-Line Arithmetic Approach

In on-line arithmetic [10], a recurrence equation is derived for an operation in which the dependencies between two consecutive result bits are removed (e.g. the carry chain of an addition) while performing the operation in a redundant number format. Without the dependencies, each result bit can then be computed independently of all others, enabling MSB first computation. Using on-line arithmetic, anytime instructions can perform MSB first addition, respectively multiplication on the mantissae of a floating-point number.

Note that on-line operations require $I = \delta + w$ steps to produce a result for an operation with two operands, where $\delta$ is the so-called on-line delay and $w$ is the width of the operands. Figure 71 depicts a sequential on-line adder that computes an addition most-significant bit first. Operating in a redundant number format reduces the complexity of the recurrence equation and the number of iterations required. Thus, binary operations in on-line arithmetic are performed in the redundant Signed Digit Radix-2 ($SDR2$) number format which defines an $n$-digit number $x \in SDR2$ as $x = \Sigma_{i=0}^{n-1} (x_i \cdot 2^i)$, $x_i \in \{-1, 0, 1\}$. Implementations, therefore, require the representation of a number $x \in SDR2$ by two $n$-bit vectors $x_{pos}$ and $x_{neg}$ as seen in Figure 71. By unrolling the depicted circuit, it is possible to omit the flip-flops (FF) and compute the addition of

![Figure 71: Sequential on-line adder. Shown is a circuit for calculating one iteration of an on-line addition, where FF denotes a flip-flop and FA a full adder. For two $n$-digit numbers $x, y \in SDR2$, $n + 2$ clock cycles are necessary to compute the $sum \in SDR2$. The computation starts by adding the most-significant digits of $x$ and $y$. Figure adapted from [5].]
all result bits entirely in parallel. Therefore, on-line arithmetic also provides a high potential for pipelining: consecutive on-line operations do not have to wait for the preceding operation to finish, but they can start as soon as the first digit of the preceding operation is computed [33]. A binary representation of the result can finally be obtained by computing \( \text{sum} = \text{sum}_{\text{pos}} - \text{sum}_{\text{neg}} \). Hence, if the next use of the computed result is not another on-line operation, a re-conversion from Signed Digit Radix-2 (SDR2) format to normal binary format is necessary. A compiler would avoid any re-conversions as much as possible, e.g. in adder trees or multiply-accumulate chains.

An on-line anytime instruction computes a result in floating-point format with value

\[
    r_{\text{on-line}} = (-1)^{s_{\text{result}}} \cdot 2^{E_{\text{result}}} \cdot \left( \sum_{k=M+1-a}^{M} (m_{k,\text{result}}) \cdot 2^k \right)
\]  

(21)

in \( a \in \mathbb{N}, a \leq w \) steps. Independent of the performed anytime instruction, the sign, exponent, and the first \( a \) MSBs (bit \( M+1-a \) up to bit \( M \)) of the mantissa are computed accurately. All other bits are not computed and are set to zero.

### 6.5.2 Bitmasking Approach

An alternative approach we investigated is called bitmasking. For additions, the idea is to mask (set to zero) the \( M+1-a \) least significant mantissa bits of the two addends of the anytime instruction. For multiplication, it was proposed to apply the mask not to the operands but to the partial products that are summed up. Compared to on-line arithmetic, bitmasking may cause higher error margins, but no hardware overhead, i.e. only additional \( AND \) gates are needed to mask the operands and some logic to generate the mask based on the instruction number of bits \( a \) to be accurately computed.

Figure 72 shows the structure of a bitmasking adder used to add the mantissae of two single-precision floating-point numbers. The result values of an anytime addition and a multiplication operation using bitmasking may differ slightly from Eq. (21), because the operands are masked. In the following, we assume that the exponents \( E_1 \) and \( E_2 \) of two operands with \( E_i = \sum_{k=0}^{E-1} e_{i,k} \cdot 2^k \),
Figure 72: Anytime adder using bitmasking for the addition of the mantissae \( x \) and \( y \) of two single-precision floating-point numbers. The mask generator generates a mask \( m \) in the form 1 ... 10 ... 0 in which the number of leading ones corresponds to \( a \in \mathbb{N}, 1 \leq a \leq 24 \). The depicted adder shown here is a carry-ripple-adder consisting of full-adders (FA) and a half-adder (HA), returning the result of the addition of the \( a \) most-significant bits of \( x \) and \( y \).

\[ E_2 \leq E_1 \text{ holds w.l.o.g., thus } \Delta E = E_1 - E_2 \geq 0. \]  
The result \( r_{\text{bitmask}} \) of an anytime addition with bitmasking is expressed in Eq. (22).

\[
\begin{align*}
MSum & = \sum_{k=M+1-a}^{M} m_{1,k} \cdot 2^{(k-M)} + 2^{-\Delta E} \cdot \left( \sum_{j=M+1-a+\Delta E}^{M} m_{2,j} \cdot 2^{j-M} \right) \\
\Delta r_{\text{bitmask}} & = (-1)^{s_2-s_1} \cdot 2^{E_1-B} \cdot MSum 
\end{align*}
\]  

(22)

In [5], hardware implementations for the above anytime instructions are presented and an analysis of area, power, and relative errors against an IEEE 754 single-precision implementation are provided based on one million instructions on uniformly distributed random operand pairs, both for additions and multiplications, respectively.

Moreover, error margins as well as error/performance trade-offs were investigated for applications from digital signal processing, image processing, and linear algebra for 10,000 randomly generated uniformly distributed input
6.5 Anytime Instruction Processors (AIP)

Figure 73: Average error of an iterative Jacobi algorithm for solving a linear system of equations with $n_e = 4$ equations and $n_v = 4$ variables against an IEEE 754 single precision implementation. Shown are four different plots for different values of a termination criterion $c$ that is reached once the $c$ most-significant bits of all computed results do not change any more between subsequent iterations. For different values of $a$, the min, max, and average errors over 10,000 runs are plotted for each value of $a$ that is kept constant during each execution of the algorithm. Figure adapted from [5].

Data sets including an iterative Jacobi solver for linear systems of equations of the form $Ax = b$ with

$$
A = \begin{bmatrix}
a_{00} & a_{01} & \cdots & a_{0n} \\
a_{10} & a_{11} & \cdots & a_{1n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{n0} & a_{n1} & \cdots & a_{nn}
\end{bmatrix},
x = \begin{bmatrix}
x_0 \\
x_1 \\
\vdots \\
x_n
\end{bmatrix},
b = \begin{bmatrix}
b_0 \\
b_1 \\
\vdots \\
b_n
\end{bmatrix}.
$$

The Jacobi algorithm solves the system of linear equations iteratively until the result for $x$ converges.

Figures 73 and 74 present some results comparing anytime instructions with IEEE 754 single-precision floating-point computations for a Jacobi iterative linear system of equations solver application and a matrix of size 4×4.

Overall, it can be observed that convergence can be reached faster the higher the gap between the number of bits used for the computation $a$ and the number of bits used for determining convergence $c$ is, as long as $a < c$. In [5], it was also shown that exploiting the programmable accuracy for floating-point additions and multiplications using AIP can save up to 15% in energy per floating-point addition, respectively 12% per multiplication when computing.
Figure 74: Average execution time normalised to the time of the program running at full accuracy \((a = 24)\) on an AIP processor of an iterative Jacobi algorithm with \(n_e = 4\) equations and \(n_v = 4\) variables for different levels of accuracy, indicated by the number of calculated most-significant bits \(a\) in relation to the execution of the algorithm operating on the single-precision floating-point format at full accuracy \((a = 24)\). The termination criterion is thereby reached once the \(c\) most-significant bits of all computed values do not change any more between subsequent iterations. \(a\) is kept constant over the execution of the program. Figure adapted from [5].

just \(a = 10\) most-significant mantissa bits (bitmasking approach) while still retaining an average error of below 0.1%.

### 6.6 Real-time Scheduling of I/O Data Transfers

TCPAs provide not only an excellent architecture for massively parallel execution, but together with our extensive work on compilation techniques (see Section 11.2), it allows the mapping and execution of nested loop programs resulting in a powerful loop accelerator. This architecture is built around the fundamental concept that most loop programs can be described in the polyhedral model. Here, the time and location at which any operation inside the loop is executed can be computed statically at compile time by linear transformations. This means that the start times of all operations are already known at compile time and that the complete control flow is static at runtime. More specifically: The execution time of a loop nest is not only predictable, but known to be static prior to execution. This is achieved by keeping all internal operations inside a TCPA synchronous. But in contrast to these local operations, the execution of a loop nest requires also external resources—mainly the loading and storing of input data and output data, respectively. In the case of input data, these memory accesses are defined indirectly by internal operations with data dependencies to external memory or vice versa in
6.6 Real-time Scheduling of I/O Data Transfers

the case of output data. This means that all external data on which an internal operation depends must be loaded into the TCPA prior to the execution of the operation, while all written output data must be stored after the execution of an operation. For this purpose, the TCPA contains a set of small memory banks that surround the PE array, as shown in Figure 67. These memory banks are then grouped into so-called I/O buffers. During execution, these buffers must be readily filled with input data and cleared from output data so that all internal operations that either require input data or produce output data can be carried out at their planned start time. If this is not the case, the TCPA stalls its execution until the delayed data transfer is completed and the execution of the loop program can continue. This does obviously break the computed runtime. Consequently, we must acknowledge that the timing predictability of TCPAs is based on the general assumption that no load or store operation interferes with the computed loop schedule.

In order to solve this issue, we investigated in [28, 29] a novel approach for handling I/O data transfers, which combines techniques throughout the stack, including (a) the generation of suitable blocks of data at compile time (Section 6.6.1), (b) the formalisation of a real-time I/O scheduling problem (Section 6.6.2), and (c) the design of a novel hardware unit called Loop I/O Controller (Section 6.6.3). This flow is also depicted in Figure 75.

6.6.1 Generation of Data Blocks

An \( n \)-dimensional loop program is characterised by an \( n \)-dimensional iteration space in which each point describes a single loop iteration. In order to map such a program onto a TCPA, this space is then tiled into subspaces that are assigned to individual PEs afterwards. Here, a subspace is called input space if it contains iterations that require data from an input variable. In the case of an output variable, it is called output space. Each point in these input and output spaces describes a single read or write from external memory and, since the PEs cannot access the main memory directly, the I/O buffers surrounding the PEs must be used as intermediate memory. This means that for every input and output space present at a PE, there must be a local memory bank inside an I/O buffer responsible for providing access to the specific variable. More specifically, prior to the execution of an iteration inside an input space, the data that is required by the iteration must be loaded into the associated bank, while after the execution of an iteration inside an output space, the written data must be stored in the associated bank. However, as element-wise reads and writes to external memory are inefficient in almost all memory architectures (for example, DDR), we want to avoid any single access in favour
for $i_0 \leftarrow 0$ to $N - 1$ do
  for $i_1 \leftarrow 0$ to $M - 1$ do
    $b[i_0, i_1] = B[i_1]$ if $i_0 = 0$
    $b[i_0, i_1] = b[i_0 - 1, i_1]$ if $i_0 > 0$
    $t[i_0, i_1] = A[i_0, i_1] \times b[i_0, i_1]$
    $u[i_0, i_1] = t[i_0, i_1]$ if $i_1 = 0$
    $u[i_0, i_1] = u[i_0, i_1 - 1] + t[i_0, i_1]$ if $i_1 > 0$
  end for
end for

Figure 75: The mapping of a matrix-vector multiplication $C = A \times B$ onto the processing elements (PE) of a TCPA is shown. Each tile of the tiled iteration space for $N = 6, M = 11$ (lower right corner) is assigned to one PE shown on the left. The grey boxes group input data (green) and output data (red) into tiles, marking blocks of data that need to be transferred either from memory to a buffer bank or vice versa. All inputs of PE $0, 0$ that belong to variable $A$ are mapped to bank $b = 4$, whose corresponding transfer space $\mathcal{K}_b$ is shown below. Here, the green nodes denote the time and address of all reads on variable $A$ from PE $0, 0$. Each transfer $K^b$ describes an address range of size $s_b$ starting with a start address $a$ that must be fetched into bank $b = 4$ at the corresponding deadline $t$ by a DMA controller.
of burst transfers of multiple consecutively stored data words. Thus, we rather want to identify suitable blocks of data in the external memory, which are then transferred in a single burst between the TCPA and external memory.

We divide the input, respectively output space assigned to each bank into multiple congruent subspaces, so that each subspace contains all iterations whose input or output data are transferred by a single and distinct block transfer. These subspaces form the so-called inter-tile space $\mathcal{K}_b^B$ of bank $b$ and describe all necessary data transfers affecting bank $b$. We further identify for every transfer $K^B \in \mathcal{K}_b^B$ the time $t$, referring to the very first access to any data element, the minimum address $a$ that is accessed, and the size $B_b$, which equals the distance between the minimal and maximal referred address. Note that the time $t$ determines the deadline of a corresponding block transfer task from memory to the respective bank. For output variables, it marks the deadline at which a previously written block must have been transferred to the memory in order to reuse its buffer location. Summarising, for all $K^B \in \mathcal{K}_b^B$, the corresponding transfer of size $B_b$ between external memory at address $a$ and bank $b$ must be completed by cycle $t$.

Although the above tiling works for any tile sizes that partition a given tiled iteration space, the choice of the parameters has an obvious influence on the number of blocks that need to be transferred and their size. As a result, we investigated methods for finding appropriate tilings in [29] and concluded that the blocks could be loaded sequentially. Hence, we can first update, i.e. fill or clear, half of the available memory bank and subsequently update the other half while the PE has access to the previous block. This allows a simple implementation of so-called double buffering without too expensive bookmarking. With the resulting transfer space $\mathcal{K}_b^B$, transfer schedule vector $\lambda_b^{K^B}$, block address function $m_b^{K^B}$, and block size $B_b$, we can now compute the full sequence of required data transfers for each bank $b$. However, this approach is simply infeasible since the memory overhead for storing all buffer transfer requests in a list, respective memory inside the I/O controller is too expensive. Thus, it is mandatory to investigate a dynamic approach that generates and schedules job instances for each buffer at runtime.

6.6.2 Real-Time I/O Scheduling Problem

The problem of scheduling data transfers for TCPAs in time can be formalised as a real-time I/O scheduling problem $\Psi = (\tau_1, \tau_2, ..., \tau_q)$ with a one-to-one relation between a task $\tau_i$ and a buffer bank $b$. First, the corresponding data transfers $K^B \in \mathcal{K}_b^B$ can be represented as job instances $\tau_i^{K^B}$ of a task
As introduced earlier, each data transfer has a given strict deadline for its termination, hence providing a deadline $D(\tau_i^{KB}, k)$ for each job instance. As the loop application itself is assumed to be invoked periodically with a given period $P$, the $k$-th periodic deadline as well as release time are defined as follows:

$$D(\tau_i^{KB}, k + 1) = D(\tau_i^{KB}, k) + P$$
$$R(\tau_i^{KB}, k + 1) = R(\tau_i^{KB}, k) + P$$

As for the execution time $E(\tau_i^{KB}, k)$ of each job instance or data transfer, we assume it can vary as traditional memory is prone to jitter. For our analysis, we only assume the transfer latency of a non-interrupted data transfer $\tau_i^{KB}$ related to a task $\tau_i$ to be bounded. Hence, we can assume a worst-case transfer execution time WCET($\tau_i$) such that for each transfer execution time $E(\tau_i^{KB}, k)$ holds: $E(\tau_i^{KB}, k) \leq$ WCET($\tau_i$). In contrast, the earliest possible start times $R(\tau_i^{KB}, k)$ of the tasks are not only related to the loop schedule, but also dependent on the bookkeeping technique of the memory banks. By constraining the construction of data blocks fetched in one transfer, as illustrated earlier, we guarantee that each data element of a fetched block is accessed before any data element of a subsequent block. By organising each memory bank as a double-buffered FIFOs, thus two entries—one block can be currently accessed (either read or written) by the PE attached to it while the subsequently accessed block potentially being involved in a transfer from or to memory over the bus. Thus, the time instance when one block has been fully read or written marks the release time $R(\tau_i^{KB}, k)$ of the next job instance to be executed. Note that the difference between input and output is marginal as transferring a written block can be seen as logically equivalent to fetching one. Thus, the only difference is the initial filling of input and the final clearing of output blocks. In order to schedule a given data transfer task set $\Psi$, each individual job $\tau_i^{KB}$ of each period $k$ needs to be assigned a start time $S(\tau_i^{KB}, k)$ in such a way that release time and deadline satisfy $R(\tau_i^{KB}, k) \leq S(\tau_i^{KB}, k)$ and $S(\tau_i^{KB}, k) + E(\tau_i^{KB}, k) < D(\tau_i^{KB}, k)$, respectively.

In the following, we assume that all data transfers must be executed sequentially, because they take place over a bus (single resource), and that the individual transfers cannot be interrupted, hence a non-preemptive schedule is required. In our context of scheduling I/O transfers for TCPAs, a transfer scheduling algorithm is needed that provably satisfies all transfer deadlines of related uninterrupted task job instances for worst-case as well relaxed execution time assumptions. Here, a non-work-conserving scheduling approach (i.e. the scheduled resource is not required to be fully utilised) has been taken.
In [29], we introduced strict EDF, a non-preemptive non-work-conserving EDF variant, which strictly follows the sequence of occurring deadlines as follows: At every time step where an uninterruptible job instance can be planned, select the job instance with the smallest deadline for execution. But contradictory to the well-known EDF algorithm that only considers already released job instances, strict EDF additionally considers all unreleased next job instances of each periodic task $\tau_i$. As a result, it may select a job instance that has not been released yet, if it is the one with the smallest deadline. In this case, the scheduler inserts idle time until this job instance gets released\(^2\). This schedule is shown in Figure 76 for the matrix-vector multiplication example from Figure 75. The big advantage of strict EDF lies in its simplicity, making it possible to be efficiently implemented in hardware. The management of arriving transfer requests could be achieved using a simple priority queue containing the deadlines of the following job instance of each task, being already released or not. Contrary to a precomputed schedule of transfers (= job instances), which typically grows exponentially with the problem size (i.e. the product of loop bounds), our dynamic scheduler queue must only manage one request per buffer at a time, thus is independent of the problem size. The design and implementation of such an I/O controller are presented next.

6.6.3 Loop I/O Controller

The loop I/O controller (LION) architecture depicted as an overview in Figure 77 creates and implements a strict EDF [29] scheduled list of pending block transfers at runtime. Moreover, it is shown that our controller implementation can generate a new DMA request with a latency $L$ of just $L = 6$ clock cycles. The architecture itself consists of three main components: A priority queue, a configuration unit and a DMA engine. The individual components are introduced in the following, while a thorough discussion of data structure implementations within the components is provided with greater detail in [28].

6.6.4 Priority Queue

Different from traditional EDF approaches, strict EDF [29] requires only a single priority queue containing all released and unreleased tasks. Thus, the heart of LION is a single priority queue (PQ), which contains for each bank $b$ the current deadline $t$ at which it must be filled or cleared next. As

---

\(^2\) Adapted to our case of data transfer scheduling, the bus would idle until this job instance becomes executable.
Figure 76: I/O schedule of five overlapped matrix-vector multiplication executions according to Figure 75 with period $P = 18$ (top) and period $P = 36$ (bottom). Every row “PE $K_0, K_1: X$” shows all job instances (coloured boxes) involved in transferring data for variable $X \in \{A, B, C\}$ to the bank connected to PE $K_0, K_1$ and their corresponding deadline (black arrows). Transfer tasks in each colour belong to one matrix-vector multiplication run. For the sake of clarity, all data transfers are assumed to have the same execution time. For $P = 18$, not all deadlines can be met, while for $P = 36$, no deadline is missed. Figure adapted from [29].

As mentioned earlier, each bank $b$ has a corresponding precomputed transfer space $\mathcal{K}_b$, whose contained elements $K_b$ describe individual data transfers for which LION must generate one DMA request each. According to the also precomputed transfer schedule vector $\lambda_b^K$ for each buffer $b$, we can deduce that at any point in time, no more than one transfer $K_b$ can be active for being issued next. Thus, for each bank $b$, only the deadline $t$ of the currently active $K_b$ must be stored inside the priority queue. As a result, the priority queue at any point in time only needs to store exactly one pending request per buffer. The required capacity is thus independent of the loop nest and schedule. At runtime, LION just selects the bank $b$ whose current transfer $K_b$ has the lowest deadline $t$ among all banks and issues the corresponding DMA request. While issuing a DMA request in its DMA engine, LION immediately computes the next request of this buffer $b$ with updated deadline $t'$ and stores the result into the priority queue.
6.6 Real-time Scheduling of I/O Data Transfers

Figure 77: LION, the loop I/O controller architecture. It consists of a mixed binary trie heap-based priority queue, a configuration unit, and a DMA engine. For the sake of clarity, all control signals and the logic for loading the initial configuration are omitted. Figure adapted from [28].

6.6.5 Configuration Unit

Since the priority queue only retrieves and updates the bank \( b \) with the lowest deadline \( t \), LION stores additional information inside a configuration unit, as shown in the middle of Figure 77. It is addressed by the retrieved bank \( b \) and stores the current transfer \( K^B \), the transfer space \( \mathcal{K}_b^B \), the start address \( a \), the transfer size \( s_b \), the block address function \( m_{b}^K \), and finally, the transfer schedule vector \( \lambda_{b}^{K^B} \) of each bank.

6.6.6 DMA Engine

With the priority queue providing the bank \( b \) with the lowest deadline \( t \) and the configuration unit’s further information, the DMA engine performs two basic tasks: First, it issues the next DMA request based on the current configuration data, i.e. bank \( b \), start address \( a \) and size \( s_b \), by sending it to an attached DMA controller. Second, it computes for the considered bank \( b \) already the update information of the configuration memory and priority queue for the next transfer expected for \( b \). It does this by computing and updating the next transfer \( K'^B \in \mathcal{K}_b^B \), deadline \( t' \) and next start address \( a' \), respectively. After this insertion into the PQ, the controller immediately
continues retrieving the next transfer with the smallest deadline in the PQ, and progresses as long as there still exists any pending request.

The entire proposed architecture provides a problem-independent solution for generating all necessary data transfers dynamically at runtime by solving the resulting real-time task scheduling problem directly in hardware, thus solving the I/O scheduling problem for TCPAs.

6.7 Conclusions and Acknowledgement

In summary, TCPAs permit a time-predictable as well as safe and low power processing [13, 18] of many computationally intensive nested loop programs. This has been shown for many applications such as layer-parallel processing of neural networks [1, 14, 16], real-time hand-sign recognition [15], and inverted pendulum control [7].

Finally, we would like to acknowledge the following list of persons for their very valuable contributions to the ideas, concepts and implementation of TCPAs through many years of research: Andreas Becher, Srinivas Boppu, Faramarz Khosravi, Vahid Lari, Shravan Muddasani, and Éricles Sousa.

References


References


6 Invasive Tightly-Coupled Processor Arrays


7 Power-Efficient Invasive Loosely-Coupled MPSoCs

Jörg Henkel, Andreas Herkersdorf, Heba Khdr, Martin Rapp, Mark Sagi, Mohammed Bakr Sikal, Thomas Wild

Abstract With increasing power densities in modern multi- and many-core processors, an intelligent power and thermal management becomes more and more important. It has to find appropriate trade-offs between contradictory requirements in terms of energy-efficient or reliable operation of the system and performance of applications. This applies also for invasive computing with its resource-aware programming concept. In this chapter, we cover different approaches to power and thermal management for invasive computing. As a starting point, we propose Thermal Safe Power (TSP) which allows better exploitation of the nominal processor performance than the usually applied Thermal Design Power (TDP), while guaranteeing thermal safety. Then we look at two approaches that use static rules, one for assigning individual i-lets to the cores within a compute tile and the other for selecting the voltage/frequency (V/f) levels of the cores jointly with mapping applications to cores. The major part of this chapter covers techniques that employ machine learning (ML)-based approaches to power/thermal management to cope with some of the limitations of the rule-based management techniques. We study several sub-problems of power/thermal management: application mapping and V/f level selection at application arrival time, application migration and application-aware boosting to adapt to dynamics in the workload during execution, and run-time power estimation and forecasting, which make up the foundation for power/thermal management.

7.1 Introduction

The ever-increasing integration density of integrated circuits and limits of voltage scaling lead to increasing power densities, which render appropriate power and thermal management concepts pivotal when running applications on a modern multi- or many-core processor. The overall goal is to reduce thermal stress and thus to prevent premature degradation and allow for reliable operation of such devices. Therefore, not only application performance has to be considered but also power efficiency and thermal constraints when managing the processor resources. In the context of invasive computing
with its resource-aware programming approach, this not only applies when applications want to (re-)invade resources and request the assignment of processor cores but also during system runtime to cope with varying load conditions. In both situations, an appropriate resource and power/thermal management is required that can help to enforce performance and thermal bounds. When assigning processor cores, the applications operating points, which are determined at design time and characterised by the number of used cores as well as their V/f settings, are taken into account. With the associated power budgets, the processor should be operated in a thermally safe manner. However, as application workloads may be unknown or vary with the input data, it also has to be guaranteed that both performance and power dissipation stay within the specified target bounds during system runtime.

Therefore, the resource management of an invasive compute architecture needs not only a mechanism for assigning processor cores among competing applications, which is done by an agent system, but also a run-time power/thermal management at the system level to cope with dynamically changing situations.

Considering all this we will first have a look at foundations of chip power densities and thermal safety. We will show that the usually applied concept of a constant Thermal Design Power (TDP) value is sub-optimal and potentially leads to either unused thermal headroom or to thermal violations. Therefore, we propose a new metric called Thermal Safe Power (TSP) which specifies a power budget which depends on the number of active cores and allows to better exploit the nominal processing capacity of an MPSoC. Then we look at (semi-)static, rule-based power and thermal management concepts. The first concept is related to application mapping and addresses the assignment of application threads (or i-lets in the context of invasive computing) to cores of a compute tile and is implemented as a hardware accelerator called Core i-let Controller (CiC). It is based on a rule set which correlates application requirements with the current status of the compute tile to optimise for individual or for a combination of multiple performance metrics. Then, \( DsRem \) is presented to jointly select the application-to-core assignments and the V/f levels of the cores. Additionally, \( DsRem \) determines the location of inactive cores to improve heat dissipation from active cores.

Finally, we present several ML-based techniques to cope with some of the limitations of the rule-based management, such as requiring design-time application models. These techniques tackle the different sub-problems of run-time power/thermal management. First, \( TcRM \) proactively considers shared resource contention between parallel applications while mapping ap-
lications and selecting V/f levels at their arrival times to avoid performance violations. It exploits the trade-off between performance and temperature during application mapping. Second, we present PCMig, a power- and cache-aware application migration technique to dynamically adjust the mapping to dynamics in the workload during execution to maximise the performance considering trade-offs between power budget and memory access latency. Third, SmartBoost proactively adjusts the V/f levels during execution to maximise the performance while avoiding thermal violations. It takes into account application characteristics to boost or throttle the applications that profit most and are impaired least, respectively. Finally, we present techniques to estimate the current power consumption of the processor and even forecast future power consumption. By providing such information, these techniques make up the foundation for power/thermal management.

### 7.2 Thermally Safe Power Constraints

A major step towards dealing with thermal issues on the chip is through efficient power budgeting techniques. Usually, system designers use Thermal Design Power (TDP) as a chip-level power budget, since it is the highest expected sustainable power while running real applications [8]. However, using a single and constant chip-level power budget might be either thermally unsafe or pessimistic, depending on the number of active cores on the chip. To clarify this further, we show in Figure 78 the maximum steady-state temperatures on a 16-core chip (estimated by HotSpot [7]) when equally distributing several per-chip power budgets among the active cores. It is assumed that the critical temperature of the chip is 80°C. As it can be observed, a per-chip power budget causes thermal violations when few cores run on the chip, because the whole power budget of the chip is consumed by just few cores, leading to high power density, thereby thermal violations. However, when the majority of cores are active, per-chip power budgets lead to significant thermal headroom. Using constant per-core power budgets, e.g. 8.06 W, (green line in Figure 78), can be a good compromise, since the temperature never exceeds the critical temperature. However, there is still unexploited thermal headroom. A more efficient power budgeting would use different per-core power budgets depending on the number of active cores, so that the maximum steady-state temperature of all cores is 80°C. That is precisely our proposed Thermal Safe Power (TSP)[18]. Thus, TSP is an abstraction that provides safe power constraints as a function of the number of active cores. For a specific floor plan and ambient temperature, TSP can be computed offline, in order to obtain safe power constraints for the worst cases, allowing the system designers to
abstract from mapping decisions. Moreover, TSP can also be computed online, for a particular mapping of active cores and ambient temperature. Our simulation experiments presented in [18] show that using TSP as a power constraint results in 12% higher average performance, compared to using constant power budgets (both per-chip and per-core).

In [11, 17], we extended the concept of TSP to consider core heterogeneity within a chip. In particular, if two cores with different areas consume the same power amount, the temperature of the smallest core will be higher, since its power density is higher. Therefore, for heterogeneous cores, the thermally safe power density needs to be obtained. However, depending on the running application and the micro-architecture of the executing core, TSP might be underutilised on some cores, while other cores could consume more power if available. Therefore, we propose to adapt TSP value at runtime to consider heterogeneous power characteristics of applications and cores. Applying this adaptive constraint allows further performance improvements without leading to thermal violations. In Figure 79, we show an example of our presented simulation results in [11], where a heterogeneous multi-core system is considered. This example shows how the TSP has been adopted for different core types as aforementioned. The power densities of the cores are below the adapted TSP values, and thereby no thermal violation is observed.

7.3 Rule-based Power/Thermal Management

7.3.1 Core i-lot Controller

A very coarse-grained approach to control power dissipation and temperature of processor cores is through mapping of applications and in a finer granularity
7.3 Rule-based Power/Thermal Management

Figure 79: Adaptive Thermally Safe Power Density Constraint to consider the heterogeneity in application/core power characteristics.

Table 14: Sensor weights for different application classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>ID</th>
<th>Fill Level</th>
<th>CPU Util.</th>
<th>CPU Temp.</th>
<th>Arr. Rate</th>
<th>Service Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-latency</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Comp.-intensive</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Low-temperature</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Multi-objective A</td>
<td>3</td>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Multi-objective B</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.3</td>
<td>0.3</td>
</tr>
</tbody>
</table>

of threads to cores. In the context of invasive computing it is the mapping of i-lets to cores. With the Core i-let Controller (CiC), a hardware-based concept has been implemented that applies a (semi-)static, rule-based assignment within an invasive compute tile. The CiC receives i-lets to be processed on the tile from the network adapter and assigns them to individual cores by writing them into core-specific FIFOs. Every core—after having finished its previous i-let—fetches the next one from its FIFO and executes it until completion.

The CiC uses a rule set, which considers the requirements of the application issuing i-lets as well as the current status of the compute tile when deciding on the assignment. The information on the requirements is transported within the i-let as a class identifier. The tile status is measured via sensor values for power dissipation, temperature and load of the individual cores and also for the fill levels and service/arrival rates of the FIFOs within the CiC. When determining the core for an arriving i-let the sensor values are weighted with factors specific to the application class and are combined into per-core cost values. The core with the lowest cost is chosen.

Table 14 shows an example for the weighting factors of different application classes. These application classes may have several metrics combined and
thus multiple objectives are considered on the assignment. Or in case of only one non-zero weighting factor, a single property like temperature or latency is optimised. These values make up the rule base, which can be modified at any time during system operation. The basic concept of the CiC has been published in [20] and a more elaborate version for the application in HPC clusters under the name Thread Control Unit (TCU) in [21].

The internal architecture of the CiC is shown in Figure 80. The class ID field of an arriving i-let is forwarded to the rule base for selecting the appropriate weights for the associated application. Together with the sensor values, which are aggregated and normalised by the monitor aggregator, they are sent to the queue selector. This entity performs the cost calculation and determines the FIFO queue in which the i-let is stored for processing. The CiC has been implemented and prototyped on FPGA and has been integrated into the invasive compute architecture, which makes up the basis of the demonstrators described in Chapter 15. On the FPGA it runs at 50 MHz and can perform 150 k mappings per second. It requires roughly 16% of the slices relative to the four LEON3 processor cores of a tile.

7.3.2 Thermal-Constrained Application Mapping and Voltage/Frequency (V/f) Selection

Besides application mapping, selecting V/f levels of the cores considering application characteristics is necessary to optimise for performance under power/temperature constraints. Therefore, in [10], we proposed *DsRem* that jointly selects application-to-core mappings and the V/f levels of the cores, aiming at maximising the overall system performance under a temperature constraint. In particular, *DsRem* determines the number of active cores that
should be allocated to each application and the V/f levels of these cores, taking into account the instruction-level parallelism and thread-level parallelism of the applications. Moreover, DsRem selects the positioning of inactive cores so that heat dissipation of active cores is improved, and thereby their temperatures are reduced. That, in turn, might allow to increase the V/f levels of the active cores leading to further performance improvement. Figure 81 illustrates how the locations of active and inactive cores affect the thermal profile of the chip. Our evaluations presented in [10] show that DsRem achieves an average of 34% performance gain compared to a thermal-aware state-of-the-art technique without leading to any thermal violations.

To cope with the continuous increase in the number of cores on the chip, managing chip temperature and system level needs to be conducted in a distributed manner. Therefore, in [12], we proposed distributed thermal management that employs multi-agent system, so that each agent is responsible for managing a subset of cores. In particular, one agent is instantiated upon arrival of an application, to obtain cores for that application and then manage these cores, by selecting their power states, i.e. active and inactive, and their V/f levels. The ultimate goal of the agent decisions is to maximise the overall system performance under a temperature constraint. Our evaluations in [12] show how our distributed thermal management technique is scalable with the increasing number of cores on the chip.

### 7.4 ML-based Power/Thermal Management and Modelling

While rule-based resource management tackles some of the challenges in power/thermal management, it ignores three critical aspects within the optimisation. First, it relies on design-time knowledge of the applications, such as performance or power models. Such information is often not available. Second, it statically assigns mappings or V/f levels to applications. However,
changing application execution phases and varying system workload require
to dynamically adjust the management. Third, it ignores shared resource
contention between applications running in parallel because the design-time
application models cannot represent this. This leads to overestimation of the
performance, and hence to suboptimal management.

These problems are complex because multiple factors like CPU microarchitec-
ture, floor plan, cooling, etc., play a role [23]. In addition, proactive manage-
ment is required, e.g. to consider the impact of shared resource contention
in advance to proactively counteract. Finally, the optimisation needs to cope
with unknown applications, for which no design-time knowledge is available.
These dimensions can not be tackled by traditional rule-based management.

ML provides powerful algorithms for system-level optimisation [22] because it
enables to build prediction models that cope with the complexity and general-
ise to unseen scenarios, such as unseen applications, by using generic features
like performance counters. These prediction models enable to implement
proactive management. This section discusses several ML-based techniques
for power and thermal management of many-core processors.

The techniques presented in this section are evaluated using the HotSniper [19]
framework, which allows to perform multi-program simulation of multithread-
ed applications with full modelling of shared resource contention, while still
offering a reasonable simulation time. HotSniper integrates the Sniper [5]
multi-/many-core simulator for performance simulation with McPAT [13] for

7.4.1 Application Mapping and V/f Scaling

While parallel computing on manycores has helped increase the overall system
performance, the heat transfer between multiple running cores can lead to
the creation of thermal hotspots, thus leading to thermal violations that can
impact the reliability of the chip, or even permanently damage it. In addition,
contention for the shared resources on the chip, e.g. the last-level cache (LLC)
can result in execution slowdown for the running applications, potentially
leading to performance constraint violations. Clustered manycore systems
can reduce chip-level contention by sharing the LLC only among the cores of
the same cluster. This indeed reduces the chip-level contention, but at the
same time, does not help with contention impacts on the same cluster. Recent
state-of-the-art system-level resource management techniques attempt to
reduce this cluster-level contention by proposing to co-map, i.e. map to the
same cluster memory-intense and compute-intense applications. While this
7.4 ML-based Power/Thermal Management and Modelling

Arriving application: cholesky

Already running application (x264 / bodytrack)

<table>
<thead>
<tr>
<th></th>
<th>Cluster 1</th>
<th>Cluster 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC accesses</td>
<td>53 \times 10^6 / s</td>
<td>9 \times 10^6 / s</td>
</tr>
<tr>
<td>Power</td>
<td>2.9 W</td>
<td>3.4 W</td>
</tr>
</tbody>
</table>

Arriving application (cholesky) when mapped to cluster 1 / 2

<table>
<thead>
<tr>
<th></th>
<th>Cluster 1</th>
<th>Cluster 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time</td>
<td>95 ms</td>
<td>83 ms</td>
</tr>
<tr>
<td>Temperature</td>
<td>72 C</td>
<td>81 C</td>
</tr>
</tbody>
</table>

Figure 82: Mapping cholesky to Cluster 1 or to Cluster 2 exposes a trade-off between cache contention and temperature.

approach reduces the contention impacts at cluster-level, it might lead to higher chip temperatures due to the high power consumption of compute-intensive applications. This trade-off is illustrated in Figure 82. x264 and bodytrack from PARSEC [3] and SPLASH-2 [30], are running on separate clusters on a clustered manycore. x264 is a memory-intensive application, reflected in its high number of LLC accesses per second and low power consumption, compared to the compute-intensive bodytrack. A new application, cholesky, arrives to the system and needs to be mapped to one of the clusters. As expected, the response time of cholesky when co-mapped with x264 on Cluster 1 is longer than its response time when co-mapped with bodytrack on Cluster 2, due to the slowdown induced by LLC contention on Cluster 1. On the other hand, co-mapping cholesky with bodytrack leads to an increase in the average cluster temperature by 9 °C, due to the high power consumption of bodytrack. In summary, mapping applications to clusters exposes a trade-off between cache contention and temperature.

At run time, such a resource management decision, i.e. application-to-cluster mapping, needs to be made at the arrival time of applications to the system. Since an application’s memory intensity varies throughout its execution, depending on the arrival times of applications, the cache contention-induced slowdown from their co-mapping would differ, making the number of possible combinations grow exponentially. Consequently, design-time profiling of the parallel execution of applications is not feasible. Moreover, slowdown prediction is too complex to be represented by an analytical model. To gain an insight into this complexity, Table 15 shows the slowdown incurred by applications from PARSEC [3] and SPLASH-2 [30], when co-mapped with
Table 15: Contention-induced slowdown incurred by App A due to App B.

<table>
<thead>
<tr>
<th>App B</th>
<th>lu.cont</th>
<th>cholesky</th>
<th>lu.ncont</th>
<th>streamclus.</th>
</tr>
</thead>
<tbody>
<tr>
<td>lu.cont</td>
<td>28.9%</td>
<td>12.5%</td>
<td>20.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>cholesky</td>
<td>27.2%</td>
<td>8.3%</td>
<td>17.4%</td>
<td>0.0%</td>
</tr>
<tr>
<td>lu.ncont</td>
<td>26.7%</td>
<td>18.8%</td>
<td>18.8%</td>
<td>0.0%</td>
</tr>
<tr>
<td>streamclus.</td>
<td>20.8%</td>
<td>3.6%</td>
<td>22.0%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

other applications. The first observation is that streamcluster is not slowed down by any application, while it slows down all applications it is co-mapped with. Secondly, application pairs show different slowdown behaviours. For instance, lu.cont suffers a slowdown of 20.8% and 27.2% from streamcluster and cholesky, respectively. However, the opposite trend is observed when lu.ncont is co-mapped with the same two applications; lu.ncont suffers a slowdown of 22.0% from streamcluster but only 17.4% from cholesky.

To this end, we propose TcRM[29], a thermal- and contention-aware resource management technique to determine application-to-cluster, thread-to-core mapping and the required V/f levels of the clusters that compensate for the contention-induced slowdown such that the performance constraints of all applications are satisfied, while minimising temperature. TcRM addresses the challenge of slowdown prediction by designing and training a neural network (NN) model at design time, to predict the cache-induced slowdown between two co-mapped applications on the same cluster at runtime, as shown in Figure 83. Upon arrival of an application A, TcRM looks for eligible clusters to host it. An eligible cluster must have enough free cores to execute application A, and there must be at least one V/f level that can compensate for the predicted slowdowns of all potential co-mapped applications on that cluster. Considering the slowdown predicted by the NN model, the new response time of applications is estimated. If the performance constraint of each application on the cluster is satisfied, this cluster is added to the set of eligible clusters. In the second step, the goal is to find a hosting cluster and the set of cores that minimise the peak steady-state temperature on the chip. The eligible cluster and the corresponding thread-to-core mapping option with the lowest peak steady-state temperature are selected. When an application leaves the system, TcRM resets the cluster’s frequency to the minimum V/f level that satisfies the performance constraint of the remaining applications.
7.4 ML-based Power/Thermal Management and Modelling

Figure 83: TcRM flow: at design time, training data is generated to train the NN model, which is used at runtime to predict the contention-induced slowdown between applications.

With an overhead of only 1% of the average execution time of applications, TcRM significantly reduces the temperature by 30% on average compared to the state-of-the-art, while satisfying the performance constraints of all individual applications. This is mainly due to the high accuracy of the NN prediction model, which achieves a low root-mean-square error (RMSE) score of only 2%, with 0.99-quantile of the prediction errors $\varepsilon = 5\%$, while maintaining a low inference overhead of 3 $\mu$s.

7.4.2 Power- and Cache-Aware Application Migration

Application mapping on thermally-constrained many-core processors with distributed shared memory needs to make a trade-off between minimising the average memory latency and maximising the power budget (Figure 84). Examples for such architectures are the RBCC, as introduced in Chapter 9, or static non-uniform cache access (S-NUCA). The average memory access latency depends on the average hop count on the network-on-chip (NoC) between the core that executes the application and the core that holds the memory. Cores close to the centre of the processor have the lowest average distance to all memory banks, which are distributed among all tiles. Therefore, to minimise the average memory latency, all applications should be mapped
Figure 84: Application mapping on many-core processors with distributed yet shared memory needs to make a trade-off between optimizing for power budget or memory latency.

close to the centre of the processor. However, mapping all applications close to the centre means they are also mapped close to each other. To avoid formation of a thermal hotspot, the power budget of the applications needs to be reduced. This reduces the feasible V/f levels, and ultimately reduces the performance. Instead, power budgets are maximised if applications are mapped far from each other, which prevents thermal hotspots even at higher power consumption. However, this would result in mapping applications far from the centre of the processor, i.e. lead to a high average memory latency, and thereby reduce the performance. Maximum performance can only be reached when the mapping makes a trade-off between the two.

The performance-maximising trade-off depends on application characteristics and the system utilisation. In particular, memory-bound applications require a low memory latency, whereas compute-bound applications require a high power budget. Additionally, thermal hotspots are more critical at high system utilisation. Both factors change over time. For instance, application characteristics change with the execution phases. Consequently, the mapping needs to be constantly adapted using application migration. This section presents an ML-based technique to optimise the performance under a thermal constraint.

In our work in [24], we study the S-NUCA many-core architecture. S-NUCA distributes the LLC banks among all cores, resulting in an application to access data that is residing in many different tiles, which ultimately leads to the trade-off between memory latency and power budget discussed above. Figure 85 visualises our power- and cache-aware application migration PCMig. It employs an NN to predict the impact of a migration on the performance
before executing the migration. This allows to achieve proactive management because no migration is executed without predicting its impact first.

The process to train the NN is depicted in Figure 85a. First, many applications are executed at different operating points (mappings and power budgets) to record their performance traces. Training data can be generated from these traces by comparing the performance of an application at different operating points to quantify the impact of a potential migration. This is repeated per execution phase of several applications and for different pairs of operating points. The training data is then used to train an NN model. The model achieves an average prediction error of less than $0.125 \cdot 10^9$ instructions per second (IPS).

The run-time application migration technique (Figure 85b) is invoked periodically to continuously adjust the application mapping to changing execution phases and system utilisation. First, migration candidates are generated. Then, the NN model is used to predict the performance impact of each migration. Finally, the best migration is executed and the process repeats. As the overhead directly depends on the number of migration candidates, their number should be small. At the same time, beneficial migrations should not be discarded too early. We select all migrations that migrate a single thread and migrations that swap two threads. Because migration is invoked periodically, the policy is capable of iteratively optimising the mapping.
We evaluate \textit{PCMig} on a 64-core many-core processor with S-NUCA cache architecture. Each core has a private 32 KB L1 cache, and holds a single 128 KB bank of the shared LLC. The processor is modelled to be fabricated in 14 nm. We create random workloads of 20 applications from \textit{PARSEC} \cite{3} and compare \textit{PCMig} to an application-agnostic mapping \textit{PCGov} \cite{25}, and to state-of-the-art application migration for performance maximisation \textit{Defrag} \cite{16}. Figure 86 visualises the results for varying application arrival rates. \textit{PCMig} increases the performance (lower average application response time) by up to 18\% over the state-of-the-art. This is achieved by dynamically adjusting the trade-off between memory latency and power budget according to the application characteristics and system utilisation. Thereby, the NN performance prediction model enables us to achieve proactive management for unseen workloads.

### 7.4.3 Application-Aware Boosting

Dynamic voltage and frequency scaling (DVFS) is a widely adopted technique that has a great impact on the performance and power. \textit{Boosting} is the problem of maximising the performance under a thermal constraint using DVFS. State-of-the-practice boosting like Intel \textit{TurboBoost} \cite{9} completely ignores application specifics. State-of-the-art techniques consider some information about the running applications but do so incompletely. For instance, many works select applications with high IPS for boosting because they tend to be compute-bound and, hence, would benefit most from an increased V/f level. Figure 87 shows a motivational example running \textit{SPLASH-2} \cite{30} \textit{lu.cont} and \textit{radix}. Without boosting (Figure 87a), both applications run at the same frequency of 2.0 GHz. \textit{Lu.cont} has a much higher IPS. Figure 87b shows that \textit{boosting the low-IPS radix} and throttling \textit{lu.cont} to maintain the same temperature improves the overall system performance by reducing the execu-
7.4 ML-based Power/Thermal Management and Modelling

Figure 87: Boosting low-IPS radix significantly improves the overall performance. This appears counter-intuitive at first but is a key observation contributing to the efficiency of SmartBoost.

The execution time of radix by 33% while increasing the execution time of lu.cont by only 2%. This is unlike what is expected. The reasons are that 1) lu.cont has high power consumption, which allows to balance boosting radix by 1 GHz with throttling lu.cont by only 100 MHz to maintain the same temperature, and 2) radix is compute-bound despite its low IPS. The key observation of our work in [26] is that boosting needs to jointly consider the sensitivities of performance, power, and temperature within the optimisation, leading to the development of SmartBoost.

To simplify boosting decisions, we develop a novel boostability metric that integrates the sensitivities of performance, power, and temperature. Since the application-dependent sensitivities of performance and power cannot be measured directly at run time, we employ an NN model to estimate these properties from run-time performance counter readings.

The core idea of the boostability metric $b$ is to quantify the relation of the performance change of an application to the change in the hotspot temperature $T_h$ when changing the application’s V/f level $f$. This relation is divided into three factors: the application-dependent sensitivity of the performance $s_{perf}$ and power $s_{pw}$ on the V/f level, and the mapping-dependent sensitivity of the hotspot temperature $s_{temp}$ on the application’s power:

$$b = \frac{\partial \text{IPS}}{\partial T_h} \cdot \frac{1}{\text{IPS}} = \frac{\partial \text{IPS}}{\partial f} \cdot \frac{1}{\text{IPS}} \cdot \frac{\partial T_h}{\partial f} = s_{perf} \cdot s_{temp} \cdot s_{pw}$$

(23)
Figure 88: The sensitivity model creation process in SmartBoost comprises profiling benchmarks to create training data and training an NN model.

The best boosting benefit is obtained by *boosting the application with the highest value of the boostability*. When throttling is required to avoid thermal violations, *the application with the lowest boostability value should be throttled first* to minimise the performance penalty for maintaining thermal safety.

Computing the *boostability* metric requires estimates for the three sensitivity values. The sensitivity of the temperature only depends on the core that the application is mapped to and the current hotspot location, and, hence, is static, allowing to characterise it at design time. The other two sensitivities depend on the application and cannot be measured directly. We develop a multitask NN model to predict these metrics from performance counter readings. Figure 88 presents the process to create training data. First, traces of applications at different operating points are collected. Since we target boosting in this work, operating points differ in their V/f level. Then, training data is generated by comparing traces of the same application at different V/f levels to quantify the change in performance and power. Finally, an NN model is trained on data obtained on some applications and tested on data obtained from other applications. Our model achieves a low prediction error of 0.80% per frequency step for the performance, and 13 mW/step for the power.

Finally, our SmartBoost technique monitors the steady-state temperature and periodically computes the boostability per running application. In case of thermal headroom or thermal violations, the application with the highest or lowest *boostability* is selected to boost or throttle, respectively. In a third optimisation step, SmartBoost checks whether throttling one application to be able to boost another one improves the overall performance. The sensitivities of power and temperature are additionally used to proactively check the impact on the temperature before changing V/f levels.
7.4 ML-based Power/Thermal Management and Modelling

We evaluate SmartBoost on a 64-core symmetric many-core processor with per-core DVFS between 1.0 GHz and 4.0 GHz with steps of 100 MHz. The processor is modelled to be fabricated in 14 nm. We create random workloads of 20 applications from PARSEC and SPLASH-2 and compare SmartBoost to the state-of-the-practice Intel TurboBoost [9] and the state-of-the-art DTPM [2]. Figure 89 shows the performance and temperature with the three techniques. SmartBoost significantly improves the performance over TurboBoost and DTPM while having negligible thermal violations. The reasons are that TurboBoost boosts/throttles cores only considering the transient temperature, i.e. unaware of application characteristics. DTPM considers application characteristics, but incompletely. For instance, it does not consider power or temperature when selecting an application to throttle. In contrast, SmartBoost makes boosting decisions based on the boostability metric, which is enabled by the NN model to estimate sensitivities of performance and power for unseen applications.

7.4.4 Run-time Power Estimation and Forecasting

As shown in the previous sections, modern resource management for multi- and many-core processors needs reliable run-time power consumption information. Obtaining such power information through sensors at core level is cost prohibitive. Therefore, model-based run-time power estimation, with core activity as input and estimated core level power values as output, is widely used [15]. Accurate run-time power estimations are needed for effective reactive resource management algorithms, i.e. the algorithms can only adapt to already observed changes in the processors power states. Extending power estimations into the future—forecasting future power states—would
enable proactive resource management. Such proactive resource management algorithms can allow for even better processor performance.

A motivational example for both run-time estimation of current power consumption as well as for forecasting future power consumption is shown in Figure 90. For reactive resource management, both the power changes in the green (smaller transients) as well as the red area (large transient) need to be accurately known to best manage power consumption as well as thermal headroom. In addition, the rapid increase in power consumption indicated in the red area shows an opportunity for a proactive resource management algorithm. A purely reactive algorithm can not adapt fast enough (< 1ms) to avoid hitting the thermal constraint, thus degrading performance.

In two of our works, we make the following selected contributions toward high accuracy run-time power estimation and power forecasting: a low-overhead non-linear model to capture non-linear relations between the observable activity, i.e. available performance counters, and power consumption [27]; a long short-term memory (LSTM) recurrent neural network (RNN)-based model accurately forecasting rapid changes in power consumption [28].

**Non-linear power estimation** [27] Previous work [14] has shown that the relation between performance counters and power consumption can be non-linear. Our goal is to generate a non-linear model for power estimation with low run-time overhead compared to simpler linear models. To achieve this, we
propose the following methodology as shown in Figure 91. Similar to previous work, first a suite of workloads are executed on the target-system while both power consumption and performance counters are traced, the design-time benchmarking phase. During the design-time statistical analysis phase, we determine suitable non-linear transformation functions $f_{\text{trans}}$ for the performance counter inputs. These transformation functions are chosen such that the modelling accuracy of subsequently generated linear models, which use the non-linearly transformed performance counter inputs, are maximised. The final phase is then the run-time estimation where the performance counters inputs are continuously transformed using $f_{\text{trans}}$ and used as inputs for the linear model producing the core level power estimations.

The non-linear functions $f_{\text{trans}}$ are chosen through an algorithm which aims to maximise the pairwise Pearson correlation coefficient between each performance counter input and core power. In this way, possible non-linear relations between performance counters and core power are transformed into more linear relations. This allows the subsequent linear model to achieve overall higher accuracy by having the non-linear transformation block, consisting of the $f_{\text{trans}}$ functions, applied to its inputs. A wide range of non-linear functions, e.g. $n$-th roots, logarithms, exponential and trigonometric, are evaluated by the algorithm. Afterwards, an ordinary least squares regression model is generated with the transformed performance counters as input. The equation for the dynamic core power consumption $P_{\text{dyn},t}$ at time $t$ with $m$ performance counter inputs is then given as follows:

$$P_{\text{dyn},t} = \sum_{i=1}^{m} (f_{\text{trans},i}(PC_{i,t}) \cdot \beta_i) \cdot f_t \cdot V_t^2,$$

(24)
where $f^{\text{trans},i}$ is the chosen transformation function for the $i$-th performance counter $PC_i$, $\beta_i$ is the corresponding linear regression coefficient, $f_t$ is the frequency and $V_t$ the voltage.

The proposed methodology is evaluated with the same simulation framework as described in previous sections. A 16-core 4x4 system with Gainestown core architecture and a NoC interconnect is simulated executing PARSEC and SPLASH-2 benchmarks. Two modelling approaches based on related work as well as the proposed non-linear approach and a so-called poly-d approach were investigated. While [14] used polynomial models up to a degree of 3 for power modelling, it kept the degree uniform for all model inputs. We therefore added another polynomial model (poly-d) which allowed higher degrees and aimed to find the optimal degree for each input, i.e., similar to the transformation which uses unique functions for each input. To obtain statistically robust results, a 5-fold cross-validation is used for model generation and validation. The resulting relative RMSE values for all four modelling approaches are shown in Table 16. The non-linear models show consistently lower mean, median and worst-case errors compared to the linear and polynomial-based models. Although the decreases in power estimation error might seem comparatively small, these can have a noticeable, and with our approach, avoidable impact on the performance of resource management algorithms. Finally, to assess the trade-off in higher model accuracy vs. higher overhead of using the transformation functions during run-time, two possible implementations were compared to a pure linear model. The overheads for today’s large out-of-order cores are conservatively estimated to be <0.1% of area for a dedicated microcontroller implementation and <0.2% of run-time for a software implementation.

**Power forecasting** [28] The goal of this work is to forecast rapid changes in power consumption such that a proactive resource management algorithm can more effectively adapt the system state to meet power and thermal require-
ments. Most previous work falls into history-based approaches, e.g. [4], or autoregressive (AR)-based modelling approaches, e.g. [6]. In the first approach, workloads are observed during their normal execution and information on their power behaviour is stored in case the workload is observed again on the system. In the second approach, a lightweight AR model is generated and continuously adapted during run-time. Contrasting these previous works, our approach trains a more complex LSTM RNN model at design time to capture more general workload/power patterns. Both, input complexity in form of performance counters as well as model complexity, are larger for our approach. This necessitates careful optimisation of the LSTM RNN architecture and designing a single LSTM RNN model such that it is reusable over a wide range of V/f operating points. The latter is necessary as storing and executing different LSTM RNN models for each possible V/f operating point is prohibitive in regard to run-time overhead.

The LSTM RNNs input and output relations are defined as follows:

\[
\hat{P}_{\text{dyn}}[k + \tau] = \text{LSTM}(P_{\text{inv}}[k, ..., k - l], P_{\text{Cinv}}[k, ..., k - l]) \cdot f_{GHZ}[k] \cdot V^2[k], \quad (25)
\]

where \(\hat{P}_{\text{dyn}}[k + \tau]\) is the expected future average power consumption at discrete time \(k + \tau\), \(P_{\text{inv}}[k, ..., k - l]\) is the current and previous power consumption up to \(l\) time steps into the past and \(P_{\text{Cinv}}[k, ..., k - l]\) are the current and past performance counters values. The power input \(P_{\text{inv}}\) has been normalised for each time step \(k\) in regard to V/f at that time step and \(P_{\text{Cinv}}\) has been normalised in regard to frequency for each respective time step. By these normalisations, we make the model inputs as invariant as possible towards changes in the V/f operating points enabling the LSTM RNNs trained at a fixed V/f operating point to be reused for other V/f operating points.

We determine a suitable LSTM RNN architecture in regard to the number of layers and number of LSTM neurons per-layer through a two-layered grid-search methodology as shown in Figure 92. The first grid search iterates over architectures with a changing number of layers \(l\). For each \(l\), we also iterate over the number of neurons \(n\) in the architecture while keeping the number of neurons equal for each layer of that specific architecture, i.e. the LSTM RNNs keeps a rectangular shape. With this we determine a viable \(l_{opt}\) for which enough complexity is achieved to capture the complex workload/power patterns for forecasting. The second grid search then iterates with a fixed number of layers \(l_{opt}\) over different neuron distributions to further increase forecasting accuracy while not adding unnecessary model complexity.
To generate the power forecasting LSTM RNN models, we use the same experimental setup as for the previous power estimation models. After choosing the best neuron architecture, we train an additional 100 LSTM RNNs with that specific architecture for further optimisation. By choosing the LSTM RNNs with lowest error on a validation data set for final evaluation we ensure that LSTM RNNs are not just overfitted on the used training data. The evaluation is done on a holdout data set which was neither used for architecture optimisation nor for final training. The accuracy on this holdout data set for power forecasting is shown in Table 17 for our proposed LSTM RNN as well as two reference-based approaches using history tables and AR models. We compare these for both mean absolute percentage error (MAPE) and instantaneous worst case-error and see that our approach is more accurate in both metrics compared to the the history table-based approach. However, the AR-based approach shows better MAPE performance and worse instantaneous worst case-error performance. Closer investigation of the different forecasts reveals that the AR-based model better adapts to gradual changes of power levels while not being able to forecast rapid power changes. In a final step we also investigated the robustness of the LSTM RNNs for changes in the V/f operating points and found that forecasting accuracy only degrades minimally.
Table 17: Forecasting MAPE values and instantaneous worst case-error values for our LSTM RNN approach and reference-based approaches for a 1ms forecasting period [28].

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MAPE</td>
<td>30 %</td>
<td>8 %</td>
<td>10 %</td>
</tr>
<tr>
<td>Inst. WC</td>
<td>5.3 W</td>
<td>3.5 W</td>
<td>2.0 W</td>
</tr>
</tbody>
</table>

Our proposed LSTM RNNs-based power forecasting methodology allows for high accuracy in forecasting rapid power changes while having slightly lower accuracy in forecasting average power levels compared to the state-of-the-art.

7.5 Conclusion

This chapter has presented several power/thermal management techniques and models to help increasing power and thermal efficiency in MPSoCs. At first, we have discussed several rule-based techniques, which depend on design-time knowledge about the applications within their decision making process. These techniques, however, cannot be applied in open systems, where the applications are not a priori known and cannot cope with runtime variations, as well. Therefore, we have employed ML to achieve our target objectives, since it enables to build prediction models that cope with runtime variations and generalise to unseen scenarios, such as unseen applications, by using generic features like performance counters. We have demonstrated how our ML-based prediction models and power/thermal management techniques were able to increase power and thermal efficiency of MPSoCs even for unknown execution scenarios. Although our proposed ML models can generalise to unknown scenarios at runtime, they still require design-time knowledge that should be representative for potential runtime scenarios. As future work, online learning can be employed to achieve the learning on the fly without the need for any design-time knowledge. On the other hand, this might lead to significant runtime overhead for the training. Thus, the opportunities and challenges of employing online learning to support power/thermal management for MPSoCs need to be investigated.

We would like to acknowledge the valuable contributions of Anh Vu Doan, Santiago Pagani, Ravi Pujari, and Muhammad Shafique.
References


8 Generation of Distributed Monitors and Run-Time Verification of Invasive Applications

Daniel Müller-Gritschneder, Ulf Schlichtmann, Alexandra Listl, Marcel Mettler, Li Zhang

Abstract In embedded applications, not only the functional behaviour is important but also non-functional requirements such as latency, throughput, power, reliability, and security properties. As those properties are only predictable to a certain extent in the invasive many-core computing paradigm, e.g. due to workload variation, they must be controlled by run-time requirements enforcers and run-time management strategies, which make their decisions based on run-time monitors. In this chapter, we study the design of those monitors. Therefore, we first present the design of an architecture for run-time verification (RV) to verify control flow and timing requirements. Secondly, we propose a scalable ASIC temperature emulator for FPGA prototypes to evaluate thermal requirements already at early design stages. Thirdly, we present an actor-based benchmark generation approach for the rapid generation of benchmark applications and ultimately, we combine those three concepts to an FPGA-based evaluation approach for run-time management strategies.

8.1 Introduction

In embedded applications, e.g. in the fields of robotics, telecommunications and autonomous driving, not only the functional behaviour of a system is important but also non-functional requirements such as latency, throughput, power, reliability, and security properties.

In order to assure that the system satisfies these requirements, one can conduct design-time analysis. For such analysis, it is important to reduce the sources of variability of non-functional properties in the system. Here, the invasive computing paradigm addresses this challenge by a strong isolation between the applications running on the same platform, see Chapter 1. Yet, there still remain sources of run-time unpredictability, which may impact the non-functional properties of an application program. For example, there still may occur conflicts on non-isolated resources, such as the main memory,
which influence the temporal behaviour of the application. Additionally, thermal interference between applications may cause thermal violations that require a reduction of the frequency. Finally, random hardware faults, due to radiation or ageing, may alter the behaviour of the hardware. If such effects are considered only with design-time analysis, this may lead to a very conservative system design with large margins. Hence, there exists also the solution to manage the system at run time and react to the system state in order to enforce the requirements. Such run-time requirements enforcement (RRE), as introduced in Chapter 2 as well as other run-time management strategies, e.g. to improve thermal isolation (see e.g. Chapter 3), depend on monitors that can observe the status of the system in order to enable an appropriate reaction. This monitoring of requirements and triggering system reactions is also known as run-time verification (RV).

This chapter describes a scalable RV architecture for tiled invasive many-core architectures in Section 8.2, which supports so-called logical and timing supervision. With this system, it is possible to verify the control flow and the temporal behaviour of an application at run time. To additionally evaluate the thermal behaviour of the processor at an early design stage, we subsequently propose a scalable application-specific integrated circuit (ASIC) temperature emulation approach for field-programmable gate array (FPGA) prototypes in Section 8.3. As RRE and thermal management strategies must be tested for applications of various run-time characteristics, we present a benchmark generator for actor-based applications in Section 8.4. Finally, we show in Section 8.5 that the combination of those three concepts can be used to build a framework for the evaluation of run-time management strategies.

### 8.2 Run-Time Verification of Invasive Many-Core Systems

Exhaustive verification techniques do not scale with the complexity of today’s heterogeneous many-core systems due to their unpredictability [12, 17]. Even with a strong isolation of applications, as being the default in the invasive computing paradigm, several sources of unpredictability remain.

In order to address this research challenge, additional methods are deployed to verify system specifications at run time. Those methods are studied in the field of RV [2, 7, 17]. RV is a lightweight verification technique that instruments a system under verification to extract its current status using events. The trace of events is then analysed by a run-time monitor, which assesses whether the system specifications are met or not.
On tiled many-core systems, additional challenges arise for the use of RV. By design, the system is organised decentrally by multiple tiles that additionally host multiple processing cores. Each of the cores generates a particular trace of events such that no global trace is available per se. However, all monitors in the system must not only receive all events their specifications are defined on but must also have a consistent view of their detection order. In [21], we address this research challenge by proposing a NoC-based RV architecture for tiled many-core processors. In the following, we first present the supported requirements of the RV architecture, followed by its architectural design and a discussion on its benefits and limitations.

### 8.2.1 Supported Run-Time Requirements

The expressiveness of the supported run-time requirements is a key feature of an RV architecture. However, in hardware, one needs to balance the expressiveness of the requirements with their associated hardware overhead. Inspired by [24], we decided to support logical and timing requirements in our architecture. Logical requirements can be used to verify the control flow of an application in order to detect synchronisation errors between software elements. And on the other hand, timing requirements can be used to verify the temporal behaviour of software and thus reveal deadlocks, livelocks and the incorrect allocation of execution times. In this subsection, we first introduce our notation of events followed by the formal definition of logical and timing requirements.

#### Supported Events

All requirements are defined based on a set of events $E$, where each event $e$ is defined by a 3-tuple $(i_e, t_e, i_c)$. In this context, $i_e$ defines the ID of the event and $t_e$ the timestamp of the event. As the monitors operate in clusters that verify requirements based on the same set of events, each event is assigned to a cluster ID $i_c$ to identify the monitoring cluster, the event is used in. This results in the advantage that the monitors only need to operate on a short event ID, which saves additional resources.

Furthermore, the architecture supports the following trigger conditions for each event.

- **Program Counter Address Detected**
  The event is fired on a specific checkpoint, i.e. program counter address.

- **Power Corridor Violation**
  The event is fired if the power consumption is outside a predefined power
interval. Thus, it is possible to detect spikes in the power consumption, which might disturb nearby analogue circuits within critical sections of the application.

- **Temperature Corridor Violation**
  The event is fired if the core temperature is outside a predefined temperature interval. Thus, it is possible to detect thermal violations and to verify thermal managers.

- **Result-Range Violation of Arithmetic Operation**
  The event is fired if the result of a specific arithmetic operation is outside a predefined range. Thus, it is possible to verify not only the control flow of an application but also the application data.

**Logical Requirements**

Logical supervision can be used to verify the control flow of an application or generally causal relationships that must hold at application run-time. In [21], we defined a logical requirement by a finite deterministic automaton with positive and negative accept states. In this context, positive accept states $v_i \in V_t$ represent the fulfilment of a logical requirement and negative accept states $v_i \in V_f$ the violation of a logical requirement. Thus, the automaton follows a 3-valued semantic and issues a verdict based on its current state $v_i$.

$$\text{verdict} = \begin{cases} \top & v_i \in V_t \\ \bot & v_i \in V_f \\ ? & \text{else} \end{cases}$$

Figure 93: An example of an automata-based requirement [21].
The automaton can either be manually constructed or synthesised from a linear temporal logic (LTL) requirement using the open source tool LamaConv [14]. An example of a logical requirement is given in Figure 93 as automaton and in Eq. (27) as LTL requirement.

$$\neg e_{closed} \land (e_{opened} \rightarrow \neg(e_{exit} \lor e_{opened}) \cup e_{closed})$$  

(27)

It requires that $e_{exit}$ and $e_{opened}$ do not follow $e_{opened}$ until $e_{closed}$ has been detected. Such a requirement can be used to verify that critical tasks close their resources after use.

### Timing Requirements

Timing Supervision can be used to verify the temporal behaviour of a code section or an application. We define a timing requirement by a start event $e_{start}$ and a stop event $e_{stop}$, which start and stop the timer, respectively, and by a timing interval $[T_{min}, T_{max}]$, which defines the accepted timing corridor. In this context, a timing requirement is fulfilled if the measured time between the start and stop event is within the interval and violated if it is outside. Similar to the logical requirements, this results in a 3-valued semantic, which issues a verdict based on the current timer value $t$.

$$\text{verdict} = \begin{cases} 
\top & t \in [T_{min}, T_{max}] \land e_{stop} \text{ detected} \\
\bot & t > T_{max} \lor (t < T_{min} \land e_{stop} \text{ detected}) \\
? & \text{else} 
\end{cases}$$  

(28)

Popular timing requirements, used in embedded systems, are set via windowed watchdog timers. Here, the software has to regularly "pat" the dog and reset the timers, e.g. to check that no deadlocks occur within a task. For complex applications, a set of such requirements can be used to monitor the progress and the deadlines of real-time-critical applications.

### 8.2.2 Architectural Design

The proposed RV architecture is illustrated in blue in Figure 94. It comprises a set of probes, a set of tracing routers and a set of tile monitors. Each probe is connected to a core by its tracing interface. Out of the trace data, the probes filter the preconfigured events and forward them via the tile monitor to the network adapter of the tracing router. Using the specialised router design, our novel tracing interconnect SortNoC as shown in Figure 97 is able to sort the events based on their detection order and to distribute them to all tile monitors. The tile monitors additionally filter the globally sorted trace for
events on which they operate and verify the configured requirements. Once a monitor detects a violation, it raises an interrupt and the operating system (OS) can address it. A more detailed discussion on the design of the individual components is presented in the following.

**Architecture of a Probe**

In Figure 95, we present a modular architecture of a probe, which consists of a timestamp generator, a set of event detectors, an arbitration logic, and a configuration block. The *timestamp generator*, illustrated in purple, is composed of a free-running counter that issues the timestamps for the detected events. To ensure that the events are sorted correctly, the counters in all probes need to be synchronised. In an FPGA design, this is possible by a global reset signal, which is already connected to all slice registers. However, in an ASIC design, the counters must be reset by an internal boot event of the RV architecture. At this point in time, no other event is present in the SortNoC so that the latency to each tile monitor is known at design time. Using this latency it is possible to synchronise the counters by adjusting the initial values of the counters.

The *detectors*, illustrated in green, filter the tracing data down to an amount that is processable at run time using the preconfigured events. Thus, each detector issues a trace element when the configured trigger conditions are fulfilled. As the choice of detectors is highly application dependent, we only illustrate a selection of detectors based on the trigger conditions, present-
ed in Section 8.2.1. Nevertheless, further detector types can be integrated accordingly, e.g. a detector that triggers when ageing effects result in timing violations [16, 19, 20]. For the timing verification the end of period (EOP) detector is of special interest. It issues an event when the timestamp generator overflows. Thus, the timing checker is able to compute a cycle-accurate latency based on the timestamps of the start and stop event and the number of EOP events. As all counters are synchronised, it is sufficient to implement the EOP detector in a single probe in the complete RV architecture.

The arbitration logic, illustrated in grey, buffers the issued events and sorts them based on their timestamps. Thus, each probe provides a locally sorted trace to its tile monitor. Furthermore, the trigger conditions, the event IDs and the cluster IDs of each detector are stored in a configuration unit. This enables the reconfiguration of a probe even at application run-time.

**Architecture of a Tile Monitor**

The architecture of the tile monitor is presented in Figure 96. It consists of a network adapter, a set of checkers (automata processors (APs) and timing checkers) and a configuration memory. The network adapter represents the interface to the SortNoC. It injects the tile-local trace into the interconnect, where it is combined with the other tile-local traces to a globally sorted event trace. From the globally sorted event trace, the network adapter then filters the events with a matching cluster ID and forwards them to the implemented checkers.
The current implementation hosts two types of checkers: logical checkers and timing checkers. The logical checkers, illustrated in green, are implemented as APs using small 18 kB on-chip static random-access memorys (SRAMs). The memory blocks store for each conjunction of the detected event ID and the current state ID of the automaton, the next state ID of the automaton. Thus, it is possible to reconfigure the verified specification at run time. If the current state of the automaton corresponds to a negative acceptance state, i.e. a violation of the specification, an interrupt will be raised. The timing checkers, illustrated in purple, verify the timing requirements using the timestamps of the start and the stop event, and the number of EOP events. Thereby, the latency $\delta$ between two events can be computed using Eq. (29), where $N_{EOP}$ corresponds to the number of EOP events and $t_{EOP}$ describes the time between two successive EOP events.

$$\delta = t_{stop} - t_{start} + N_{EOP} \cdot t_{EOP}$$  \hspace{1cm} (29)$$

Furthermore, the timer needs to be able to detect overruns of a latency requirement without a stop event. Only in this way, it is possible to detect deadlocks in applications. This can be achieved using the condition in Eq. (30) to detect an overrun, where $W_t$ corresponds to the bit width of the time-stamp generator.

$$T_{max} \leq (N_{EOP} - 1) \cdot 2^{W_t}$$  \hspace{1cm} (30)$$

In this condition, it is crucial to decrement $N_{EOP}$ by one as the difference $t_{stop} - t_{start}$ may become negative. This may happen if the timestamp generator overruns between the start and the stop event.
Finally, the configuration memory, illustrated in orange, can be used to buffer different probe configurations. Thus, it is possible to request a reconfiguration of a probe with a minimal intrusiveness of one bus access. Nevertheless, it is still possible to run the RV architecture fully non-intrusively by configuring the monitors and the probes before the run-time of the application.

**SortNoC – A New Tracing Interconnect Architecture**

The SortNoC, illustrated in Figure 97, addresses the main challenge in the design of a decentralised RV architecture, the sorting and the distribution of the detected events. This can be achieved by a forward and a backward path in the interconnect. Along the forward path, all trace elements, here in green and yellow, are sent to a predefined target router (here $r_3$). Within this path, the events are sorted using a timestamp-based arbitration schema within the crossbar of each tracing router. Thus, each tracing router merges its input traces to a sorted output trace. This is only possible if events that have been issued at the same point in time also arrive simultaneously at the crossbar. Therefore, we implemented a delay stage at the local input port of each router, which balances the hop distance of each router to the target router. As a result, we can ensure that the green event arrives after the earlier issued yellow event at $r_3$ even though the green event got injected at the local input port of $r_3$. In general, the required delay of each router can be computed using Eq. (31),
where \( N_{diam} \) corresponds to the diameter of the NoC and \( N_{h2t} \) to the hop distance of a router to the target router.

\[
N_{delay} = N_{diam} + 1 - N_{h2t}
\]  

(31)

Using this schema, we obtain a globally sorted trace at the target router of the SortNoC. Thus, the trace elements only need to be broadcasted to all other trace routers along the backward path. As this path will always be free of congestions, all input buffers along this path can be replaced by registers to save additional hardware resources.

### 8.2.3 Benefits and Limitations of the Run-Time Verification (RV) System

In this section, we discuss the consequences of the design decisions for the RV system. The hardware-based implementation comes with multiple benefits. While software-based implementations change the temporal behaviour of applications [8] and introduce significant performance overheads [23], our hardware implementation is completely non-intrusive. Thus, it provides short detection latencies, as illustrated in Figure 98. Here, the marked regions correspond to the standard deviations of the measurements. On average, it takes 141 cycles from the detection of an event until the interrupt service routine is called, which enables a fast response to the violation of any specification. Nevertheless, a hardware implementation comes also with the drawback that the number and types of probes and monitors are already fixed at the design time of the system. Hence, the choice must be carefully decided.

Furthermore, the capability to generate a globally sorted event trace using SortNoC is a major benefit of the architecture. It allows the decentralised monitors to verify run-time requirements of complex applications on tiled many-core processors. However, the globally sorted trace comes with a throughput limitation of one event per cycle per system (EPCPS). This becomes visible by evaluating the interconnect latency for different event rates, as illustrated in
8.3 Thermal Emulation of ASICs on FPGA Prototypes

As introduced in Chapter 7, novel thermal and run-time management techniques have become increasingly sophisticated. As a result, the techniques need to be tested from early design stages on. At this point in time, the real platform is not available. Thus, management strategies must either be evaluated by pure simulation-based approaches or by FPGA-based approaches. While full-system simulators, such as gem5 [3], are orders of magnitudes slower than the real platform and fast simulators, such as Sniper [5], introduce inaccuracies by multiple levels of abstraction, FPGA prototypes are a common approach to drastically reduce the simulation time and to provide a

(a) The average latency and the standard deviation for a fixed 6x6 NoC architecture.

(b) The average latency and the standard deviation for a fixed injection rate of 0.1 EPCPS.

Figure 99: The scalability of the SortNoC approach.

Figure 99a for a 6x6 design. Nevertheless, the available throughput is still sufficient considering that the typical event detection rates in RV are in the order of milli- and microseconds [4, 10]. For example, on an 100-core processor, which computes 0.7 instructions per cycle per core, every 70th instruction could trigger an event. This is more than any software implementation could provide under a reasonable performance overhead constraint. Furthermore, the latencies of the interconnect scale linearly with the size of the architecture, as illustrated in Figure 99b, making it a good choice even for large-scale many-core processors.

Finally, the hardware overhead of the RV system is 51268 LUTs, 71188 FFs, 4 RAM36s and 128 RAM18s for a default configuration where the checkpoint and the out-of-range detectors support 32 events and where 32 logical and timing checkers are implemented within each tile monitor.

8.3 Thermal Emulation of ASICs on FPGA Prototypes
platform for early software development. However, the power consumption and the thermal behaviour of FPGAs and ASICs differ significantly due to the different implementation approaches. To address this challenge, we propose techniques to emulate the power consumption, the thermal behaviour and DVFS of ASICs on FPGA prototypes.

### 8.3.1 ASIC Power Emulation

The power consumption of a processor is an important metric for the design of thermal management strategies. For example, the techniques described in Chapter 7 limit the power consumption of a processor using DVFS to satisfy the thermal requirements of the chip. In order to test these techniques on FPGA prototypes, the power consumption of the real ASIC platform needs to be emulated. We study the design of such an ASIC power emulator for FPGA prototypes in [9, 18]. Thereby, we first synthesise the processor design and run gate-level simulations to characterise the switching activity of the processor for all instructions individually. In this process, we differentiate between cache misses and cache hits on the instruction and the data cache. This information can then be used by the Synopsys Tool PrimePower to simulate the power consumption of the pipeline, the register file, the instruction cache and the data cache for each scenario, respectively. In the power emulator, illustrated in Figure 100, the simulation results are stored in lookup tables (LUTs). Here, we use one LUT for the dynamic power consumption of each component and one LUT for the static power consumption of the complete core. At run time, the emulator selects the power consumption of the core out of the LUTs based on its status signals. Since the power consumption of the pipeline is predominantly determined by the executed instruction, we use the opcode and the pipeline stall signal to select the correct power consumption.
Similarly, the power consumption of the register file is determined based on the pipeline stall signal and the power consumption of the instruction and the data cache are based on the respective cache miss signals. With this process, we are able to estimate the static power consumption $P_{stat_0}$ and the dynamic power consumption $P_{dyn_0}$ at each cycle for a reference frequency $f_0$ and a reference supply voltage $V_{DD_0}$. As these may not correspond to the emulated $V/f$ level, the static power needs to be scaled according to Eq. (32) and the dynamic power needs to be scaled according to Eq. (33) in the $V_{DD}$-scaler and in the $f$-scaler, respectively.

$$\begin{align*}
P_{stat} &= P_{stat_0} \left(\frac{V_{DD}}{V_{DD_0}}\right) \\
P_{dyn} &= P_{dyn_0} \left(\frac{f}{f_0}\right) \left(\frac{V_{DD}}{V_{DD_0}}\right)^2
\end{align*}$$

### 8.3.2 ASIC Temperature Emulation

FPGAs have a different thermal behaviour than ASICs because both technologies use different methods to implement the same behaviour of the system. Hence, an FPGA prototype must emulate the thermal behaviour of its ASIC implementation using power emulation, as presented in Section 8.3.1, and a thermal model of the chip. The thermal model is typically based on a compact RC-thermal model, which is also used by the state-of-the-art thermal simulator HotSpot [13]. It comprises four conductive layers, which model the silicon die, the thermal interface material (TIM), the heat spreader and the heat sink, and a thermal node for each component of the processor on each layer. Furthermore, the thermal nodes are connected via thermal conductances and capacitances to model the thermal dependency.

Even though it is possible to directly implement the numerical solution of the thermal model, as presented in [1], it will not scale well. It requires a centralised design as all thermal nodes on the chip influence each other to a certain extent. Thus, the power consumption of each thermal node needs to be sent to the monitor and the temperature needs to be sent back to the thermal node. Especially on large-scale multi-FPGA ASIC prototypes, this introduces a significant overhead in terms of I/O and routing resources. Furthermore, the computational complexity of the numerical solution scales quadratically with the number of thermal nodes.

To overcome this challenge, we proposed a decentralised temperature emulation approach in [22]. Here, we compute the temperature of a thermal node by
a linear regression model that only depends on the power consumption of the node, the current temperature of the node and the temperature of all neighbouring nodes, as illustrated in Figure 101. Formally, the relationship for node \(i\) can be expressed by Eq. (34), where vector \(T_i(n) = [T_i^j(n)]_4\) corresponds to the temperature of the thermal node \(i\) on all conductive layers \(j\) at iteration \(n\), vector \(T_i,dir(n)\) corresponds to the temperature of the neighbouring thermal nodes of \(i\) in the cardinal direction \(dir\), matrix \(\beta_i\) corresponds to the regression matrix and scaler \(P_i(n)\) corresponds to the power consumption of node \(i\).

\[
T_i(n) = \beta_i \begin{bmatrix} P_i(n) \\ T_i(n-1) \\ T_i,north(n-1) \\ T_i,east(n-1) \\ T_i,south,0(n-1) \\ T_i,south,1(n-1) \\ T_i,west(n-1) \end{bmatrix}
\]  

(34)

If a node is located at the border of the floor plan and no direct neighbour is available, we use the ambient temperature instead. As thermal conductance varies at the border of the chip and the size of \(\beta\) depends on the number of neighbours, we fit the linear model for each thermal node independently against HotSpot. In order to evaluate the accuracy of this emulation approach against the HotSpot, we emulated the temperature of different processor architectures over \(1,000,000\) iterations and compute the mean absolute error (MAE) of the emulated temperature and the maximal MAE over all iterations. The results are illustrated in Table 18. For all architectures, the MAE is below \(0.03 \degree C\). Also, the maximal MAE is for most architectures less than \(0.05 \degree C\). Even the maximal MAE of the heterogeneous many-core processor, consisting
Table 18: The temperature emulation error for different architectures measured within periods of 2 μs intervals over a total emulation time of 64 ms.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Cores</th>
<th>Thermal Nodes</th>
<th>MAE</th>
<th>max (MAE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvasIC ProFPGA Platform</td>
<td>80</td>
<td>96</td>
<td>0.004 °C</td>
<td>0.034 °C</td>
</tr>
<tr>
<td>Alpha EV6</td>
<td>1</td>
<td>30</td>
<td>0.007 °C</td>
<td>0.049 °C</td>
</tr>
<tr>
<td>Heterogeneous Many-Core</td>
<td>92</td>
<td>147</td>
<td>0.022 °C</td>
<td>0.540 °C</td>
</tr>
<tr>
<td>Intel Gainestown Dual Core</td>
<td>2</td>
<td>13</td>
<td>0.008 °C</td>
<td>0.047 °C</td>
</tr>
</tbody>
</table>

(a) DVFS on an ASIC [25].

(b) DVFS Emulation on the FPGA [22].

Figure 102: The voltage and frequency transition between two V/f levels.

of various core types and accelerators, is well acceptable at 0.55 °C. The hardware overhead of the emulation approach is 1113 FF, 496 LUTs and 4 DSPs per thermal node on a Xilinx Virtex7 FPGA.

8.3.3 ASIC Dynamic Voltage Frequency Scaling Emulation

DVFS is typically used by run-time management strategies to balance the trade-off between the power consumption and the speed of a processor. In order to accurately evaluate the performance of those techniques, it is necessary to consider the energy and the timing overheads that each transition between two V/f levels introduces.

Figure 102a illustrates the time progression of the voltage and frequency levels during the transition between two V/f levels. In the case of up-scaling, the voltage converter first needs to increase the voltage before the frequency can be adjusted. This procedure delays the frequency transition by the voltage
transition time $\tau_x$, which introduces a timing overhead of $\frac{f_e-f_0}{f_1} \tau_x$ and an energy overhead of $E_{uc,up}$. Furthermore, the processor needs to be stalled while the phase-locked loop (PLL) adjusts the frequency, which introduces an additional timing overhead of the PLL lock time $\tau_{PLL}$ and an additional static energy overhead of $E_{PLL}$. In the case of down-scaling, the PLL adjusts the frequency before the voltage regulator decreases the supply voltage. As a result, the timing overhead is only introduced by the lock time of the PLL while the energy overhead is introduced by the static energy consumption $E_{PLL}$ during the lock time and the additional energy consumption $E_{uc,down}$ during the underclocking of the processor. In summary, the timing and the energy overhead are illustrated in Eq. (35) and Eq. (36), respectively.

$$\tau_O = \begin{cases} 
\tau_{PLL} + \frac{f_e-f_0}{f_e} \tau_x & \text{up-scaling} \\
\tau_{PLL} & \text{down-scaling}
\end{cases}$$ (35)

$$E_O = \begin{cases} 
E_{PLL} + E_{uc,up} & \text{up-scaling} \\
E_{PLL} + E_{uc,down} & \text{down-scaling}
\end{cases}$$ (36)

In order to consider these overheads on an FPGA prototype, we propose a DVFS emulator in [22] based on Figure 102b. In this process, we stall the processor pipeline during the frequency transition. Furthermore, we model the timing overhead, introduced by the voltage transition time, as a linear function in dependency of the difference in $V/f$ levels $\delta_{vf}$ between which the processor switches. This relationship is expressed in Eq. (37), where $\tau_m$ and $\tau_o$ are used to model the slope and the intercept of the function.

$$\tau_x \approx \tau_m \delta_{vf} + \tau_o$$ (37)

Similarly, we modelled the energy consumption of the processor during the voltage transition by a step-wise increase of the voltage levels. In this procedure, the slew rate of the voltage regulator can be calibrated by the transition time $\tau_e$ between two consecutive voltage levels. As the slew rate during down-scaling is typically faster than during upscaling, we use a parameter $\tau_{e,up}$ and $\tau_{e,down}$ for up- and downscaling, respectively. Furthermore, we model the voltage overshoots during up-scaling by a one-time energy overhead $E_{os}$. Hence, the FPGA-based emulation approach comprises five parameters $\tau_{e,up}$, $\tau_{e,down}$, $E_{os}$, $\tau_m$ and $\tau_o$, which can be calibrated for any combination of voltage regulators and PLLs using linear regression.
8.4 Generation of Invasive Applications

Finally, the processor speed is emulated by stalling the processor pipeline. In this process, the highest frequency corresponds to one active cycle and zero stall cycles. Similarly, the processor speed can be reduced by one third using two active cycles and one stall cycle. Hence, this approach enables to emulate any frequency on the emulation platform.

In order to evaluate the accuracy of this emulation approach, we calibrate it for three exemplary processors and voltage regulators. In Table 19, we compare the MAE and the mean relative error (MRE) of our approach with a macro model presented in [25]. The results show that our model outperforms the macro model in most characteristics. Even though the macro model achieves better results for the energy consumption of the processors during up-scaling $E_{uc,up}$, it should be noted that the absolute error of the FPGA model is sufficient.

### Table 19: The accuracy of the DVFS model.

<table>
<thead>
<tr>
<th>Processor</th>
<th>$E_{uc,up}$</th>
<th>$E_{uc,down}$</th>
<th>$\tau_{uc}$</th>
<th>FPGA-Model [22]</th>
<th>Macromodel [25]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2 Duo E6850</td>
<td>220 $\mu$J</td>
<td>19.0 $\mu$J</td>
<td>6.35 $\mu$s</td>
<td>7.97%</td>
<td>0.81%</td>
</tr>
<tr>
<td>with LTC3733</td>
<td></td>
<td></td>
<td></td>
<td>1.22%</td>
<td>13.9%</td>
</tr>
<tr>
<td>Samsung Exynos 4210</td>
<td>1.63 $\mu$J</td>
<td>0.27 $\mu$J</td>
<td>2.13 $\mu$s</td>
<td>10.9%</td>
<td>1.80%</td>
</tr>
<tr>
<td>with LTC3568</td>
<td></td>
<td></td>
<td></td>
<td>0.67%</td>
<td>3.56%</td>
</tr>
<tr>
<td>MSP430</td>
<td>8.31 $\mu$J</td>
<td>0.29 $\mu$J</td>
<td>5.23 $\mu$s</td>
<td>1.25%</td>
<td>5.01%</td>
</tr>
<tr>
<td>with LTC3632</td>
<td></td>
<td></td>
<td></td>
<td>0.02%</td>
<td>4.90%</td>
</tr>
</tbody>
</table>

8.4 Generation of Invasive Applications

Benchmarking is a common approach to evaluate the performance of computing systems. For that purpose, two different types of benchmarks are used: standard industrial benchmarks and synthetic benchmarks. Standard industrial benchmarks are selected real-world applications that are reasonable in size and complexity. Yet, they are typically difficult to port between hardware architectures, are not scalable and are only available to a limited
extent. Synthetic benchmarks on the other hand can be rapidly generated and thus are easily portable and scalable. Dujmović presents such a benchmark generator in [6] based on a recursive expansion (REX) process, which generates a procedural application with a specific breadth $B$ (i.e. the maximal number of statements per code block) and a depth $D$ (i.e. the maximal level of nesting).

In [22], we present an actor-based [11] benchmark generation method for for many-core processors based on the REX process, which we are going to use in Section 8.5 to evaluate the performance of run-time management strategies. For that reason, we extended the REX process by two additional parameters, which allows us to model arbitrary run-time characteristics. The first parameter $P_{fp}$ defines the probability that a variable assignment uses floating-point arithmetic. Hence, it controls the number of executed floating-point instructions, which typically consume more power than integer instructions. The second parameters $N_M$ defines the size of the memory on which the benchmark operates. Thus, it is possible to control the cache hit and miss rates of the generated application. Furthermore, we add an input/output interface for message passing.

In Algorithm 1, we present the extended REX procedure for many-core processors. First, input data of size $N_I$, of preceding actors are used to initialise the memory range of the actor. In the case that there is no preceding actor, the memory remains uninitialised. Subsequently, the code is generated using the $get\_statement$ procedure, which selects one out of $N$ different procedural statements. The code blocks within conditional statements or loops are in turn generated using $B$ statements. Thus, the REX recursively calls the $get\_statement$ procedure until either the maximal level of nesting is reached or an assignment statement has been selected, which terminates the expansion process. Finally, the output data of size $N_O$ of the actor is initialised by the local memory range and transferred to the preceding actor.

In Figure 103, we show that this procedure can be used to generate actors with various run-time characteristics. For that purpose, we generate 50 actors for each of the following input parameter combinations and evaluate their performance based on their execution time, their cache hold rates and their integer and floating-point instruction rate.

- Depth $D \in \{1, 2\}$
- Breadth $B \in \{2, 3\}$
- Floating-point probability $P_{fp} \in \{0, 0.25, ..., 1\}$
Algorithm 1 Recursive Expansion Process [22]

1: procedure rex(D, B, P_{fp}, N_M, N_I, N_O)
2: code = preprocessing(N_M, N_I)
3: code += statement(D, B, P_{fp}, N_M, I+rand()%(N-1))
4: code += postprocessing(N_M, N_O)
5: return code

7: procedure get_statement(D, B, P_{fp}, N_M, selector)
8: D = D – 1
9: if D == 0 then
10: selector = 0
11: switch selector do
12: case 0
13: assignment(P_{fp}, N_M) ▷ Terminate Expansion
14: case 1
15: ”if” + get_condition() + get_block(D, B, P_{fp}, N_M)
16: ...
17: case N – 1
18: ”do” + get_block(D, B, P_{fp}, N_M) + ”while” + get_condition()

20: procedure get_block(D, B, P_{fp}, N_M, selector)
21: block = ””
22: for 1 ← 0 to B – 1 do
23: block += get_statement(D, B, P_{fp}, N_M, rand()%N)

- Memory size $N_M \in \{256 B, 2028 B, 32768 B\}$

Figure 103a and Figure 103b show the instruction and data cache hold rate of the generated benchmarks over the execution time. The instruction cache (IC) hold rate shows a decrease until a run-time of 10 us. This behaviour is attributable to the low number of instructions in such short actors. As the execution time of the actors increases, also the variability of the hold rate increases. In contrast, the actors show a high data cache (DC) hold rate for short execution times. This behaviour can be explained by the fact that the data cache first needs to be filled before data locality can be exploited. Hence, we see a high variability in the hold rate for higher execution times. Furthermore, Figure 103c and Figure 103d show the integer and floating-point instruction rate over the execution time, respectively. The integer instruction rate ranges between 300 and 1400 1/us and shows a slight increase with the
execution time. This behaviour matches with the decreasing DC hold rates, as a lower hold rate enables a higher instruction count. Finally, the floating-point instruction rate ranges uniformly between 0 and 200 1/us. In summary, the evaluation shows that the extended REX process exhibits a great diversity in performance characteristics. Hence, such a design space exploration (DSE) can be used to generate tasks with various run-time characteristics out of which the best-suited actor for an arbitrary test scenario can be chosen.

8.5 FPGA-based Evaluation of Strategies for Thermal and Resource Management

The evaluation of thermal and resource management strategies on FPGA prototypes comes with many advantages over simulation-based methods. FPGA prototypes are by orders of magnitudes faster than full-system simulators. Hence, it is possible to evaluate the run-time management strategies on longer
and more complex applications. Furthermore, FPGA prototypes provide more accurate results, as simulators introduce inaccuracies by multiple levels of abstraction.

However, only ASIC temperature emulation is not sufficient to adequately evaluate run-time management strategies. A proper evaluation requires monitoring support to analyse the performance impacts of each management decision. Furthermore, a large set of benchmark applications is required, on which the management strategies can be tested and compared. Hence, we proposed in [22] an FPGA-based evaluation approach that combines our concepts on RV (Section 8.2), ASIC emulation (Section 8.3) and benchmark generation (Section 8.4). An overview of the complete framework is illustrated in Figure 104.

Here, the benchmark generator provides an arbitrary number of applications with different run-time characteristics. The applications can then be used to evaluate the strengths and weaknesses of the run-time management strategies. Therefore, the scheduler sends the pending jobs to the resource manager, which controls the dynamic thermal manager (DTM) and allocates the processing cores. Using the status signals of the cores and the emulated V/f levels, the ASIC emulation platform is able to emulate the power consumption of the processor and thus, also its temperature. The temperature and power values are then fed back to the resource manager to close the control loop. Thus, the resource manager is able to balance the trade-off between speed and power consumption. Finally, the RV architecture is used to monitor the status signals of the system and its power consumption and temperature to gain...
insights into the system behaviour based on which the run-time management strategies can be compared.

In the following, we demonstrate the evaluation approach on the proFPGA platform, see Chapter 15, hosting an 80-core processor. For that purpose, we compare a state-of-the-art thermal management technique based on power budgeting [26] with a state-of-practice hardware DTM [15]. The DTM monitors the temperature of the cores and reacts to any thermal violation by throttling down the V/f levels to the minimum level. Once the core temperatures decrease below a certain threshold, the V/f levels of the cores are re-set to their peak values, i.e. \( f_{max} \). While the DTM technique is reactive and non-predictable, power budgeting is based on static V/f levels, which are computed at design time. Thus, the worst-case schedule w.r.t. temperature of parallel running actors is computed at design time. Using the power profile of each actor, it is then possible to obtain a thermally-safe frequency \( f_{safe} \) for each of the actors.

For the comparison, we generate two actor graphs using the REX process introduced in Section 8.4, which are illustrated in Figure 105a and Figure 105b, respectively. The first actor graph follows the scatter-gather pattern, which is commonly used in parallel applications. In this case, \( a_{00} \) and \( a_{13} \) can never run in parallel with any other actors. On the other hand, \( a_{01} \) to \( a_{12} \) will always run in parallel. Hence, \( a_{00} \) and \( a_{13} \) are assigned to a high \( f_{safe} \), while the other
8.6 Conclusion

Actors are assigned to a lower $f_{safe}$. Comparing both thermal management strategies on the FPGA prototype yields an execution time of 172 ms for power budgeting and an execution time of 201 ms using the DTM. In this case, power budgeting does not introduce any pessimism and thus outperforms the DTM, which suffers from V/f transition overheads.

The second application consists of two actor graphs, which run in parallel. The first actor graph comprises an early parallel phase while the second actor graph comprises a late parallel phase. Using power budgeting, we needed to consider for each actor in the first graph, the worst-case scenario of the second graph. For example, the power budget of actor $a_{01}$ is computed based on the worst-case scenario, where the actors $a_{01}$ to $a_{04}$ and $a_{12}$ to $a_{15}$ run simultaneously even though this is an unlikely case. Comparing both thermal management strategies for this application yields an execution time of 222 ms for power budgeting and an execution time of 199 ms using the DTM. In this scenario, power budgeting is overly pessimistic so that it predominates the V/f transition overheads introduced by the DTM. In summary, the evaluation approach points out the strengths and weaknesses of both techniques using the interplay of benchmark generation, ASIC emulation and run-time verification.

8.6 Conclusion

This chapter presented the design of several run-time monitors to enable RRE in the context of invasive computing. First, we presented an RV architecture for the supervision of logical and timing requirements. In this concept, the requirements can be formulated based on events that support multiple trigger conditions including power and temperature corridors. In order to support the power and temperature evaluation already on FPGA prototypes, we also presented power and temperature monitors that emulate the behaviour of an ASIC implementation and consider the timing and energy overheads of DVFS. For the comparison between different run-time management strategies, we additionally presented an actor-based benchmark generator, which is capable of generating applications with a large variety of run-time characteristics. Finally, we combined all three works in an FPGA-based evaluation framework for thermal and resource management strategies. Here, the RV architecture is used to evaluate the thermal behaviour of the processors, the ASIC emulation approach is used to support thermal monitoring already on FPGA prototypes and the generated benchmarks enable an application independent analysis of the management strategy under test.
We would like to acknowledge the valuable contributions of Qingqing Chen, Elisabeth Glocker and Doris Schmitt-Landsiedel.

References


References


Invasive NoCs and Memory Hierarchies for Run-Time Adaptive MPSoCs

Jürgen Becker, Andreas Herkersdorf, Nidhi Anantharajaiah, Oliver Lenke, Akshay Srivatsa, Thomas Wild

Abstract The interconnect infrastructure and the memory subsystem take a decisive role when it comes to optimise the performance of compute architectures. In order to provide an efficient hardware platform adapted to the needs of invasive computing, specific optimisation concepts for both on-chip communication and memory hierarchy have been investigated, which are introduced in this chapter. The invasive NoC (iNoC) makes up the basis for differentiated communication services among the compute tiles of invasive architectures. Different variations in terms of the network topology as well as the used routing mechanisms have been studied to increase either performance, resilience to errors or efficiency. In respect to the memory subsystem, major contributions have been made in respect to region-based cache coherence (RBCC), which allows to dynamically resize shared memory regions within the tiled architecture to provide also bigger places in X10-based invasive programs. Further, near memory hardware acceleration has been studied to improve memory-intense processing and especially to support functions of the run-time system like copying graph data structures among memory partitions and thus speeding up the startup of i-lets on remote compute tiles.

9.1 Introduction

When evolving to multi- and many-core processor architectures with dozens or more cores, a scalable interconnect to allow efficient communication is pivotal as well-known bus-based infrastructures do not scale. Networks on Chip (NoCs) are therefore the choice to meet the communication requirements of such massively parallel architectures. Many different topologies and protocols have been proposed to cope with the challenges in this domain. Likewise, memory access in such architectures makes up a general bottleneck known as memory wall. Therefore, in addition to well-established cache hierarchies, physically distributing multiple memory blocks over the chip or allocating several off-chip memories as well as allocating hardware accelerators near or even within memory arrays are approaches to tackle this problem.
This chapter covers a selection of concepts proposed in the context of invasive computing to optimise inter-tile communication and to reduce memory bottlenecks. This new, resource-aware programming paradigm uses the PGAS (partitioned global address space) programming model based on X10 as programming language where shared memory programming is done within so-called places (see Section 1.3), whereas message passing occurs among places. In an invasive compute architecture an X10 place is originally mapped onto a single compute tile, which consists of 4 or 5 LEON3 processor cores that are connected via a shared bus. Further, a tile-local memory is attached to the bus as well as an L2, which caches remote memory accesses, e.g. to the global off-chip memory. To connect the tiles of the invasive computing architecture, we provided a special NoC called iNoC, which allows reserving parts of the NoC bandwidth between individual tiles to support predictable execution of applications stretching over several tiles [12] [22]. Additionally the iNoC supports best effort communication services and provides specific features helpful for invasive computing. Further, we have investigated variations of the NoC topology in terms of the hierarchy and special fast communication links among tiles as well as specific approaches for multicast and adaptive routing. A special NoC feature called in-NoC circuits (INCs) was implemented in [14] to aid the region-based cache coherence mechanism, which is mentioned next, in the fast exchange of coherence messages.

With respect to the memory subsystem of invasive architectures we introduced region-based cache coherence (RBCC) to extend the number of processor cores which are provided with a coherent view to the memory beyond a single compute tile [20]. Appropriate hardware extensions allow to dynamically change coherency regions and thus to provide the basis for re-sizing X10 places at system run time. Another improvement of the memory subsystem was the introduction of specific near memory hardware accelerators. One example is a graph copy accelerator to transfer graph data structures from one memory partition of the global memory to another while performing the necessary address translations on the fly [16]. This accelerator is speeding up the start of i-lets in remote compute tiles which profits from this hardware-based copying of the associated data to the memory partition of the target tile.

In addition to the concepts described in detail in this section, with smart eviction policies for cache memories we have also proposed further concepts for optimising the memory subsystem. A hybrid voting-based eviction policy (HyVE) for cache data management combines multiple stand-alone eviction policies together and evaluates them using concepts from voting theory [21]. Hardware accelerators to support inter-tile atomic operations, i.e. for proper
synchronisation mechanisms in tile-based architectures [18] or to enhance inter-tile queue management [17], have also been proposed in this context. A selection of the mentioned contributions to the communication infrastructure and the memory subsystem of the invasive compute architecture will be presented in the following.

9.2 iNoC as Basic Invasive Interconnect

The invasive NoC (iNoC) is the communication infrastructure, which connects all tiles of the invasive compute architecture and makes up the next hierarchy level on top of the bus-based intra-tile interconnect. In addition to regular best effort data transfer, the iNoC supports the reservation of communication channels with guaranteed bandwidth among tiles. With this feature, interconnect resources can be managed using the principles of invasive computing, analogue to the invasion of processor cores. This allows to give Quality-of-Service (QoS) guarantees to applications which have real-time, safety or security requirements and thus need predictable communication. A hybrid NoC router and a network adapter (NA) establishing the link to the intra-tile bus were developed which support guaranteed as well as best effort services based on virtual channels [10, 23]. Figure 106 shows the basic components of iNoC. As an additional feature to support invasive computing, the iNoC network adapter provides a specific DMA mode which allows to spawn an associated i-let on a remote tile and thus to start processing directly after the data transfer is finished.

Investigations of features in the areas of fault tolerance, security, power management and dark silicon were also conducted. In terms of fault tolerance, techniques to handle faults, caused e.g. by manufacturing defects or ageing effects were studied. A lightweight second layer network that allows coping with failed routers was implemented [11]. The feature supports localisation of
failed routers and establishing redundant routing paths using hardware units to circumvent broken nodes and links. It can also be used to save energy by manually disabling regular tiles and routers on its path. Investigations into extending the 2D NoC to 3D were also conducted, which resulted in a lower average latency for inter-tile communication [8]. Please note that the basic iNoC with its feature extensions covered in this section is a communication infrastructure independent from the SortNoC described in Section 8.2.2.

Prototyping, debugging and evaluation of large scale NoCs on a hardware implementation can get challenging. This problem is addressed in [13] where a hybrid prototyping approach is introduced that combines software-based virtual platforms (VPs) and FPGA prototyping. This concept provides FPGA resources for the NoC components under test and builds a virtual platform representing a larger many-core system for common evaluation. In addition, new design paradigms in NoCs were investigated next to make the iNoC more flexible, dynamic and adaptive to requirements of different applications. These include developing hierarchical multi-layered topologies with adaptive routing, dynamic multicast communication support and self-organising techniques. Brief descriptions of these features are highlighted next.

### 9.3 Adaptive Multi-Layered NoC and Dynamic Multicast Support

Increasingly, multiple applications of different criticality are being executed on the same system-on-chip (SoC) platform to reduce cost. If the NoCs on such systems can adapt to the requirements of different applications at run time, then the overall performance of the system can be improved. Static routing is usually preferred in the NoC when critical applications are using it, as they require guarantees. Therefore, it is challenging to utilise adaptive routing in mixed criticality NoCs. The design and implementation of a multi-layered NoC with an adaptive routing algorithm designed to be beneficial for mixed criticality systems is presented in [3].

#### 9.3.1 Multi-Layered Topologies

Mesh topologies are commonly implemented in NoCs due to their scalability, ease of implementation and uniform nature. Here, a multi-layered NoC targeting mixed criticality systems is presented. A run-time congestion-aware adaptive routing algorithm which routes packets on different layers of such a multi-layered NoC depending on their criticality to improve latency and
throughput was implemented. The adaptive routing algorithm is scalable and flexible and not based on tables.

Critical and non-critical packets usually share the same communication resources like links and routers. It can be beneficial, if the paths of critical and non-critical packets are decoupled to reduce the influence on each other. One option is to have dedicated links using virtual channels. In such cases they still share router resources and the arbitration policy on such routers can still have an impact on critical packets due to the behaviour of non-critical packets. Another option is to have dedicated physical links and routers for the critical packets. That needs a high amount of resources and can require the knowledge of mapping at design time. We combine these approaches by implementing layers in a NoC and having sections of layers occupied exclusively by critical applications. This resulted in a multi-layered mesh-based topology to take into account that a topology needs to be generic and uniform such that it can be used by a large set of applications and is not customised to a small set of applications.

![Diagram of a multi-layered NoC](image)

**Figure 107**: An example of the multilayered topology. A 5×5 3-layered NoC.

To demonstrate the advantages of such a topology and the associated routing algorithm we consider an example of a 5×5 configuration with three layers, as illustrated in Figure 107. In this example, Layer 3 is a 5×5 mesh topology represented by black links. Layer 2, denoted by orange links interconnects alternate routers of Layer 3, forming a 3×3 mesh topology. Layer 1, the top most layer in this example denoted by green links, connects alternate routers belonging to Layer 2. Therefore, the number of ports of a router varies depending on its position in the network. The minimum number is 4 and the maximum
possible number of ports is 9. In this example, the priorities of layers are \( \text{Layer}_1 > \text{Layer}_2 > \text{Layer}_3 \).

### 9.3.2 Adaptive Congestion-Aware Routing Algorithm

To route packets in the above described multi-layered topology, an adaptive congestion-aware routing algorithm was developed. We start by explaining the routing algorithm without the Congestion Avoidance (CA) feature. In this algorithm, the path with the shortest hop count between a source and destination node is selected. To avoid routing tables which are not scalable, we propose a routing algorithm which computes the path at run time. This makes the implementation of the algorithm flexible and scalable. The routing algorithm at every router forwards the packets on paths which use the least number of hops. To achieve this, a Near Node (NN) function was designed. It determines the nearest node which connects to layers having longer distance links. This NN function is used to compute near nodes at the source and destination nodes. This results in the path between the source and destination nodes split broadly into three sub-paths.

Let us explain this by using an example of a pair of source and destination nodes as \((x_s, y_s)\) and \((x_d, y_d)\) respectively illustrated in (38). The first sub-path is between the source node \((x_s, y_s)\) and the nearest node to the source \((x_{sn}, y_{sn})\). The second sub-path is between the nearest node of the source \((x_{sn}, y_{sn})\) and the nearest node of the destination node \((x_{dn}, y_{dn})\). And finally the third sub-path is between the nearest node of the destination \((x_{dn}, y_{dn})\) and the destination node \((x_d, y_d)\). Within each of the sub-paths, the XY routing algorithm is utilised to travel between nodes. The XY routing algorithm is selected to achieve minimal routing between these nodes and to keep the algorithm simple.

\[
(x_s, y_s) \rightarrow (x_{sn}, y_{sn}) \rightarrow (x_{dn}, y_{dn}) \rightarrow (x_d, y_d)
\]  

(38)

This is a static routing algorithm, where all packets are routed across the shortest hop paths all the time. This can cause congestion around high port routers which connect multiple layers. To deal with this challenge, an adaptive routing with Congestion Avoidance (CA) was developed which gathers traffic information by reading the input buffer status of routers. This information is used to prioritise critical packets to route on paths with a short hop count.

In cases where multiple applications of varying criticality are sharing the routers and links, critical applications are prioritised and assigned the shortest
hop paths at run time. A one-bit field is reserved in the header of each packet to indicate if it is a critical packet or not. In high traffic scenarios, non-critical packets are assigned paths with longer hop counts to decrease influence over critical packets. Thus decreasing the congestion scenarios and improving overall performance.

For evaluation, the routers are designed using System Verilog and results are generated using HDL simulator ModelSim. The results of algorithms with (NN with CA) and without congestion avoidance (NN) are compared against the popular XY routing (XY). An extended XY routing for layered topologies (Layer XY) is also developed for fair comparison. Tests were conducted under uniform traffic scenarios for different packet lengths. Some of the results for packet length 10 are highlighted in Figure 108. Improvement could be observed in latency and throughput for the algorithm with congestion avoidance (NN with CA). With congestion avoidance there is an improvement of up to 16% in throughput. The algorithm with congestion avoidance has a decrease of up to 56% in latency compared to XY routing and decrease of up to 29% in latency when compared against the algorithm without congestion avoidance. It should be noted that the proposed routing algorithm utilises up to 30% of the router resources, such a network is more suitable for large scale NoCs which can benefit from the presence of multiple layers.

### 9.3.3 Dynamic Block-based Multicast Routing

Most NoCs focus on unicast traffic, where communication is between a single source and a single destination. Multicast traffic where communication is between one source and multiple destinations is extensively seen in large-
scale multiprocessor systems. A motivation for the multicast feature includes support of coherency protocols for distributed shared-memory programming. Usage of multiple unicast packets in the above example degrades the performance and is an inefficient use of resources. Such programming models can greatly benefit from efficient hardware-based multicast features in NoCs. Hence, we present a multicast technique which is scalable, with low packet overhead and latency [1].

Figure 109: A 4×4 mesh with 6 destinations in a block defined by DstS and DstE.

It is a block-based multicast routing, where a cluster of potential destination nodes in a block is defined at run time. A bit vector is used to represent the destinations within this block. Dimensions of the block and the bit vector are stored in the multicast packet header. This technique is advantageous when destinations are close together. It provides run-time support for both unicast and multicast routing with few changes and provides flexibility in the platform. When a source needs to send data to a group of destinations, a block is identified which contains all the destination nodes. The block is defined by two node addresses, which represent nodes within the block which are nearest and farthest to the source node.

Let us illustrate the concept of a block using a simple example illustrated in Figure 109 for a 4×4 mesh. Here a source node needs to send a message to 6 nodes highlighted by a rectangular region which we refer to as a block. This block is defined by the source using a Start Destination Node (DstS) and an End Destination Node (DstE). These nodes are present diagonally opposite to each other in this example as shown. A packet is built at the source and forwarded by the routers following XY routing to DstS. At DstS the packet is duplicated and three possible directions are identified: Local (L), Branch (B) and Tree (T). The Local direction forwards a packet to the local port. The Tree direction is part of the path to DstE. The Branch direction branches away from this Tree path. So at DstS, the packet is replicated thrice and forwarded
9.3 Adaptive Multi-Layered NoC and Dynamic Multicast Support

in each of the L, B and T directions. Duplication occurs at each node in the block and the number of duplications depends on if the node is a destination or not. Path taken by the packet within and outside the block is illustrated in the Figure 109. If not all the nodes within the block are destinations, then the Bit Vector (BV) is utilised to denote which nodes are destinations. BV indices are mapped to the nodes within the block. In this example, a 6 bit bit vector is used to represent the destination status of each node. To summarise, the proposed block-based multicast scheme has an optional bit vector, whose length is the total number of nodes in the block. It supports dynamically changing the block at run time when the positions of destinations change.

We evaluated the block-based multicast routing on a Virtex7 FPGA VC707 board and tested it using traffic seen in a fault-tolerant application implementing redundancy. Investigations into how the address overhead (AO) scales when the size of the network increases was also conducted. To present the benefit of the block-based technique; we compared it against all destination encoding where all destination addresses are stored in the packet. We consider 16 potential destinations grouped together in a block of 4×4 in a 2D mesh network. We look at networks of sizes 4×4, 8×8, 16×16, 32×32, 64×64 and 128×128. In Figure 110, the address overhead of multicast packets using all destination encoding and block-based is shown. The address overhead of block-based encoding scales better and has a 79% decrease in overhead for a 128×128 NoC. Depending on the number of destination nodes within a block, a bit vector can be optional. Results with and without bit vector are also shown in the figure for reference.

Figure 110: Address overhead when all destination encoding is used (left) compared with address overhead in BB multicast (right). Both with bit vector(above) and without bit vector(below) are shown for reference.
9.4 Adaptive NoC based on Ant Colony Optimisation

Performance on mixed criticality systems can be improved if the NoC is flexible and can support provisioning and isolation of resources. Techniques like temporal and spatial partitioning of resources are commonly used to ensure that real-time requirements are met. Here we focus on spatial partitioning, particularly on how the NoC in such systems can aid by isolating traffic within these partitions. In this section, we investigate a distributed routing algorithm based on the Ant Colony Optimisation (ACO) metaheuristic [6] to aid in spatial isolation.

![Figure 11: An example illustrating spatial isolation over time on a 4x4 NoC.](image)

We study the potential of such an adaptive routing algorithm to aid in traffic isolation within regular and irregular topologies which can require minimal and non-minimal routes. Let us explain this briefly using an example. An illustration of three applications of varying criticality occupying a 4x4 mesh NoC is shown in Figure 11. Initially a non-critical application is occupying 6 nodes. The routers here have the freedom to utilise all links in the 4x4 region to communicate with each other. Over time, two critical applications arrive. With each arrival, the partition allocated to the non-critical application is changing to avoid influence on the critical applications. From the NoC perspective, the region allocated to the non-critical application vary over time since strict isolation between applications is desired. If the NoC can adapt to the changing partition shape, compute new routes within this varying region at run time, traffic isolation can be achieved. A challenge the NoC faces in such a scenario includes ensuring that all possible partition shapes are routable. This results in the need to discover minimal and non-minimal routes in regular and irregular topologies. To meet these topology-agnostic routing requirements, we propose the ACO-based routing for non-critical application to communicate within its partition shape. While critical applications utilise deterministic routing algorithms like XY routing.
The ability for self-organisation and path-finding of ants and other social insects is the inspiration for ACO [6], which can be used to solve various combinatorial optimisation problems. Inspired by the foraging behaviour of ants, the network is explored with probing packets which discover paths. The information of these new paths is distributed to the routing tables. The routers make decisions based on a combination of long and short term information about the network state. This leads to a self-learning NoC, where the routing is topology agnostic, and the computed paths are optimised for throughput and latency. We present the potential of utilising such a routing algorithm for spatial isolation of resources at run time [2]. The contributions here include design of an adaptive routing algorithm for NoCs based on ACO metaheuristic and adapting the ACO to be suitable for hardware implementation. A hardware solution for run-time computation of paths to aid spatial isolation of traffic is presented. We also demonstrate the capability for discovering minimal and non-minimal routes for partition shapes having regular and irregular topologies.

9.4.1 Distributed Routing based on Ant Colony Optimisation

Now we provide a brief overview of the AntNet algorithm, focusing on how paths are discovered and how the information is stored. The following is based on the improved version AntNet-FA as described in [5].

The following data structures are maintained at each node $k$. Pheromones, which indicate the goodness of a port for a particular destination are organised per node in a matrix $T^k$ as shown in Table 20. Each column contains pheromone values for a certain destination. A pheromone value $\tau_{nd}$ represents the attractiveness of choosing neighbour $n$ when moving towards a particular destination $d$. In the context of AntNet, ants are network packets that travel together with ordinary data packets but are routed according to a special policy. An ant packet starts as a forward ant at some source node $s$ and is sent to a destination node $d$. When it reaches this destination, the ant becomes a back-

<table>
<thead>
<tr>
<th>$T^k$</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>$\tau_{11}$</td>
</tr>
<tr>
<td>$D_2$</td>
<td>$\tau_{12}$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 20: Pheromone table at node $k$.  

265
ward ant and returns to $s$ on the reverse path. Forward ants are periodically generated at every node at run time. The respective destinations are chosen randomly with more weight on those for which the generating node sees more traffic. Forward ants store the address of each visited node in their packet header ($V$). During routing, a neighbour $n \in \mathcal{N}_k$ is selected randomly with the probability $p_{nd}$, which is a combination of the corresponding pheromone value $\tau_{nd}$ and link attractiveness $l_n$. Here we compute link attractiveness based on the available input buffer space in downstream routers. The probability $p_{nd}$ is computed as shown in Eq. (39). The constant $\alpha$ determines the trade-off between exploration of new paths and exploitation of the ones already discovered. When the forward ant arrives at the destination, it becomes a backward ant. The backward ant is then routed according to the path saved in $V$ during the forward run. When a backward ant visits an intermediate node $k$, the pheromone table $\mathcal{T}^k$ values are updated. The pheromone value, which corresponds to the forward ant’s routing decision is increased to reward the newly discovered path. The new pheromone value is given by Eq. (40). Instead of incrementing with a constant value, AntNet uses an adaptive reinforcement factor $r$ which reflects the quality of the path.

$$p_{nd} = \begin{cases} \frac{\tau_{nd} + \alpha l_n}{1 + \alpha (|\mathcal{N}_k| - 1)} & n \in \mathcal{N}_k \setminus V_{s \rightarrow k} \\ 0 & \text{else.} \end{cases}$$  \hspace{1cm} (39)$$

$$\tau'_{fd} = \tau_{fd} + r \cdot (1 - \tau_{fd})$$  \hspace{1cm} (40)$$

Once the backward ant has returned to its source node, the march is completed and the ant is released. Data packets are routed according to probabilities derived from the pheromone table.

In our design the summary of steps for discovering routes within a region is as follows. A forward ant packet which comprises of one flit is injected by each node in the region. If a forward ant successfully reaches the destination, it returns back on the same path and updates the pheromone tables along the way thus distributing the routing information. A path is now said to be successfully discovered between a source and destination pair. If a forward ant is not successful in reaching the destination, it cannot return to the source node. After the waiting period elapses, the node sends another forward ant packet. This repeats until a path is discovered. Ant packets are not further injected after a path is discovered, to avoid adding unnecessary traffic in the
NoC. The learning phase is considered complete when every node has one discovered path to every other node in the region.

We use the iNoC router as the base router in our architecture. XY routing is utilised for packets requiring computation of deterministic routes. The ACO-based routing is designed for non-critical applications. For critical applications, usually the shape of the region they consume is known at design time. When the application starts running, border routers are communicated so a boundary is established. Since these regions can have different topologies, the algorithm is tested and compared against a range of partitions with varying shapes and sizes. A selection of irregular and regular partitions are shown in Figure 112. The results include latency in terms of average hop count, time required to compute all paths in a partition, percentage of ant packets successful and overhead in terms of total ant packets needed. Results for the selected partition shapes are discussed next.

![Figure 112: Selection of irregular and regular topologies where non-minimal paths are required.](image)

![Figure 113: Performance of ant packets for 8 node topologies where non-minimal paths are required.](image)

We tested 8-node networks of different topologies to determine the time it takes to compute the paths by the ACO-based routing. The results are shown in Figure 113. We see that the time required to compute all the paths is the highest for the first partition (8_a) which has a high degree of irregularity since more non-minimal routes need to be computed. The percentage of success is the least for it because it is the least clustered, thus available feasible paths in all directions are limited. Therefore more packets need to be injected to discover feasible paths. For larger network sizes, if the maximum possible number of nodes in a partition is known, then the size of the tables can be set accordingly so as to avoid having all source destination pairs in the table. The
ACO-based router, while highly flexible and can route all partition shapes, does consume more resources compared to XY routing and further investigations are possible.

9.5 Region-based Cache Coherence

Modern computing systems have evolved into tile-based many-core architectures with distributed memories. One of the challenges for such systems is to efficiently support the shared memory programming model. Existing global coherence approaches consume significant overheads which further increase with the size of the many-core system, leading to scalability issues. Re-examining application characteristics reveals that a single workload rarely utilises all processing and memory resources of a many-core system. For example, the work in [19] shows that the performance of most benchmarks saturates when parallelised beyond 16–48 threads. This questions the need for global coherence in many-core systems, motivating alternate coherence schemes.

9.5.1 Concept

We propose Region-Based Cache Coherence (RBCC) [20], a hardware-based coherence mechanism that uses a divide-and-conquer approach to provide scalable and flexible coherence. RBCC establishes inter-tile coherence among a configurable cluster of tiles known as a coherence region. A coherence region is designed with several configurable properties such as the number of constituent tiles, the spatial location of these tiles and the address range for which a coherent view is requested. Furthermore, the RBCC concept allows for several such coherence regions to exist simultaneously. All coherence region attributes can be controlled by the application based on its requirements. Figure 11.4 illustrates how three different coherence regions spread over the prototype architecture, drawn in three different colours.

9.5.2 Features

**Scalability.** By design, RBCC limits inter-tile coherence support to a subset of tiles within a large many-core system. This reduces the width of a typical directory entry and achieves significantly smaller memory overheads compared to global coherence schemes. For example, consider an N tile many-core system with a maximum of $M_{\text{max}}$ tiles making up a single coherence region. If the directory uses $T$ tag bits and $F$ flag bits, the memory savings in the
directory when using RBCC compared to global coherence can be formalised using the below equation:

\[
\text{Directory Reduction (\%)} = \left(1 - \frac{(M_{\text{max}} - 1) + T + F}{(N - 1) + T + F}\right) \times 100
\]  

(41)

Let us assume the following parameters: 32 bit address space, 32K directory sets, 5 bit offset, \(F=2\), \(T = 12 = 32 - \log(32K) - 5\). Compared to global coherence, the directory size can be reduced by 73% for a 64-tile many-core system with a maximum of 8-tiles within a single coherence region. The directory savings increase if \(M_{\text{max}} \ll N\), which translates to several multithreaded applications executing simultaneously within small-to-medium sized coherence regions of a large many-core system.

**Flexibility.** RBCC’s scalable inter-tile coherence quickly turns unappealing if coherence regions can only be configured at design time. This would restrict application execution to certain predefined coherence regions, which may result in under-utilisation of the many-core system. Therefore, the RBCC concept supports flexible coherence regions that can be altered at run time. The idea is to allow applications/users to influence coherence region attributes (shape, size, location, coherent memory range) dynamically at run time. RBCC’s flexibility feature fits well with the resource-aware computing principles of Invasive Computing by providing dynamically resizable X10 places, especially places beyond tile boundaries.
9.5.3 Design and Functionality

The RBCC concept is realised using a Coherence Region Manager (CRM). The CRM is a programmable hardware module instantiated within every tile of the many-core system. Figure 115 represents the CRM and its various sub-modules. Each sub-module has designated tasks, i.e. to setup/dissolve coherence regions, to guarantee a coherent shared memory view, or even re-configure coherence regions at run time. The following sections briefly describe the design and functionality of each sub-module.

**Figure 115:** Overview of the internal structure of the Coherence Region Manager (CRM).

**Snoop Unit.** This is the front-end of the CRM and has two main tasks—listening to memory transactions and maintaining a Coherence Configuration Table (CCT). The CCT keeps track of all active coherence regions and their attributes associated with the local tile. Using the CCT, the CRM’s Snoop Unit listens to incoming memory transactions on the primary AHB bus. It then classifies these transactions into commands that are sent to the Management Unit using a FIFO.

**Management Unit.** This is the back-end of the CRM whose primary task is to interpret the commands received from the Snoop Unit and translate them into coherence actions. The coherence actions can be briefly classified into four categories:

- Directory Update: Track remote sharer tiles of a given memory block
- Write-back: Update the main memory with the latest cache data
- Generate Invalidations: Send-out invalidation messages to remote tiles
9.5 Region-based Cache Coherence

- Execute Invalidations: Invalidate a memory block of the local caches

Configuration Unit. This sub-module is responsible for all run-time coherence region configurations and reconfigurations. The Configuration Unit has access to the Snoop Unit’s CCT and the Directory’s sharer information. Additionally, it contains a request FIFO and a response FIFO in order to support multiple coherence region configuration requests. Its tasks include setting up a coherence region, if it is a first-time configuration, else to trigger a reconfiguration. In the case of coherence region reconfigurations, the Configuration Unit selectively clears the Directory entries to execute requests for updating coherence regions.

FIFO Interface. This sub-module acts as a communication interface between the Snoop Unit and the Management Unit. The FIFO is also used to buffer transactions and is designed with run-time controllable fill level indicators to avoid overflows.

Directory. This sub-module holds the necessary book-keeping information (bit-vector) per memory block which is required for inter-tile coherence. The Directory sub-module is designed as a sparse memory structure to minimise hardware overheads. The Directory’s storage requirement is further reduced owing to the RBCC concept, making it substantially smaller and scalable compared to global coherence directories.

9.5.4 Evaluation

The CRM has been integrated as part of a 4×4 tile-based many-core system and synthesised on the FPGA prototyping platform described in Chapter 15. The RBCC concept is evaluated using the feature extraction task of a video streaming application that is described in Chapter 13. The application supports both, the shared memory programming model (rbcc mode) and an MPI-based programming model (mp mode), making it an ideal use case for RBCC evaluations. In the rbcc mode, the application relies on the CRM to ensure inter-tile coherence. In the mp mode, the CRM is disabled and the application uses explicit software messages for all communication. Retro video game clips (inherently low resolution) are used as inputs to the feature extracting task so as to not overload the hardware prototyping platform. The application is executed first on the four corner tiles of the many-core system. The remaining tiles are used to generate background traffic for further investigations.

Figure 116 shows the execution time per frame of both rbcc and mp modes for two video clips (Donkey Kong and Snake) on the FPGA prototype system. The second half of the graphs illustrate the execution time if background traffic is
on the NoC. For both clips, the rbcc mode clearly outperforms the mp mode by up to 42%. The injection of background traffic introduces spikes in the execution times for both modes. However, the rbcc mode still outperforms the mp mode by up to 35%.

The video streaming application also exhibits malleable properties allowing it to expand/shrink or relocate to different tiles at run time. This again makes it an ideal use case to evaluate RBCC’s run-time flexibility features. Two scenarios, both of which are applicable within invasive computing paradigm are used to investigate RBCC’s run-time reconfiguration properties. For both scenarios, the CRM’s Configuration Unit adapts the CCTs and clears the necessary Directory entries at run time, making sure that the application can resume safely.

**Expanding the Coherence Region.** Figure 117 demonstrates the application benefiting from more computing resources, where the coherence region is correspondingly expanded on application request. Results show that the application’s execution time reduces by up 47% owing to these dynamic expansions.

**Relocating the Coherence Region.** Figure 118 illustrates how the application benefits from relocating the coherence region. Initially, the application is executed on the corner tiles in the presence of background traffic. Relocating to a cluster configuration consisting of adjacent tiles minimises inter-tile communication latency, thereby improving the application’s execution time. The coherence region is correspondingly adapted to the application’s new location, thereby reducing the execution time by up to 21%.
9.6 Near-Memory Computing

The problem of missing data to task locality cannot be neglected in modern computer architectures. Even a continuous improvement of data caching cannot fully bridge the widening gap between processor speed and memory performance. As an alternative to modern computer architectures, near or even in memory computing can be a promising solution to improve data to task locality in modern systems.

Many NoC-based many-core systems like the invasive architecture still provide one or a few global SDRAM memory modules and therefore can profit from near memory acceleration. The PGAS (partitioned global address space)
programming paradigm is a promising alternative to shared memory pro-
gramming if no global cache coherence is provided by the hardware memory
system like in [4] or [7]. It divides the global memory into disjoint partitions
and assigns them to each X10 place, e.g. a tile, as used in invasive program-
ming. In this case, inter-tile communication requires a message-passing-based
approach to transfer data between different memory partitions. In order to
accelerate the transfer of message passing buffers, these systems often provide
a hardware DMA engine that efficiently copies a buffer from one memory par-
tition into another. However, the prevalence of pointer-based data structures
implies the necessity to serialise data structures before copying them. After the
transfer, the receiver has to deserialise them again. Copying object-oriented
data structures directly via a hardware DMA unit without serialisation would
result in incorrect pointer addresses.

Sophisticated software approaches meanwhile tackle the serialisation over-
head and copy data directly via remote load-stores. Figure 119, which is based
on the concept described in [15], shows such a mechanism schematically.

![Figure 119: Logical sequence of interactions for advanced Message Passing approaches without graph serialisation.](image)

In order to transfer a graph from one memory partition to another one, the
following steps are performed: 1.) The sending CPU has to issue a cache-
writeback of the whole graph. However, in order to find all objects and to
derive their size, a whole graph traversal is performed during this step as well.
2.) After the graph writeback operation has been completed, the sending
CPU signals a remote CPU to read the graph once. 3.) Then, the receiving
CPU loads the graph remotely via the NoC and its local L2 Cache, creates a
duplicated graph and stores it in its own memory partition. It thereby allocates
for each object an individual memory buffer.

Although this strategy outperforms conventional message-passing approaches
which require serialisation of the data structures, it still suffers from a missing
9.6 Near-Memory Computing

hardware support of the DMA engine. Moreover, the remote execution of the graph copy operation, which is performed “far-from-memory”, leads to an unintended cache pollution with the original graph as well as to a significant NoC utilisation.

9.6.1 Concept of Near-Memory Graph Copy

With distributed, object-oriented data structures, DMA-like hardware engines are not applicable in many cases. We thus provide a near-memory accelerator (NMA) that efficiently transfers arbitrary object graphs between memory partitions in tile-based MPSoCs and thereby avoids the necessity of costly serialisation of all objects [16]. This NMA is accompanied by a tightly coupled near-cache accelerator (NCA) for the graph traversal and write-back per tile. Figure 120 shows the individual steps of this approach applied to a 2×2 prototype architecture, as illustrated in Figure 121.

1.) Instead of traversing the graph itself, the sending CPU triggers the NCA which
2.) performs a cache-writeback of the whole graph. It additionally creates a list of objects and their size within a predefined memory buffer.
3.) After the writeback has been finished, the receiving core is triggered. It uses the list of objects to allocate individual memory buffers and appends the associated pointers to the object list.
4.) It then uses the NCA to invalidate all cache lines that will be overwritten by the NMA.
5.) After the NCA has finished, it subsequently triggers the NMA to perform the graph copy task and thereby hands over the list of preallocated memory buffers to the NMA.
6.) The accelerator performs the graph copy asynchronously and adjusts all pointers “on the fly” in order to create an independent copy. After completion, it signals via an i-let the completion of the graph copy task.
7.) Finally, the graph is read and processed by the receiving CPU. Despite the increased number of steps involved, the overall performance benefits heavily from offloading major steps into dedicated hardware accelerators.

Figure 121 illustrates how the novel hardware components integrate into the system architecture. NCAs are present in each tile close to the L2 last-level cache. Like the L2 cache, they are shared between all cores within this tile. In contrast, the NMA is only instantiated in the memory tile and handles requests from all cores in a serial manner. A FIFO in front of the NMA buffers incoming requests in case the NMA is currently busy.

Even though the design of dedicated hardware accelerators is naturally very task-specific, our approach is not limited to a narrow range of applications. We integrated it directly into the system software in order to accelerate the PGAS-specific remote-procedure-call (RPC) via an i-let which performs graph
copies in order to offload tasks to another place. Thus, any application that uses the PGAS RPC mechanism benefits from a hardware-accelerated system software and the concept is fully transparent to the application code.

### 9.6.2 Implementation

Having described the hardware software co-design approach conceptually, we now provide a detailed description of implementation aspects.

**Object Model** Our simple object model supports the X10 language for invasive programming in our prototype system. All objects can be divided into an object-specific header followed by a payload section of variable size. The
payload section can contain pieces of primitive data, pointers to other objects, or array-descriptors in an arbitrary order. The latter consists of two words, of which the first one points to the array's backing-store which holds the data, and the second one stores the number of elements in the array. In order to support the accelerator, we extended the array-descriptor by a third word that holds the size of each array element. Our object model allows both arrays of primitive data and arrays of object-pointers. Furthermore, the object model was extended by 4 words of scratch space within the object header. These are used by the hardware module during the graph traversal. Our approach supports arbitrary object graphs including both cyclic and acyclic structures.

**Object List**  In order to support garbage-collector compatibility, the near-memory accelerator copies all objects into individual, preallocated buffers of correct size. This concept is in contrast to [16] which copied all objects together into one buffer. We use an object list to transfer information on the object-sizes and corresponding pointers between the cores and the hardware accelerators.

When calling the near-cache accelerator to perform the graph traversal and writeback, the operating system hands over a pointer to the object list to the NCA. During the graph traversal, the NCA stores the object-pointers of the source-graph in the first half of the object list and the object-sizes in the second half. It also performs a cache-writeback of the whole object list before completion.

The receiving tile thus can read this information. It allocates the corresponding buffers for the copied objects in its own memory partition and stores these pointers in the second half of the object list by overwriting the size information. This updated version of the object list gets then again forwarded to the near-memory graph copy unit. It creates an independent copy of the original graph, stores each object in the corresponding buffers and adjusts all pointers automatically.

### 9.6.3 Evaluation

We evaluated the effectiveness of the whole hardware/software co-design approach with the IMSuite [9] benchmarks, which are designed to exploit the features of PGAS and X10. The IMSuite is a collection of classical distributed algorithm kernels which are written in a way that they distribute data amongst several places, compute on it, and finally gather and verify the results. For an evaluation of the novel hardware components, we divided the total execution time of the benchmarks into two disjoint parts: the computation time per-
forming arithmetic algorithms and the communication time, which denotes the duration of inter-tile-communication routines.

Figure 122 compares the communication time with and without the hardware graphcopy accelerator. In the smaller 2×2 design, the approach is able to reduce the communication time by 86% in the BF, DST and VC benchmarks, whereas the KC and LCR benchmarks show a reduction in communication time of at least 49%. Also in the 4×4 design, the NMA is able to reduce the communication time up to 87% in some benchmarks. However, the KC benchmark is different, since the communication time is even increased compared to a pure software approach. An analysis of the data structures showed very small graph sizes, thus the base overhead of the operating system cannot be outperformed in this particular case. On average, the communication time can be reduced by 68% in the 2×2 design and by 48% in the 4×4 design. The larger configuration in general performs more graph copy operations between tiles, thus more tasks are offloaded simultaneously to one NMA and the performance benefits are slightly decreased compared to a smaller setup.

![Communication time measurements of the IMSuite benchmarks with and without NMA and NCA, normalised to the variant without hardware acceleration.](image)

A comparison of the overall benchmark run-time with and without the novel hardware components is displayed in Figure 123 for the 2×2 architecture and for the 4×4 architecture, respectively. In the 2×2 variant, the evaluation showed a reduction of the total execution time, i.e. the time within the benchmarks region-of-interest, compared to a state-of-the-art software approach in every
Figure 123: Run-time measurements of the IMSuite benchmarks with and without NMA and NCA, normalised to the variant without hardware acceleration.

case. The speedup is between 5% in the LCR benchmark and 41% in the BF benchmark. Also in the 4×4 design, most of the benchmarks benefit heavily from the hardware accelerator, only three benchmarks are slightly decelerated due to their very small object sizes and the thus relatively large operating system overhead. On average, the benchmarks run 24% faster in the 2×2 design and 35% in the 4×4 design.

9.7 Conclusion and Acknowledgement

The previous sections presented the contributions to support invasive computing via the communication infrastructure of many-core architectures, considering both the on-chip interconnect and the memory subsystem. First the basic interconnect with invadable communication resources, which enable guaranteed bandwidth services among tiles of the invasive compute architecture was described. Further, various approaches to optimise resilience, latency and efficiency of the communication architecture were introduced. As a second key aspect, enhancements in the memory subsystem of the tile-based invasive compute architecture have been introduced, one for extending X10 places dynamically beyond single tiles and the second for near memory acceleration of graph copy operations that support the issuing of $i$-lets on remote tiles. In general, the activities described in this chapter complemented the
overall efforts to build an efficient compute architecture tailored to support invasive computing.

We want to acknowledge the valuable contributions by Stephanie Friederich, Jan Heisswolf, Leonard Masing, Sven Rheindt and Aurang Zaib.

References


C: Compiler, Simulation and Run-Time Support
10 Invasive Run-Time Support System (iRTSS)

Lars Bauer, Jörg Henkel, Timo Höning, Wolfgang Schröder-Preikschat, Christian Eichler, Jefterson Gonzalez, Benedict Herzog, Tobias Langer, Sebastian Maier, Jonas Rabenstein, Phillip Raffeck, Florian Schmaus

Abstract Invasive programs, as well as hardware designed specifically to run them, need an operating system that does not make applications pay for resources, functions or features that are not needed in operation. With this in mind, we provide in the following methods, principles and abstractions for the application-aware extension, configuration and adaptation of invasive computing systems. These are technically integrated into an invasive run-time support system (iRTSS), a highly scalable heterogeneous operating system that comes in various forms and ports (LEON, x86-64, AMD64, ARM64) depending on the use case. The system addresses special-purpose MPSoC-based as well as general-purpose multicore/manycore machines.

10.1 Introduction

Invasive computing primarily means self-adaptive and resource-aware operation of the computing system. The operating-system support must be designed accordingly. This means, among other things, the provision of the following functions, divided here into three complexes: (set basics) low-latency system operation, highly scalable resource management, application-specific and problem-oriented customised system functions; (facilitate prediction) on-demand partial virtualisation and devirtualisation of selected computing resources, dynamic reconfiguration of the operating system, strategic control according to self-adaptivity and decentralised resource management; (enforce properties) accomplishment of required quality criteria, identification of the performance corridors of resource allocation, transparent access to the different main-memory subsystems, control of background noise (i.e. indirect overhead), and externalisation of system parameters (as to timing and energy consumption) in order to be able to give a precise prognosis on the effectiveness of a particular resource allocation decision of the operating system. In the following description, the focus is on the latter complex, namely operating-system measures for feature enforcement.
Figure 124: iRTSS architecture on a multi-tile system. Colours indicate invaded resources.

Figure 124 provides a high-level view of the current iRTSS architecture. Key elements are OctoPOS\(^1\), the parallel operating system (POS) that implements the *mechanisms* of iRTSS to make all capabilities of the underlying hardware available to higher (software) levels, and the agent system, which provides global iRTSS *strategies* for resource management through means of self-adaptation to cope with the scalability problem in large multicore systems, logically residing between the run-time libraries for various kind of invasive-parallel applications and the OctoPOS kernel. OctoPOS makes the computing platform accessible to processes as a “virtual machine” that functions either as native (“satellite”) neighbouring to or as guest (“planet”) managed by a host operating system (Linux). The key aspect in the design and development of OctoPOS is to make all the capabilities of the underlying hardware available to higher (software) levels in an *unfiltered* way.

### 10.2 OctoPOS

In terms of architecture and construction, OctoPOS belongs to the so-called *multi-kernel-based* [2] operating systems. Each tile of an MPSoC specialised in invasive computing is equipped with its own OctoPOS copy, but these copies are customised with regard to the respective tile functionality. Tiles are differentiated, for example, in those that are only used for computation, populated by common cores or special processor architectures such as an

---

\(^1\) Prefix ‘Octo’ indicates the 8th generation within a particular family of special-purpose operating systems—but also refers to a nature which is highly parallel in its actions as well as adaptable to its environment: the octopus, being able to act in parallel by means of its tentacles, adapt itself through colour change, and, due to its highly developed nervous system, attune to dynamic environmental conditions and impact.
i-Core (Chapter 5) or a TCPA (Chapter 6), provide central main memory or enable I/O. As a result, OctoPOS functions for operating special processing units, managing central memory or interacting with the outside world of the MPSoC can only be found in the corresponding tiles.

The multi-kernel approach is mainly due to the fact that processor cores of different tiles do not have a cache-coherent view of the common, global main memory. Consequently, the kernel-level interaction between the respective OctoPOS instances takes place through message passing, whereby not only data is transferred here, but also the identity of the execution strand that is supposed to process this data at the remote side. This is similar to active messages [11], but differs in that the execution strand specified in the message is remotely fed to the OctoPOS process scheduling. Within a tile OctoPOS benefits from the existing cache coherence between the processor cores of this tile and operates in this scope according to the principles of a shared memory processor. This makes OctoPOS a hybrid operating system that is at home in both worlds, namely shared and distributed memory systems.

This fact, which has its origin in the computer architecture of the MPSoC relevant here, has a significant impact on the way in which energy-efficient processes are implemented, including occasional isolation in their own address spaces in the main memory. These aspects therefore also form the guideline for the following considerations in this section, starting with memory and address spaces, through processes and threads, to time and energy as the first-class resources in OctoPOS.

### 10.2.1 Memory and Address Spaces

Like ordinary computations, the invasive species to be supported by OctoPOS are unthinkable without an executing program in main memory, namely a process determined by the instructions and data of its program: “a process is that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor.” [4] Depending on the application, the processes operate on the entire (tile-local as well as global) main memory, in a partitioned global address space (PGAS, Section 11.3.2). This type of address space is logically distributed to a collection of execution strands, so called i-lets (see Section 1.2) of these processes. Each of these strands has both an affinity for a portion of the globally shared address space and a private realm.
PGAS identity mapping

Each local main memory of a respective tile as well as the global memory tiles are assigned its own area in the PGAS, which means that all of these memory entities have their own globally unique (logical/virtual) address. However, so that the locally resident and functionally identical OctoPOS instances in each tile have the same view of their own program components, which considerably facilitates their interaction in the distributed operation of the MPSoC, these instances are always located at the same (logical/virtual) address at the beginning of the PGAS: Following the pattern of identity mapping [13], the identity of a single logical OctoPOS entity gets assigned to the different physical OctoPOS instances in the different tiles.

Virtual shared memory

PGAS-accessible main memory is convenient for easily identifying shared data uniquely via a memory address, but this usually does not relieve programs of having to deal with the shallows of distributed memory. The processes are confronted with different memory access latencies, which, if at all, can often only be covered with difficulty; the processes that may be running in different tiles (i.e. partitions) cannot rely on cache coherence, which entails expensive and extensive copying of shared data structures; and the actual degree of strictness of the memory consistency will cast a large shadow on programming. For these reasons, OctoPOS provides an operating-system abstraction with virtual shared memory (VSM) that allows the processes not to have to deal with such problems [23].

VSM is considered a context-sensitive feature of OctoPOS and is implemented on a per-page basis, supported by an appropriate memory management unit (MMU). As with other resource requests, this feature is enabled at the time a certain memory area becomes invaded by an application process. At that moment not only the quality of the memory consistency in the respective area is selected, but also the desired handling of false sharing is determined. In addition to multiple-reader/single-writer semantics per page of the virtual address space [24], OctoPOS supports a multiple-reader/multiple-writer mode of operation [14].

As shown in Figure 125, in this mode changes are made only to the (local) secondary copy of the (local) primary copy from a globally shared page. This applies to each process that accesses the same global page, but through different addresses on objects located on that page: Sharing granularity is the page, not the individual byte. In order to re-establish a globally consistent view, per
Figure 125: VSM in a per-page multiple-reader/multiple-writer mode of operation. Processes $A$ and $B$ are assigned to cores of different tiles and unwittingly share the same PGAS page by accessing different data items on that page (false sharing). Process $A$ writes to the relevant page because it is modifying a data element (upper cell, page copy on write), while process $B$ concurrently reads another but unknowingly neighbouring data element (lower cell, page copy on reference). The page in question is replicated twice and becomes the primary copy in the local memory of the tile of the respective process. A secondary copy of the page is immediately created locally for the writing process $A$. Process $B$ does not create the secondary copy of the page until it later writes to it, if at all. Both processes freely make their changes in the secondary copies of the shared page. Finally, the processes form a change index independently of one another (in parallel) by page-by-page XOR of the primary and secondary copies. As soon as the change index is available, both page copies are deleted. This happens as soon as the processes leave the program sections in which the respective data manipulations take place. The generated change indexes are then also merged one after the other via XOR, resulting in the updated shared page. In this final phase, each of the two processes $A$ and $B$ acts as a merger.

page two exclusive disjunction (XOR) steps are performed in sequence. The first step computes the change index from the primary and secondary page in a local operation, while the second step subsequently merges all change indices in order to consistently restore the original page. Both steps benefit from dedicated hardware support for in-memory processing, if provided (cf. Section 9.6), but are also available in a software variant for reasons of generalisation and for comparison purposes.

By help of the multiple-reader/multiple writer operation mode, the VSM subsystem of OctoPOS furthermore can exploit performance benefits of weaker memory consistency models. In release consistency configuration, for instance, memory consistency is granted at a page granularity basis. Via the acquire and release system calls, user applications can specify their need for the latest data or publish their own changes, respectively. This allows the overall reduction of unnecessary copy operations.

Page duplication always takes place on-demand, using copy on write [30] and copy on reference techniques [47], respectively. This also means that in the case of complex shared (static/dynamic) data structures, only those pages that
actually contain the structure elements referenced by the processes are made available locally on demand. Proactive copying of a complete data structure, in which only individual structural elements are then changed, added, or removed, is therefore not necessary or can be supported in conjunction with the programming language runtime system (cf. Section 11.4.4).

A particularly important point about the VSM concept outlined here is that the measures described do not take place mandatorily in OctoPOS, but appear as an optional feature of the operating system. VSM is activated and deactivated when processes enter and exit appropriately marked program sections. On the one hand, when main memory is invaded, relevant pages can be announced as VSM-compatible, on the other hand, individual program sections can be designed to be VSM-based. Both must be explicitly expressed in the application programs either parametrically or declaratively, either manually by the programmer, as is currently the case, or automatically by the compiler. In this way, the overhead in the operating system and the background noise it generates can be controlled in an application-oriented manner in line with the resource-aware programming paradigm of invasive computing.

**Adaptive memory protection**

The same with regard to the problem-oriented control of system functions that is possible on the application level applies in principle to the address space management that OctoPOS implements, not only to the handling of data stored and shared in the main memory. Usually, hardware-based memory protection is widely used in many areas of data processing and is a fundamental building block for security and protection (see also Chapter 12). However, improving protection measures by means of dedicated hardware like an MMU does not have to be seen as set in stone, but rather depends on the application domain and the facts of the programming system and hardware, respectively. State-of-the-art operating systems statically determine whether or not software entities are subject to memory protection. Costly operations will follow due to multi-level page-table handling, TLB (translation look-aside buffer) invalidations, and the release of inter-processor interrupts. This all reduces performance, increases operating-system noise, and makes system behaviour unpredictable. If application programs are type-safe, then MMU-based memory protection becomes a pretty much superfluous feature—as long as fault tolerance is neglected. Contrariwise, if the programs are type-unsafe, the lack of protection of this type opens door and gate for malware.

A way out of this dilemma is adaptive memory protection that is capable of dynamically changing the per-program protection state [8]. If statically
configured in OctoPOS, this feature is triggered at load/unload time of application programs and applies in the background of running processes. If it is known in advance that the MPSoC will exclusively host processes of type-safe programs (e.g. X10), MMU-based memory protection is statically configured out from OctoPOS. If this a priori knowledge is not available or one has to assume that processes of type-unsafe programs (e.g. C/C++) are also fed into the system, MMU-based memory protection is statically configured in, but initially switched off by OctoPOS. In this configuration, only processes of type-unsafe programs are physically (MMU-based) enclosed in their address spaces and thus isolated from other processes in the MPSoC. Processes of type-safe programs can continue to access the main memory of the MPSoC unhindered, their respective reference string (i.e. the sequence of per-process consecutively generated main-memory addresses) is considered correct or harmless by the code generated by the compiler and, thus, needs not be subject to any monitoring by the MMU.

This approach allows the overhead and background noise, respectively, inherent in an operating system related to memory protection to be controlled at the application level. One immediately apparent source of overhead is system calls. Memory protection requires a differentiation between user and system privileges, and user-level (unprivileged) processes must be denied direct access to the page tables and MMU registers managed by the operating system. But another source is responsible for the much larger overhead, namely the TLB. This primarily applies to scenarios with multithreaded processes of non-sequential programs that are to be executed on a multicore/manycore processor, which forms the basis of every MPSoC. TLB entries must be invalidated or even the TLB completely emptied if such a process makes changes to its address space, for example requesting or freeing main memory, which means that pages are generally to be mapped or unmapped by the operating system. In such a case, the operating system must not only clear the respective page descriptor from the page table and update the TLB of its core—which is the core of the thread that causes the address space change of its process—but also from the respective TLB of all other cores hosting a thread of that very process (TLB shootdown [3]). Not only does this measure provoke subsequent TLB misses, which requires a (hardware or software-managed) page-table walk to find the page descriptor to be cached, but also that an IPI (inter-processor interrupt) must be sent to each affected processor core so that its TLB content is kept consistent as part of the operating system’s interrupt-handling procedure. These are all measures that must be taken into account with all of today’s processors—and which also apply in OctoPOS if memory protection is to be provided for an application.
10.2.2 Processes and Threads

Address space isolation costs. These costs must be accepted if the operating-system features for the physical containment of processes are essential for the given use case. However, if the use case does not require these features, there is no technical reason to slow down, disturb, or handicap processes or make them difficult to predict at all. The same applies to the measures to bring a program into execution.

Versatile execution strand

A basic principle of invasive computing is the constrained exclusive use of resources. This primarily affects the processor cores required and available for executing the (especially non-sequential) application programs. The basic component of invasive programs, as far as the linguistic representation of a computing unit is concerned, is the \textit{i}-let—within an invasive program, a basic block intended for autonomous execution on a processor core (see also [41]).

OctoPOS represents an \textit{i}-let as a tuple of an instruction pointer and a data pointer, the former addresses the first machine instruction in that basic block to be executed by a processor core and the latter identifies a problem-specific data area [26]. Basically, an \textit{i}-let is a coroutine that shares the same runtime stack with other instances of its kind. Apart from preemptive procedures for stack-based task scheduling [1] or interrupt-prioritised threading [17], with which a processor core can be efficiently multiplexed under system control even with a single-stack solution typical of an event-based operating system like OctoPOS, \textit{i}-lets must be subjected to cooperative processing.

Thus, \textit{run to completion}—of the current CPU burst—is usually the execution principle followed by OctoPOS for \textit{i}-lets. End of current CPU burst means either termination or blocking of an \textit{i}-let, the latter because it cooperatively released the processor core while waiting passively for a certain event to occur. In such a situation, provided system constraints allow, OctoPOS will allocate another ready-to-run \textit{i}-let to the vacated processor core, if available. Upon event occurrence, the waiting \textit{i}-let is unblocked and added to a ready list sorted by arrival time (FCFS) if the relevant core is occupied by another \textit{i}-let and no other core of the processor is free or allowed to be used.

The main motivation for this very simple kernel-level representation and processing of an execution strand is the comparatively large number of processor cores that can be expected from an MPSoC for invasive computing. Consequence of this is that single-threaded cores will be the normality, multi-threaded ones the exceptional case [26]. These strands not only serve to...
implement user-level i-lets, but also contribute, within OctoPOS, as so-called “system i-let” to the efficient execution of special kernel-level system operations in a variety of ways: such as writing back unused VSM pages in the background, post-processing of inter-tile data transfers on the send and receive side, handling of inter-kernel remote procedure calls analogous to active messages [11], or as a basis for run-time requirement enforcement (RRE [40], cf. Chapter 2).

The latter regarding RRE is based on an i-let instance that ensures compliance with certain quality characteristics for a computation to be carried out within an invasive program. This special i-let serves as aide for just that i-let that acts as a lead within the invasive program for which certain quality features are to be enforced. Both control-flow entities together form a so called squad,\(^2\) which then is handled by OctoPOS as a single logical unit of processing (see also [34]). In this setting, the aide is closely tied to OctoPOS events via a system i-let. These events are generated by OctoPOS itself or signalled by the hardware and then forwarded by OctoPOS to user level. As i-lets can be pinned to different/own processor cores, aide-based (commonly user-level) event handling then is rendered possible without the need for multiplexing the core that runs the (always user-level) lead.

**Scalable concurrency control**

Cooperative i-let scheduling does not relieve OctoPOS of synchronising simultaneous processes when accessing shared resources or running program sections which, for correct operation, respectively can only be of sequential nature. Of course, the same also applies to the invasively parallel program executions operated by OctoPOS. In principle, there is a need for explicit coordination of the cooperation and competition of simultaneous processes, both “vertically” due to (possibly nested) interrupt requests from the periphery or asynchronous upcalls from lower system-software levels (pseudo-parallelism) and “horizontally” due to actions of a non-sequential program executing on multiple processor cores (true parallelism). Depending on the use case, the same or different synchronisation paradigms can or must be used, that is to say, non-blocking (lock/wait-free) or blocking methods.

Wherever applicable, OctoPOS practices *non-blocking synchronisation*, with lock-free or wait-free progress guarantee [15], depending on the problem. Especially the simpler lock-free, but also and preferably the more complex

\(^2\) Borrowing from hip-hop culture, the term denotes an informal group of individuals characterised by friendship and solidarity.
wait-free approaches are suitable for synchronising dynamic data structures like queues, which are used in many places in system software such as in the implementation of (asynchronous) critical sections [33], signalling of execution strands [12] or work-stealing-type processing of jobs or buffered data [35]. An important point here are constructive measures at the relevant points in the system software, which then no longer result in random overlapping patterns of simultaneous processes and thus facilitate non-blocking synchronisation—or make synchronisation unnecessary at all.

However, blocking synchronisation is also an issue in OctoPOS, but here it is mainly due to application requirements for system functions to be provided. Thus, private semaphores [6] are made available, the implementation of which is not itself designed as a conventional critical section and instead corresponds to a non-sequential basic block that is non-blockingly synchronised. Similarly, asynchronous critical sections, so called guarded sections [9], or reconfigurable locks [31] emerged that are exposed when needed.

**Execution-strand migration**

Synchronisation of simultaneous processes, when the processor executing the non-sequential program has more than one core, can also be achieved through process migration [32]. The basic idea behind this approach, migration-based synchronisation (MBS), is that control flows are migrated to data, instead of moving data to control flows. The consequence is an improvement of data locality that reduces the worst-case execution time (WCET) of critical sections, and indirectly, worst-case blocking bounds. Unlike remote core locking (RCL [25]), which pioneered this approach, MBS does not require any changes to the program whose processes are to be migrated at critical sections.

MBS can be generalised by means of static analysis. Independently of critical sections of a program, this method automatically identifies exactly those points where migration of an execution strand can be advantageous [29]. The analysis considers the overhead of the respective migration measure (e.g. expected timing behaviour [36, 37] or energy consumption [44]) and the resulting overall balance of the program to be executed under real-time conditions.

Migration is not only done for reasons of improved resource-efficient operation of OctoPOS, but also to allow OctoPOS to react to environmental changes and enforce non-functional properties such as temperature and energy demand or communication latencies [28]. However, this presupposes that the exclusive claim of an i-let to a compute core does not refer to the physical but to the
virtualised resource. This form of core virtualisation then basically allows the migration of an i-let as an enforcement reaction. In the case of thermal side-channel detection (cf. Section 12.7), the mechanism can be used to counteract security concerns: Thermally conspicuous cores are cleared by migrating the execution strands on them, thereby disrupting or preventing undesired information flows.

10.2.3 Time and Energy

Resource awareness goes beyond knowledge of the currently available processor and memory units: the execution of any application and any operating-system routine comes at costs regarding time and energy. Predictability and enforcement of restrictions of such non-functional properties in combination with the adaptation of the system’s behaviour to environmental conditions (e.g. available power budget, ambient temperature) are important properties for iRTSS in particular and invasive computing in general.

Monitoring and measurement

A prerequisite for resource awareness are means to monitor and measure actual resource consumption at run time. For many resources (e.g. allocated memory, processor cores), this data is inherently available within the system software. For non-functional resources like time and energy, however, internal or external tools are required to determine the current resource allocation. Tools for time measurements with different precision levels are usually already embedded into hardware (e.g. hardware timers, timestamp counters). However, managing energy as a resource challenges the system software due to the unavailability of measuring points at the system level. One interface available for Intel and recent AMD processors is the running average power limit (RAPL) interface. RAPL offers precise energy measurements at the processor level, which OctoPOS makes available within iRTSS, thus enabling online energy-monitoring capabilities via system i-lets. These capabilities form the basis for global resource-enforcement strategies as implemented by the agent system. For whole-system measurements (instead of or in addition to processor-level measurements) and hardware platforms without internal measurement capabilities, either external energy measurement devices or modelling approaches are utilised.

Energy models

Modelling the energy demand for a hardware platform utilises either a white-box or a black-box approach. For the LEON-based hardware, a white-box
Invasive Run-Time Support System (iRTSS) approach is feasible as detailed hardware models are available (cf. Chapter 7 and Chapter 8). For other hardware platforms, black-box models, for example, based on integer linear programming (ILP) [38] and feed-forward neural networks (FFNNs) [19] are used. Provided with sufficient training data gathered by the aforementioned methods, both types of energy models are capable of precise energy-demand estimations and can substitute the measurement-based energy-monitoring techniques.

**Enforcement and utilisation**

Building upon the monitoring capabilities, iRTSS supports resource-efficient execution and allows to enforce resource guarantees making it truly resource-aware [40]. For example, monitoring in combination with the DVFS implementation of OctoPOS allows the enforcement of power corridors, where the execution environment is dynamically adapted to stay within a lower and upper power limit (cf. Chapter 14). In general, iRTSS utilises statically [39] and dynamically [27] derived information to enforce limitations for applications with little to no system noise (cf. Section 10.2.2).

Besides enforcement, the inherent resource awareness of iRTSS also leads to predictable (in terms of energy and time) system components [10], for example, VSM with known non-functional properties (i.e. energetic VSM). Predictability in this context means detailed knowledge about the best-, average- and worst-case energy and execution-time demand. Besides predictable system components, this knowledge is also utilised, for example, to selectively enable or disable hardware features (e.g. hardware security mitigations [16]), depending on an application’s demands, for a resource-efficient execution.

### 10.2.4 Evaluation of the Concepts

Developing applications for highly parallel MPSoC platforms is a demanding task—a common concept is the fork/join parallelism (“spawn/sync” in Cilk terms). Our wait-free “Nowa” approach to orchestrating fork/join concurrency [35] allows to develop fast, parallel applications that outperform established concurrency platforms like OpenMP. The evaluation with up to 256 hardware threads shows that our approach achieves, on average, a higher speedup than the other, mostly-locking counterparts OpenMP, Fibril, Cilk Plus, and Intel TBB. Figure 126 illustrates this comparison for the matmul and nqueens benchmarks. Like many other considered benchmarks, our approach achieves the highest speedup of all compared concurrency platforms in those two benchmarks, irregardless of the number of threads.
10.2 OctoPOS

![Comparison of Nowa, our wait-free continuation-stealing approach, against other concurrency platforms.](image1)

![Virtual shared memory for non-cache-coherent platforms.](image2)

VSM can reduce the programming effort required to develop parallel applications for cache-incoherent MPSoC platforms. Figure 127 compares the bandwidth provided by the VSM to the system’s L2 cache: Once the size of the working set surpasses the size of the L2 cache (left vertical line), the VSM provides superior performance. Once the working-set size exceeds the size of the VSM’s page cache (right vertical line), additional copy operations lead to increasing overhead and thus declining performance.

The enforcement techniques implemented in OctoPOS can be employed in different scenarios, ranging from handling limited power supplies to thermal management. Our system uses energy awareness for scheduling purposes to optimise the accumulated electricity costs while still retaining the required quality-of-service level [18]. The evaluation (cf. Figure 128) shows that our system reduces the overall energy costs compared a state-of-the-art HPC workload manager by (a) shifting computations to times when energy prices are low and (b) exploiting heterogeneity properties of the hardware (i.e. execution units).
10.3 Agent System

The task of the iRTSS Agent System (AS) is to provide a decentralised and scalable resource management for invasive applications. It manages the claims, the configuration of the underlying hardware, and it implements the invade and retreat functions that are used by applications to express their resource demands according to Chapter 1. The AS allows the applications to specify constraints (e.g. the number of regular cores) as parameters for the invade calls. The agent of an application then negotiates with other agents for resources to optimise the performance of their applications.

The AS was designed to allow an efficient binding to the InvadeX10 language and thus the coexistence of C/C++ and InvadeX10 applications. Its internal implementation within iRTSS is tightly coupled to OctoPOS and adopted its event-based operation. The agents heavily rely on the remote procedure call (RPC) paradigm for communication, a functionality provided by OctoPOS with hardware support by the iNoC (see Chapter 9) to achieve a low-latency exchange of information. Once the AS has come to a viable resource-management decision, it uses functions provided by OctoPOS to configure the underlying hard- and software to allow the application to infect its claim.

10.3.1 Expressing Resource Requirements and Scalability by Constraint Systems

We implemented a sophisticated constraint hierarchy that allows the applications to express complex quality-of-service requirements (‘constraints’) and scalability/performance information (‘hints’). An application can express multiple acceptable resource allocations (using the Or constraint) that are each composed of multiple constraints (using the And constraint). For instance, constraints can specify a minimum and maximum amount of cores (PEQuantity), whether it should be regular RISC cores, i-Cores, or TCPAs (PEType), and whether or not the tiles need to be reserved exclusively (Tile-Sharing).
In addition, the ‘hints’ (a special constraint type) allow the application to express how beneficial it would be to obtain certain resources. For instance, the expected performance gain if the application acquires an i-Core instead of a regular RISC core, or how the application performance scales when changing the number of assigned cores with the given bounds (PEQuantity). This information is required to make decisions about which resources are actually given to which application in order to optimise the overall system performance.

For applications that demand guaranteed service, we implemented a special set of constraints, the so-called Actor Constraints. They are used by the ActorX10 applications described in Chapter 3 to express their Operating Points (OPs). Unlike the regular Invasive Constraints (described above) that operate on core level, Actor Constraints operate on tile level, i.e. they specify how many tiles of which type (RISC, i-Core, etc.) are needed. Additionally, they specify communication latency and bandwidth constraints between the individual tiles. In addition to acquiring resources, the AS also has to map the application to the tiles such that latency and bandwidth constraints are fulfilled, which also depends on the status of the iNoC (see Chapter 9), i.e. how many guaranteed channels are still available to reserve a minimum bandwidth between two tiles. A backtracking algorithm is used in the AS to identify a suitable mapping for an acquired set of resources.

10.3.2 Decentralised Agent System (AS) Infrastructure

In order to enable a decentralised and scalable implementation of the AS, a corresponding software infrastructure is needed for basic AS-internal operations, such as finding the agent of an application, finding agents in the near proximity (e.g. as negotiation partners), communication between agents, even if they were migrated to a different tile, etc. To allow for parallel execution and scalability, we implemented lock-free data structures (lists and hash maps) in iRTSS for this infrastructure by using the Compare-and-Swap primitive.

Figure 129 shows the main components of the decentralised AS infrastructure. There is (at least) one agent per application that negotiates the application’s resource demands with other agents in the system. An agent is essentially just code (e.g. the invade call or internal APIs; identical for all agents) along with data (the agent’s individual context) that is stored in the tile-local memory (TLM) of an arbitrary tile (ideally belonging to the claim that is managed by the agent). Agents can be migrated to other tiles and therefore, when an application wants to communicate with its own agent $A$, we first have to find the tile that currently stores the context of $A$. Similarly, if another agent $B$
wants to communicate (via RPCs) with \( A \) to bargain about resources, it also has to find it first.

There is one \texttt{TileManager} per tile that helps in finding agents, by storing information about all agents that are (or recently were) on that tile. Agents that migrate to/from this tile register/unregister to/from the \texttt{TileManager}. Additionally, agents that migrate to other tiles leave a forwarding address behind that is stored in the \texttt{TileManager}. If an agent tries to communicate with an agent that once resided in a particular tile but recently migrated away, then the \texttt{TileManager} might know where that agent migrated to. But only the last few migrations are stored and in case the \texttt{TileManager} does not know about a particular agent, then it has to be searched. In order to increase the scalability for systems with many tiles, we introduced the hierarchy level of clusters. A cluster is a pure software construct that comprises several tiles (the particular number is compiler-time configurable). Each cluster contains one \texttt{ClusterManager} that acts as a local directory service by storing which agents reside in a cluster. This helps searching for a particular agent, instead of querying all tiles individually.

### 10.3.3 Resource Management for Complex Constraints

For complex application resource constraints we developed a distributed constraint solver [45] that is based on the Distributed Constraint Optimization Problem (DCOP). A variety of algorithms exist to solve a DCOP formulation. The \textit{Maximum Gain Message} (MGM) algorithm is the best-suited candidate for our requirements. MGM is an agent-based local search algorithm that is \textit{incomplete} (i.e. it is not guaranteed to find an optimal solution) and that comes with an \textit{anytime property} (i.e. it gradually improves its solution and can
always return the best solution found so far). Starting with MGM as a baseline, we analysed its performance and behaviour when applied to our specific scenario of manycore resource management, and we conceived multiple novel heuristics that improve the execution time and solution quality significantly, altogether building the foundation of our new Resource Management MGM (RESMGM) algorithm [45].

It is executed by every agent involved in a particular resource bargaining decision and communication takes place via asynchronous messages (RPCs) between neighbouring agents, i.e. agents that have a constraint between them. The algorithm uses two types of messages that are iteratively exchanged between neighbouring agents. The ok?-message contains an agent’s current variable assignment (i.e. which resources it aims to use in its claim and the resources that it knows that another agent aims to use). The agents that receive this message add that knowledge to their own variable assignment. After receiving the ok?-messages, an agent uses its local view to calculate its current constraint cost and determines for which assignment of its variables it can minimise it. The agents subsequently communicate their improvements via improve-messages to all their neighbours. To avoid oscillations, only the agent with the maximum improvement among its neighbours is actually allowed to change its variable assignment. In each iteration, the solution quality of the system improves monotonously. The algorithm can be aborted at any time and returns the best solution found so far, or, if in any iteration no agent was able to make an improvement, then the algorithm terminates and each agent involved in the negotiation knows its new claim.

10.3.4 Resource Management for Malleable Applications

Often, applications do not require complex constraints as handled in Section 10.3.3, but their performance scales with increasing number of cores. Applications that can adapt to more or less cores at run time are called malleable. For applications that use the PEQuantity constraint (see Section 10.3.1) and are malleable, we can grow and shrink their claim, depending on how many other applications execute at a certain time and how many resources they request. The application performance typically does not scale linearly, i.e. growing a claim from four to eight cores might bring significant performance advantages, but growing it from 40 to 80 might only provide relatively small improvements. Therefore, it is globally beneficial, if an application that does not scale well passes over some of its cores to an application that scales better. The degree of scalability can be profiled upfront and can be expressed by a
simple application model. In our constraint system, we use Downey curves [7] that characterise the application scalability by using two parameters.

We have developed a bargaining protocol that achieves global resource management by local exchange of information regarding the type of resources, local system health and expected benefits in the global application mapping. Figure 130 shows an example how this protocol is employed to resolve an invade request from application B. First, the agent of application B requests cores in a suitable location of the system. The suitable location is determined by the resource demands (e.g. if the application demands for an $i$-Core, the location should contain an $i$-Core) and monitoring information of the local system health (e.g. the agent would avoid areas that are already overheated). In the second step, the agents belonging to the applications that have cores allocated within the selected location compare their own Downey curve parameter with the ones from application B and locally decide whether the performance gain of re-assigning some cores to application B is larger than their own performance loss when retreating from those cores. Depending on these comparison results, they sent offers to the agent of application B. From all the offered cores, the agent of application B selects the most suitable ones and informs the affected agents. Once all the claims have been adopted accordingly, application B can infect its invaded cores.

For evaluation, we have compared our distributed resource management with an idealistic centralised approach. For a system with 1024 cores running 32 applications, our approach achieves about 84% of the average application speedup while only introducing 1% and 13% of the computation and bandwidth overhead that the centralised approach requires. Note that the introduced overheads were not incorporated in the system performance evaluation, to be able to compare the raw decision quality. The details for our distributed resource management (DistRM) strategy are published in [22].
10.3.5 Adaptive on-the-fly Application Performance Model

The resource management strategies of the agent system rely on on-the-fly performance estimates for different core allocations. As the scalability of an application may change at run time and also depends on the system utilisation, using an offline-profiled estimation may not always be sufficient. Therefore, we have developed an adaptive on-the-fly application performance model that allows to learn and predict the application performance for different core allocations [21]. For every resource allocation used by the application, we monitor the application performance and compare it with the expectation of the performance model (Downey curve). The difference is then used to adapt the parameters of the Downey curve to reflect the observations [21].

To evaluate the adaptability of our performance model, Figure 131 shows a scenario where the number of applications increases abruptly. The adaptation of our proposed adaptive performance model (APM) to the new operating conditions is clearly visible. Extensive evaluations show that the average performance-estimation error is reduced from 14.7% to 4.5%, resulting in high-quality application mappings that are almost always better than the mappings from the state-of-the-art multi-application multi-step (MAMS [46]) application mappings. As baseline, we added a simple adaptive resource management approach to Figure 131 that only considers the number of cores, but ignores their communication distance, i.e. it is topology agnostic (Top.-agnostic). On the other side, we also compared with the very complex mapTG approach [43] that, beside requiring 4 orders of magnitude more time to estimate the application performance compared to our APM (2 ms vs. 0.12 μs) [21], did not lead to better application mappings.

![Figure 131: Mapping efficiency for an abrupt change in the workload when using different application performance models [21].](image-url)
10.3.6 Enforcing Security through Resource Management

We have shown the critical role of resource management mechanisms to enforce computation and communication constraints. Non-functional requirements, such as security, can also be enforced through resource management techniques based on the principles of invasive computing.

In side-channel attacks (SCA), an adversary tries to exploit physical information leakages, such as temperature [5] or timing behaviour [20], to obtain secret information from the victim. Cache-based SCAs are a particular type of attack that exploit the intrinsic timing nature of cache memories to extract critical information (i.e. private keys) from variations of memory access times on cryptographic security applications (from here on called secure applications). For the attack to be successful, the attacker and the victim must share a cache level, e.g. two applications executing on the same tile. Hence, a resource manager such as the agent system can avoid the attack by applying dynamic task migration, moving the secure application from one tile to another, where it will no longer share a cache memory with the attacker.

Although the principle behind migration as a solution to SCAs is simple, identifying the moment when to trigger migrations due to undergoing SCA is a complex problem. To determine the right moment to apply dynamic task migration, we propose a migration function heuristic \( m_f \) which ponders two factors: performance degradation on the secure application and time.

![Figure 132: Effect on the normalised performance (IPC), shared time ratio, and migration function output for AES under the fast, slow and intermediate attack scenarios, launched at \( t = 10\text{ms} \).](image)

When applying this heuristic, we are able to trigger migrations due to slow, intermediate, and fast potential attackers. By migrating the secure application to a different tile this way, we ensure isolation from the attacker and prevent the attack. Figure 132 shows the results of a simulation for a slow, intermediate and fast attacker on a secure application (i.e. AES from OpenSSL 1.1.1k [42]). Figure 132a shows a performance-driven migration, due to the effect of the...
fast attacker on the secure application’s performance (IPC). On the other hand, Figure 132b shows a migration triggered by time, when exceeding a maximum allowed threshold under a slow attack. For an intermediate-strength attack (see Figure 132c) a combined-effect migration occurs by adding the performance and time factors. For all scenarios, as the shared time between the attacker and the secure application is kept in the order of tens of milliseconds before migrating, the attack is avoided.

10.4 Acknowledgement

Other people were involved in the development of iRTSS over the entire project period, without whose commitment the results presented here would not have been possible. We would therefore like to express our great gratitude to: Gabor Drescher, Christoph Erhardt, Sebastian Kobbe, Daniel Lohmann, Benjamin Oechslein, Anuj Pathania, Jens Schedel, Volker Wenzel, and Alexander Würstlein.

References


306


10 Invasive Run-Time Support System (iRTSS)


11 Compilation and Code Generation for Invasive Programs
Gregor Snelting, Jürgen Teich, Andreas Fried, Frank Hannig, Michael Witterauf

Abstract Invasive computing offers many unique benefits to applications and developers alike—reaping these benefits, however, requires to first overcome the challenges posed by the compilation of invasive programs. This chapter summarises these challenges and respective solutions for heterogeneous architectures, in particular tightly-coupled processor arrays (TCPAs) and general-purpose CPUs. Regarding TCPAs, we show that symbolic loop compilation, an approach to compile loops symbolically without knowing the loop bounds and number of available processing elements (PEs) at compile time, offers flexibility unmatched by other approaches, but is required to fully leverage the advantages of invasive computing and its resource awareness. As an example, symbolic loop compilation also enables other fruitful techniques such as run-time requirement enforcement on loops. For general-purpose CPUs, the compiler’s main concern is to provide an efficient implementation of invasive language constructs on top of the mechanisms provided by the operating system. We show that with a carefully integrated software and language design we can meet this requirement with less overhead than in conventional systems. In particular, we consider the challenge of quickly moving data between the memory partitions of tiled processor architectures, both purely in software and with hardware accelerators.

11.1 Introduction
To exploit invasive computing concepts to a full extent, compiler support is paramount to unburden the application developers. We therefore developed a compiler framework for code generation, program transformations, and optimisations for a wide range of heterogeneous invasive architectures, including RISC cores, TCPAs, and i-Core reconfigurable processors. Refer to Figure 133 for an overview of the compiler framework’s flow.

First, given the source code of an application written in X10, using the invasive language primitives invade, infect, and retreat as introduced in Chapter 1, specially annotated loops are extracted, parallelised and compiled onto TCPAs. We call this part of the compiler *LoopInvader* and it employs techniques
for symbolic loop compilation, as presented below. Next, non-loop code is compiled onto the requested target architecture using the code generation framework libFIRM [3] and supported by the so-called invasive X10 runtime, which implements the fundamental building blocks of invasive programming, such as the invade-infect-retreat primitives, see Chapter 1 for introductory details.

For TCPA targets (see Chapter 6), we achieved a breakthrough in symbolic loop compilation techniques [19, 20], enabling the determination of a latency-minimal symbolic mapping of a parametric loop onto a TCPA without knowing the number of PEs (that is, the claim size, see Chapter 1) in advance (see Section 11.2.2). In addition, we developed a method to generate compact PE programs and other required configuration data from this symbolic mapping efficiently at run time (see Section 11.2.3). This is accomplished by frontloading all intractable steps of loop compilation to the compile time, such as scheduling and routing.
Using this compiler framework, we investigated the two intertwining leitmotifs of invasive computing research: predictability and run-time requirement enforcement.

Predictability in the context of invasive computing concerns the determination and guarantee of bounds on non-functional aspects of invasive parallel programs during execution, such as performance, fault tolerance, and security. We investigated the benefits of invasive computing for adaptive and on-demand fault tolerance [24], and, in particular, developed a novel fault tolerance technique tailored to TCPAs to counter the increasing proneness to (bit) errors of modern, complex systems [17]. This technique automatically transforms a given loop to provide a requested level of modular redundancy by exploiting the large number of PEs on a TCPA.

Going beyond predictability, we investigated run-time requirement enforcement (RRE) with the focus on enforcement of multiple non-functional execution qualities such as user-specified performance and energy corridors—that is, to guarantee bounds on them by dynamically reacting to run-time changes in the system instead of “mere” prediction. In particular, we formalised run-time requirement enforcement on loops [26]. Parallelised loops pose unique challenges because of their uncertainty in the input sizes and number of PEs, which significantly effects the non-functional properties of a loop, such as latency.

For general-purpose CPUs, we focused on program optimisation. Within libFIRM, we investigated using program synthesis techniques in compiler construction. Using these, we are able to automatically generate a complete set of optimisation patterns, as well as rulesets for instruction selection\(^1\). In addition, we developed a novel construction algorithm for Static Single Assignment form (SSA form), which is simpler than previous ones, but produces equally good or better code. As a “spin-off” innovation, this algorithm has since been taken up by the V8 JavaScript compiler among others.

In the following, we summarise the highlights of our contributions.

### 11.2 Symbolic Loop Compilation for TCPAs

Figure 134 gives a visual overview of symbolic loop compilation. First, let us summarise the problem symbolic loop compilation solves: Not only are loops parametric by nature, but, in invasive computing, the number of available PEs is also not known at compile time. This is due to both the resource

\(^1\) The translation from compiler intermediate representations to machine languages.
Figure 134: Conceptual overview of symbolic loop compilation. To enable flexible run-time mapping, where the loop bounds and number of available PEs are unknown, symbolic loop compilation decouples the compile-time phase (symbolic mapping) and run-time phase (instantiation) of compiling a loop. This is achieved via a parametric data structure called a symbolic configuration: it is generated by symbolic mapping and then used by the instantiation phase to generate a concrete configuration according to the then known loop bounds and number of PEs. Figure by Michael Witterauf, reprinted from [22, p. 4] under Creative Commons license – CC BY-NC-ND 4.0.

Figure 134 explains the process of symbolic loop compilation. The compilation phase includes the following steps:

1. **Symbolic Configuration Generation**: This step involves generating a symbolic configuration that depends on the unknown loop bounds and the number of available PEs.
2. **Symbolic Mapping**: The symbolic mapping phase is characterized by the generation of a symbolic configuration that is then used by the instantiation phase.
3. **Instantiation**: This phase is executed at run time and determines the actual allocation of PEs based on the known loop bounds and the number of available PEs.

The diagram illustrates the process with the following components:

- **Any available allocation size**: \( R \times C \)
- **Any loop size**: \( N \times M \)
- **1x3 PEs**
- **2x3 PEs**
- **4x3 PEs**

The diagram also illustrates the process with applications App 1: 2x5, App 2: 1x3, and App 3: ?x?.
11.2 Symbolic Loop Compilation for TCPAs

phases [27]: Symbolic mapping (Section 11.2.2) is performed off-line and solves the involved NP-complete problems, generating a symbolic configuration. A symbolic configuration is a compact representation of programs and configuration data parametrised in the loop bounds and number of PEs. At run time, instantiation (Section 11.2.3) is performed once the parameter values are known, and a concrete configuration is generated from the given symbolic configuration.

In the following, we first introduce the loop model our symbolic loop compiler supports, and then the highlights of both symbolic mapping and instantiation.

### 11.2.1 Polyhedral Model

In the polyhedral model, the iteration space $\mathcal{I}$ of an $n$-dimensional for-loop is represented as a subset $\mathcal{I}$ of $\mathbb{Z}^n$, and each iteration is identified by a vector $\bar{t} \in \mathcal{I}$. In particular, our loop model is based on piecewise linear dependence algorithms (PLAs) [21]. A PLA is a set of equations $S_i$ quantified over $\mathcal{I}$ that interrelate the instances of a set $X$ of affinely indexed variables, where the instance of $x \in X$ at index $\bar{t}$ is denoted $x[\bar{t}]$. We assume the form

$$ S_i: x_i[Q_i\bar{t} - \vec{d}_i] = op_i(x_{i,1}[Q_{i,1}\bar{t} - \vec{d}_{i,1}], x_{i,2}[Q_{i,2}\bar{t} - \vec{d}_{i,2}], ...) \text{ if } \bar{t} \in \mathcal{I}_i, \tag{42} $$

where $x_i \in X$ is the variable with instances defined by $S_i$ as the result of operation $op_i$ and $x_{i,j} \in X$ are the variables with instances used by the definition. Each variable is $m_i$-dimensionally indexed using an affine transform given by a matrix $Q \in \mathbb{Z}^{m_i \times n}$ and a vector $\vec{d} \in \mathbb{Z}^{m_i}$. Which instances of $x_i$ are defined by an equation $S_i$ is restricted by its condition space $\mathcal{I}_i \subset \mathbb{Z}^n$. Note that no instance of a variable $x \in X$ may be defined more than once by a PLA

**Example 11.1.** Consider the following artificial, yet illustrative example that writes the first $N$ bits of an integer scalar $in$ into an array $bits$:

```plaintext
for 0 ≤ i < N do  ▶ iteration space is union of all condition spaces
  $S_1: x[i] = in[]$ if $i = 0$ ▶ read scalar input $in$ in first iteration
  $S_2: x[i] = y[i - 1]$ if $i ≥ 1$ ▶ otherwise, use value shifted $i$ times
  $S_3: y[i] = x[i]$ shr 1 ▶ shift right by constant 1 for next iteration
  $S_4: bits[i] = x[i]$ and 1 ▶ extract bit and output into $bits$
```

PLAs prescribe neither place nor time of execution; feasible execution orders are only implied by the dependences between equations. A reduced depend-
Figure 135: The two phases of symbolic loop compilation. While symbolic mapping is performed at compile time, instantiation is performed at run time. Figure adapted from [27].

**Symbolic Mapping**

- Reduced Dependence Graph
- PE Allocation (Tiling)
- Modulo Scheduling
- Register Allocation
- Propagation Channel Routing
- Polyh. Syntax Tree Generation
- Symbolic Configuration

**Instantiation**

- Symbolic Configuration
- Concretization
- Program Instantiation
- Periphery Instantiation
- Interconnect Instantiation
- Concrete Configuration

**Reduced Dependence Graph** (RDG) is a directed graph \((V, E)\) that makes these explicit and is used as the major intermediate representation during symbolic mapping, the compile-time phase.

### 11.2.2 Symbolic Mapping

Figure 135 shows the detailed compile flow of both the symbolic mapping and instantiation phases. Symbolic mapping front-loads the most computation-heavy compilation steps to the compile time and produces a data structure called symbolic configuration. Symbolic mapping consists of the following steps:

1. **Processor allocation** by symbolic tiling [19] and **symbolic modulo scheduling** [25] define a space-time mapping of the loop, assigning each operation a time, a PE, as well as a functional unit and corresponding instruction.

2. **Register allocation** [22] assigns registers for storing and communicating intermediate data in and between PEs, as well as reading and writing input respectively output data.

3. **Routing of propagation channels** [27] allocates routes on the interconnect network for propagating intermediate values according to the data dependences.

4. **Generation of a polyhedral syntax tree** [23]. This data structure is a compact program representation that is parametrised in the loop bounds and number of PEs. Hence, given any valid values of these parameters, the corresponding PE programs can be generated from it.
5. **Compilation of access mapping** collects information about all accesses to external data, be it input or output, in the form of *access mappings*. Especially symbolic modulo scheduling, register allocation, and routing of propagation channels ought to be performed at compile time because they are NP-complete problems, which makes solving them adequately at run time with only limited resources difficult. The final output of symbolic mapping is a symbolic configuration, which retains parameters as symbols, but from which concrete configurations can be instantiated once the parameter values become known.

**Polyhedral syntax trees**

As motivated above, the generation of *compact* PE programs depends both on the loop bounds and the number of allocated PEs. Symbolic modulo scheduling provides the necessary information: selected instruction, their condition spaces, functional unit, and time offset. Only their composition depends on the still unknown parameter values. A *polyhedral syntax tree* [23] represents this information hierarchically such that it is equivalent to the forest of PE program syntax trees over all parameter values. Its building blocks are so-called *fragments*.

**Definition 7** ([23]). A fragment $F$ is any syntactic constituent of a program.

For example, the assembly instruction `addi rd0 rd1 10` can be structured into five fragments: the mnemonic `addi`, the registers `rd0` and `rd1`, the literal `10`, and finally the entire instruction itself. Whether an operation is executed within an iteration $\vec{i}^*$, for example, depends on its condition space. Fragments may thus be iteration-dependent.

**Definition 8** ([23]). A polyhedral fragment $f(\vec{i})$ maps an iteration $\vec{i} \in J$ to a fragment $F$.

We denote specific polyhedral fragments by *polyhedral (fragment name)*, for example polyhedral register or polyhedral instruction. Since fragments are syntactic, representation as a tree is natural.

**Definition 9** ([23, 27]). A polyhedral syntax tree (PST) is a triple $f = (J, a, G)$ of a condition space $J$, a tuple $a$ of attributes, and a set of children $G$, each of which is again a polyhedral syntax tree. To avoid ambiguity, we write $\text{domain}(f)$ for $J$, $\text{attr}(f)$ for $a$, and $\text{children}(f)$ for $G$ of $f$, but we use the term
node for both a polyhedral syntax tree itself and its children. Each node is of one of two types: If \( \text{attr}(f) = (F) \), that is if the tuple only contains a fragment \( F \), then \( f \) is a fragment node. Otherwise, \( f \) is a meta node that stores implementation-specific syntactic meta-information. A node of a polyhedral syntax tree satisfies the following properties regarding its immediate children: All children are of the same type; if the children are fragment nodes, their condition spaces must be disjoint; if it has children, its condition space is the union of its children’s condition spaces; no two children may have the same attribute values. The evaluation \( f(\vec{I}) \) of a polyhedral syntax tree is the sub-tree where all nodes \( g \) with \( \vec{I} \notin \text{domain}(g) \) are removed.

The evaluation of a polyhedral syntax tree at a concrete iteration \( \vec{I} \) results in a syntax tree that represents the sequence of instructions issued in that iteration. Thus, concatenating the instruction sequences in execution order for all \( \vec{I} \) in the iteration space \( I \) yields an unrolled assembly program for the entire loop. This observation serves as the basis for an efficient program generation algorithm as described in [27].

**Example 11.2.** The polyhedral syntax tree \( f \) generated for the running example is shown below, having the condition spaces (in brackets) after tiling only for the leaf nodes (since all other condition spaces are unions of these). Fragment nodes are set in typewriter, meta nodes in italic. For TCPAs, a polyhedral syntax tree has the following semantics: The second level represents the functional unit program \( fu \), the third the time offset \( \tau \), the fourth the instruction \( mnemo \), and the last two the instruction’s operands.

![Diagram of polyhedral syntax tree](image)

The generation of a polyhedral syntax tree from a reduced dependence graph is described in detail in previous work [23].
11.2.3 Instantiation

Instantiation denotes the generation of a **concrete configuration** from a given **symbolic configuration** and an assignment of values to the parameters—in particular the concrete loop bounds and the number of allocated processing elements. It comprises these steps [27]:

1. **Concretisation** substitutes all occurrences of parameters in the symbolic configuration, that is the loop bounds and number of PEs, with their now-known values.

2. **Program instantiation** is the most complex step, influencing the execution time of instantiation the most, and further divided into three sub-steps:
   a) **Control flow analysis** first determines a set of processor classes, that is, partitions the allocated PEs into sets of PEs that execute the same program. Then, for each processor class, control flow graphs representing the individual functional unit programs (compare orthogonal instruction processing in Section 6.3) are generated from the polyhedral syntax tree.
   b) **Control signal allocation** allocates a set of binary control signals that govern the synchronised execution of the individual PE programs.
   c) **Program generation** generates a PE program for each processor class from the individual control flow graphs and control signals.

The processor classes are determined solely from the polyhedral syntax tree and *without* actually generating the programs. This makes the time complexity of program instantiation linearly proportional to the number of processor classes, *not* to the number of PEs. It therefore scales well with increasing TCPA sizes.

3. **Periphery instantiation** generates configuration data for the global controller from the allocated set of control signals. In addition, it also generates configuration data for the I/O buffers and the I/O controller (see Section 6.6.3) to realise the timely loading and storing of input and output data.

4. **Interconnect instantiation** generates the configuration data for the interconnect network that includes both connections for propagating intermediate data, as well as input and output data.
Figure 136: Flow from the symbolic configuration for the running example to a concrete configuration and its relation to the target TCPA’s hardware components. There are two processor classes, where $\mathcal{P}_1$ is mapped to the first two PEs, and $\mathcal{P}_2$ to the third PE within the $1\times3$ allocation. A so-called global controller orchestrates the synchronisation between these programs using the generated control signal, which is propagated along the interconnect using the shown green, dashed connections. The purple, solid connections, on the other hand, show the configured data connections used to propagate intermediate results as well as input and output data. Finally, the buffers are configured according to the access mappings. Figure by Michael Witterauf, reprinted from [22, p. 118] under Creative Commons license – CC BY-NC-ND 4.0.
11.2 Symbolic Loop Compilation for TCPAs

Figure 137: Processor classes visualised. To the left, the distinct condition spaces annotated to the leaves of the polyhedral syntax tree concretised for \( N = 16 \) and \( t = 3 \) PEs are listed. The top visualises the tiled iteration space, where \( k \) represents the tiles and \( j \) the iterations within a tile; due to the non-perfect tiling, the right-most tile is not full. Each colour of the iterations represents a distinct sequence of instructions that is executed in iterations with that colour. In each row, a 1 or 0 denotes whether the iteration in the respective column is part of that condition space. Two PEs are in the same processor class—and thus execute the same program—if they have the same sequence of ones and zeroes, as do the left two PEs. Figure by Michael Witterauf, reprinted from [22, p. 124] under Creative Commons license – CC BY-NC-ND 4.0.

Figure 136 illustrates how these steps transform a given symbolic configuration into a concrete configuration, and how the parts of the concrete configuration map to the TCPA’s hardware components.

The most influential concept for instantiating PE programs is that of processor classes. The set of processor classes can be determined by only looking at the condition spaces annotated to the concretised polyhedral syntax tree, each distinct program required needs to be generated only once. This is illustrated in Figure 137.

For more details and experiments, see [27], where we demonstrated symbolic loop compilation for a set of loop programs from a variety of domains (image/signal processing, linear algebra, machine learning).

11.2.4 Run-time Requirement Enforcement on Loops

Run-time requirement enforcement (see also Chapter 2) is a technique to guarantee that non-functional properties stay within given bounds, also called a requirement. We consider requirements as absolute, interval-constrained bounds on an application’s non-functional properties, similar to [18]. Examples for loop applications include latency requirements (the loop must execute within 20 ms), throughput requirements (the loop must process at least 25
frames per second), and energy requirements (the loop must use at most a certain amount of energy). Formally, we define a requirement on a loop program as follows.

**Definition 10 ([26]).** Let $p$ denote a non-functional property of the execution of a loop program $L$. A requirement $r_p$ on $p$ is an interval or corridor

$$r_p := [p_{\text{lower}}, p_{\text{upper}}].$$

The actual values of a non-functional property $p$ during the execution of a loop usually depend both on the loop bounds $\vec{b} = \vec{u} - \vec{l}$ (upper minus lower bounds\(^3\) in each dimension) and its mapping $m$ onto the target architecture; we emphasise this dependency by writing $p(m, \vec{b})$.

**Definition 11 ([26]).** A loop program mapping $m = (m_1, m_2, \ldots) \in M \subset M_1 \times M_2 \times \ldots$ is a tuple of mapping parameters (such as allocation, schedule, frequency).

**Example 11.3.** For TCPAs, we model a mapping as a tuple $m = (\mu, \hat{R}, \hat{C})$ of a symbolic mapping $\mu \in M_{\text{sym}}$ (as determined by the symbolic mapping phase described in Section 11.2.2), and a PE allocation $\hat{R} \times \hat{C}$.

**Definition 12 ([26]).** A mapping $m$ of a loop program $L$ satisfies a requirement $r_p$ for bounds $\vec{b}$, formally $m \models_{\vec{b}} r_p$, if $p(m, \vec{b}) \in r_p$ for the entire execution of $L$ with bounds $\vec{b}$. A set of requirements $R = \{r_{p_1}, r_{p_2}, \ldots\}$ is satisfied if all its requirements are satisfied:

$$m \models_{\vec{b}} R \iff \forall r \in R : m \models_{\vec{b}} r$$

This definition implies the capability to determine a non-functional property’s value from the mapping $m$ and loop bounds $\vec{b}$. For example, for a latency requirement on TCPAs, we can use the schedule determined during symbolic mapping to cycle-accurately compute it; other architectures might use the result of WCET analyses or similar.

To enforce that a loop program satisfies a set of given requirements for loop bounds $\vec{b}$, we must find a mapping at run time that satisfies them.

\(^3\) Note that for non-rectangular iteration spaces, $\vec{u}$ and $\vec{l}$ are the bounds of the rectangular hull of the iteration space.
Definition 13 ([26]). Run-time requirement enforcement on loops involves determining a mapping \( m \) of a loop program \( \mathcal{L} \) at run time such that \( m \models_b R \) for a given set \( R \) of requirements and loop bounds \( \vec{b} \):

\[
\text{given } \mathcal{L}, R, \vec{b} \\
\text{find } m \models_b R
\]

An entity performing run-time requirement enforcement on loops is called an enforcer.

An enforcer may freely choose among all mappings that satisfy \( R \). This freedom can be leveraged to optimise other non-functional properties \( P \). A prime example is energy—the program must stay within the latency corridor and consume as little energy as possible.

Definition 14 ([26]). Run-time requirement enforcement on loops with secondary objectives describes run-time requirement enforcement methods on loops that, while satisfying a set of requirements \( R \), also minimise a set of other non-functional properties \( P = \{ p_1, p_2, \ldots \} \) according to objective functions \( f_1, f_2, \ldots \):

\[
\text{given } \mathcal{L}, R, P, \vec{b} \\
\text{minimise } f_1(p_1, m, \vec{b}), f_2(p_2, m, \vec{b}), \ldots \\
\text{such that } m \models_b R
\]

Since the enforcer must solve this optimisation problem at run time, and since the execution time of the enforcement itself needs to be considered, it makes sense to consider simplified formulations of this problem. We refer to [26] for solutions.

Also of interest is the inverse problem: Find a symbolic mapping \( \mu \in M^{\text{sym}} \) and a region of PEs \( \vec{R} \times \vec{C} \) that satisfy a given upper bound \( L_{\text{upper}} \) on the latency \( L \) given the loop bounds \( \vec{b} \):

\[
m' := (\mu, \vec{R}, \vec{C}) \models_b [0, L_{\text{upper}}],
\]

where we assume that the worst-case execution times for enforcement and configuration of the target TCPA have already been subtracted from \( L_{\text{upper}} \). As one possible solution, Algorithm 2 iteratively determines the minimum
number $\hat{R} \cdot \hat{C}$ of PEs to satisfy the given requirement by evaluating the latency function $L$ (as determined during symbolic mapping). The algorithm iterates over the available PE allocations $\hat{R}, \hat{C}$, ordered in ascending number of PEs $\hat{R} \cdot \hat{C}$, and then over the set of symbolic mappings $M^{sym}$. Using the concrete loop bounds $\hat{b}$, it calculates the latency $L$. If the calculated latency $L$ for the current $m'$ is smaller than the upper bound $L_{upper}$, this mapping is returned since it satisfies the requirement. Due to the order the loop iterates in, we have found a mapping that achieves the latency bound with the minimum number of PEs.

Algorithm 2 Determining a minimal number of PEs to satisfy a given latency requirement $n_L = [0, L_{upper}]$

| Inputs: | set of symbolic mappings $M^{sym}$, latency bound $L_{upper}$, concrete loop bounds $\hat{b}$ |
| Output: | mapping $m'$ such that $m' := (\mu, \hat{R}, \hat{C}) \bigg|_{\hat{b}} [0, L_{upper}]$ |

for $(\hat{R}, \hat{C}) = (1, 1), (1, 2), (2, 1), \ldots, (R, C)$ do
  for $\mu \in M^{sym}$ do
    Compute latency $L$ symbolic schedule
    if $L \leq L_{upper}$ then
      return $m' := (\mu, \hat{R}, \hat{C})$
    no mapping found—error

For more details and experimental results, also see Section 6.4.

11.3 Compilation for General-Purpose CPUs

The Invasive X10 Compiler $x10i$ as shown in Figure 133 is based on the standard X10 compiler, but with front end extensions for invasive constructs, and a new back end for SPARC architectures, as well as optimisations for efficient utilisation of invasive hardware and operating system support, respectively.

11.3.1 Invasive X10 Front End

X10i uses the existing X10 front end which is available as open-source software. It can thus reuse the existing syntax and semantic checking code as well as the program representation. However, x10i needs to generate code for SPARC, the TCPA hardware, and the $i$-Core instruction set extensions, and it needs to ensure an optimal implementation of the invasive constructs. As the invasive platform is not a standard Posix/MPI platform, x10i also needs to provide a run-time library compatible with $iRTSS$ (see Chapter 10).
To tackle these requirements, x10i adds a custom back end to the existing X10 compiler front end. Programs analysed by the standard X10 front end are transformed into the libFIRM intermediate representation to facilitate generating SPARC code and supporting the i-Core extensions. This transformation also lowers parallel programming constructs like async, at, invade and infect into the APIs provided by the iRTSS. Figure 138 shows X10i’s overall structure [28].

To demonstrate this mapping from X10 language constructs to the iRTSS API, consider the following example:

```x10
finish {
    async { calc1(); }
    async { calc2(); }
}
```

The X10 code calls the functions calc1 and calc2 asynchronously within a finish statement. The X10 front end compiles the code to the following calls to the X10 run-time library:

```java
x10.lang.Runtime.finishBlockStart();
x10.lang.Runtime.executeParallel(()->{calc1()});
```

323
x10.lang.Runtime.executeParallel(()->{calc2()});
x10.lang.Runtime.finishBlockEnd();

Since the X10 run-time library partly implements functionality that is already provided by the iRTSS, x10i directly uses iRTSS API wherever possible. For instance, the methods of our example look like this:

```java
x10.lang.Runtime.finishBlockStart() {
    ...  
currentState.signal = simple_signal_init();
    ...  
}

x10.lang.Runtime.executeParallel(closure) {
    ...  
infect(currentState.claim, closure);
signal_add_callers(currentState.signal)
    ...  
}

x10.lang.Runtime.finishBlockEnd() {
    ...  
simple_signal_wait(enclosing.signal);
    ...  
}
```

Figure 139 shows the interaction of the involved i-lets at run time.

### 11.3.2 PGAS Programming on iRTSS

By taking advantage of the design of iRTSS, x10i can forgo having a user-level scheduler as part of the X10 run-time [13]. OctoPOS (see Section 10.2) offers the abstraction of an i-let, a short snippet of code that is assumed to run to completion most of the time. Additionally, as OctoPOS is built around exclusive resource allocation, it does not support preemption, making it possible to efficiently create and schedule a large number of i-lets. This approach can be viewed as putting a user-level-like scheduler into the kernel. Hence, there is no need for a user-level scheduler in the X10 run-time system. Thus, we directly map each activity (async block) to exactly one i-let and leave all further scheduling decisions to OctoPOS. Additionally, we also implement inter-tile operations (at) using i-lets. The resulting system offers the common operations with a very low overhead: 1469 cycles to start an activity on the local tile, and 1981 cycles to start an activity on a remote tile.
Table 21: A selection of optimisations rules found missing in GCC, LLVM, or ICC. “c” denotes a compile-time constant, so operations on c are free by means of constant propagation.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Compilers missing the rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>−x &amp; 1 → x &amp; 1</td>
<td>LLVM, ICC</td>
</tr>
<tr>
<td>∼(x + c) → ∼c − x</td>
<td>GCC, ICC</td>
</tr>
<tr>
<td>∼(c − x) → x + ∼c</td>
<td>GCC, ICC</td>
</tr>
<tr>
<td>x</td>
<td>(x ^ y) → x</td>
</tr>
<tr>
<td>(x + x) &amp; 1 → 0</td>
<td>ICC</td>
</tr>
</tbody>
</table>

11.4 Compiler Optimisations

11.4.1 A New Algorithm for SSA Construction

Static Single Assignment form (SSA form) is a cornerstone of any modern compiler intermediate language. However, the traditional SSA construction algorithm due to Cytron et al. [10] requires advanced program analyses, and requires that the input program be first built in a separate non-SSA intermediate language.

The libFirm compiler library contains a new, easier SSA construction algorithm that does not need previous analyses, and allows a program in SSA form to be constructed on-the-fly [1]. Furthermore, this algorithm was simple enough to be formally verified using the theorem prover Isabelle/HOL: It is proven both correct and optimal, i.e. it constructs the SSA form using the minimum number of \( \phi \) functions [7].

11.4.2 Compiler Synthesis

One advantage of the new SSA construction algorithm is that it applies local optimisations during the IR construction. Local optimisation rules, such as \( x + 0 \rightarrow x \) and \( x \& x \rightarrow x \), do not require any global analysis and, thus, can be applied at any time during compilation. These rules reflect the wisdom of the compiler developers about mathematical identities that hold for the operations of their intermediate representation. Unfortunately, these sets of hand-crafted rules guarantee neither correctness nor completeness.

This problem can be solved by compiler synthesis, i.e. automatically generating a key part of a compiler. For local optimisations, libFirm provides a generator for local optimisations, called Optgen [5]. Optgen enumerates all
local optimisations up to a given pattern size and verifies them using an SMT solver. Thus, Optgen guarantees completeness and correctness of the generated optimisations rules. Furthermore, Optgen can generate an optimisation test suite that helps to find missing local optimisations for state-of-the-art compilers. Using this test suite, Optgen identified more than 50 missing local optimisations with up to two operations in GCC, ICC and LLVM. Table 21 shows some examples of the optimisations found.

The same approach is also useful for instruction selection: The instruction set architectures (ISAs) of current processors are extended at an ever-increasing pace. In addition, we see the advent of application-specific, reconfigurable processors (such as the i-Core), which add new instructions for each application that runs on them.

A compiler should be able to support ISA extensions as quickly as possible. In order to support a new instruction, the compiler needs a model of it during instruction selection, i.e. the translation from the compiler’s intermediate representation to machine language. This model usually takes the form of a set of IR patterns that describe for which IR constructs the new instruction may be substituted.

Maintaining the set of patterns by hand is a tedious and error-prone process, which we have demonstrated by analysing the instruction selectors of both GCC and clang [6]. For example, take the instruction andn introduced to x86 with the BMI extension. With arguments x and y it computes \( \sim x \& y \), and the instruction selector will have a pattern to match that expression. However, it will then miss the equivalent expressions \( y \oplus (x \& y) \), \( x \oplus (x | y) \), and \( y \ominus (x \& y) \).

Therefore, libFIR also incorporates Selgen, a tool to automatically generate IR patterns for a set of machine instructions, ranging from just a new ISA extension to a sizeable portion of the x86 instruction set [6]. Selgen takes as input the semantics of both the compiler’s IR operations and the machine instructions. Then, it synthesises an exhaustive set of IR patterns for each instruction. We can thus be sure that the compiler is able to identify every opportunity to use a new instruction.

In particular, Selgen efficiently supports instructions which access memory by identifying the relevant subset of the address space. This vastly simplifies

\[{4}\] Note that this is not a case of missed optimisations because every expression contains two operations, and is therefore equally expensive to compute without a dedicated instruction.
the synthesis problem from modelling $2^{32}$ bytes to only the few ones actually accessed by the target instruction.

Selgen covers most of the integer subset of the x86 instruction set, as well as some instructions from the BMI extension. Thus, it easily covers the SPARC integer instructions, as they are less complex compared to the x86 ones. Furthermore, it demonstrates that the approach can handle small and mid-size special instructions.

### 11.4.3 Register Permutations

The tight software-hardware collaboration in invasive computing allows us to explore extended instruction sets: Previous work [4, 9, 11] showed that register allocation for programs in SSA-forms leads to an optimal assignment of registers. Translating out of SSA form however requires parallel copy constructs. These are traditionally implemented by sequences of mov and xor instructions for copying and exchanging values. In practice, minimising the number of these instructions is an NP-hard problem (for all known register allocation strategies). Using instruction set extensions that allow performing multiple exchanges within a single cycle, so-called permutation instructions, parallel copy constructs can be implemented more efficiently. This improves the run-time of the generated code and allows the usage of simple and fast copy-coalescing techniques.

libFIRM’s register allocators incorporate two approaches the generate permutation instructions [12]: The first is an efficient greedy algorithm that generates optimal code for all practically relevant parallel copy constructs. The second is based on dynamic programming, and solves the problem in all cases. Both approaches are proven to be optimal [8].

With an emulated version and the FPGA-based hardware implementation, we were able to conduct benchmarks using real programs. We used the programs from the SPEC CPU2000 benchmark as inputs to the compiler and evaluated the quality of the generated code. To find out under which circumstances the permutation instructions offer the biggest benefit, we tested different compiler configurations. We varied the used register allocator and its parameters as well as the used coalescing strategy. It seems that the permutation instructions are especially useful in just-in-time compilation scenarios because here, the compiler cannot apply a computationally costly algorithm to minimise the number of parallel copies.
11.4.4 Optimising Inter-Tile Data Transfer

Compute tiles in the invasive architecture have tile-local caches which are not synchronised, so there is no globally coherent view on the memory. This requires splitting the main memory into different regions for each tile, so that each region is only written to by one tile. Sending data from one tile to another requires using the DMA support in the iNoC. To achieve good performance programs have to be written in a way that communication between tiles happens in bulked data transfers. A form of read sharing is possible between multiple tiles, if the software enforces a synchronised cache flushing protocol.

X10i offers multiple possibilities to avoid and optimise inter-tile data transfers in X10 programs as described in [2].

One particular challenge on our platform is optimising programs written in the modern, object-oriented programming language X10. While X10 makes it easy to distribute work over multiple coherence domains by providing built-in support for message passing, X10 programs, due to their high-level nature, often transfer pointered data structures.

However, passing pointered data structures over a message passing interface entails costly (de-)serialisation. Consider the situation that tile $S$ has a linked list in its memory partition and wants to send it to tile $R$. Tile $S$ must first convert the list to a format suitable for message passing, i.e. serialise it to a byte stream, which $R$ then receives to reconstruct (deserialise) a copy of the original list. The (de-)serialisation causes a large overhead, both memory-wise and computation-wise. As such pointered data structures occur frequently in general-purpose applications, especially if written in high-level object-oriented languages like X10, it is important to accelerate their transfer.

Data transfers can be accelerated on non-cache-coherent systems by exploiting shared memory and managing coherence in software, i.e. explicitly triggering the needed cache write-backs and invalidations. However, existing techniques for transferring data in this fashion were limited to contiguous “flat” data structures not containing any pointers. Hence, in our setting, most data transfers would still require (de-)serialisation.

Therefore, X10i features a novel approach for transferring pointered data structures between shared memory partitions without requiring coherent caches [12, 14]. In this approach, the receiver directly accesses the data structure in the sender’s memory partition and makes a deep copy of it, i.e. clones it, in the receiver’s partition, thereby avoiding the need for serialisation and temporary buffers. To guarantee correctness, the software forces the necessary
cache write-backs and invalidations with object granularity. Figure 140 shows a schematic comparison of this direct data transfer approach compared to existing techniques. In a programming language following the partitioned global address space (PGAS) model, such as X10, the compiler and runtime system can issue the cache operations safely and fully automatically with zero overhead. Existing X10 programs do not have to be modified to benefit from this technique. Benchmarks using the standard X10 benchmark suite IMSuite show that direct cloning reduces the time spent for communication by up to 39.8% compared to copying with serialisation. When large object graphs need to be transmitted between tiles, this corresponds to an overall speedup of 2.3×.

Moreover, both the previous approaches and the new technique benefit greatly from range-based cache operations (called range operations in the following), i.e. hardware-supported operations that write back, invalidate, or flush all cache lines relevant to a given (logical) address range. This functionality has been requested before in prior work on software-managed cache coherence, however, so far it was never explored in the context of a non-cache-coherent architecture.

To exploit this opportunity, the invasive hardware prototype offers these non-blocking range operations, and X10i is able to use them. The range operations are implemented as an instruction set extension to the LEON3 processors, offloading the actual work to an enhanced cache controller. Whenever the processor does not execute a load or store, the cache controller uses this spare cycle to work on range operations. For a cache operation spanning $n$ cache lines, the cache controller needs $n$ spare cycles. If the cache controller finds enough spare cycles before the next cache operation issues, it can effectively
hide its latency. From the processors point of view, the cache operations then execute in one clock cycle. For an average data transfer, this is usually possible.

### 11.4.5 Near-Memory Computing

The optimisation we have just seen already cuts out a major source of inefficiency by directly copying object graphs between tiles. However, another challenge remains since the InvasIC MPSoC is a distributed-memory system: The CPU cores on the tiles are located “far from” the main memory. This means that main memory accesses need to traverse the NoC, which incurs a latency of at least 100 cycles. We can therefore expect large performance gains by running some recurring computations “near memory”, i.e. on dedicated hardware (the near-memory accelerator or NMA) positioned close to the main memory controller.

The conventional way to accelerate copying data in memory is the DMA engine found in many embedded systems. However, this is insufficient for object graphs because it can only copy contiguous blocks of memory, and the objects in an object graph usually occupy several small memory regions.

The invasive hardware prototype (see Chapter 15) therefore includes NEMESYS, Near-Memory Graph Copy Enhanced System-Software [15], leveraging the near-memory accelerators presented in Chapter 9. Concentrating on system software yields several benefits: Since every application uses the underlying system software, every application can profit from the NMA. Moreover, applications do not need to be rewritten to take advantage of new system software.

The core component of NEMESYS is an NMA unit which uses a hardware implementation of the Deutsch-Schorr-Waite algorithm [16] to traverse the source object graph and copy the objects to the destination. This algorithm is particularly suited for hardware implementation since it can do without a recursion stack by storing reverse pointers in the objects themselves. Figure 141 shows the basic steps of the algorithm: allocating a new object, copying it recursively, and going back to the parent object using the reverse pointer. For a detailed description of the hardware, see Section 9.6.

In addition, the NMA needs to detect and re-use objects it has already copied in order to preserve the graph structure. Otherwise, the destination graph would contain two copies of an object which is reachable via two different paths from the root.
Figure 141: Object graph traversal in NEMESYS. On the left, the source graph is shown, and to its right the steps needed to copy $o_4$ to $o_4'$. The red triangle marks the object currently under consideration by the NMA. First, the NMA allocates a new object and sets $o_2'$ to point to it. It then moves to the new object, setting the reverse pointer, and copies $o_4$ to it, recursively following its pointers if necessary. Finally, it returns to $o_2'$ by following the reverse pointer.

NEMESYS accomplishes this with a hardware hash table using the classical universal hash function $H_3$ due to Carter and Wegman. To hash $n$ bits of input down to $k$ bits, $H_3$ defines a set of functions $h_M$ parameterised by a $k \times n$ bit matrix $M$. The hash $h_M(x)$ is then given by $h_M(x) = \bigoplus_{i=0}^{n-1} M_i x_i$, where $M_i$ is the $i$-th column of $M$, $x_i$ is the $i$-th bit of $x$, and $\bigoplus$ is the exclusive or operation. This function is easily and cheaply implemented in hardware, only requiring $n \cdot k$ XOR gates for a fixed matrix.

Evaluating the performance of NEMESYS on the X10 IMSuite benchmarks, we find speedups between $1.35\times$ and $3.85\times$ compared to a pure software implementation.

11.5 Acknowledgement

Many researchers and students have contributed to the challenges of compilation of invasive programs and their mapping to heterogeneous multi-tile architectures as considered in this chapter. At this point, we would like to express our particular thanks to the following list of persons for their very valuable contributions to the ideas, concepts and compilation-related implementations through many years of research: Srinivas Boppu, Matthias Braun, Sebastian Buchwald, Manuel Mohr, Éricles Sousa, and Alexandru Tanase.

References


12 Security in Invasive Computing Systems

Felix Freiling, Wolfgang Schröder-Preikschat, Gregor Snelting, Ingrid Verbauwhede, Franziska Schirrmacher, Furkan Turan, Simon Bischof

Abstract We study security aspects of resource-aware programming in heterogeneous multicore systems. Based on a well-defined attacker model, we first survey specific security requirements of invasive computing and then present security mechanisms at different architectural layers.

12.1 Introduction

Security remains an elusive property in computing systems for many reasons. Firstly, it takes the actions of a malicious entity into account which is usually called the attacker. Obviously, attackers are a nuisance that no system designer loves since it seems hard already to get a system right if only “normal” actions can happen. Designing for security, however, needs to take additional “abnormal” actions into account: actions that happen at a wrong time or in a wrong context, or simply actions that are unexpected altogether. Since security mechanisms need to take additional behaviour (by the system or its environment) into account, secondly, there always is a price of security. This price is paid in additional resources, be they time, space, hardware or software. Certainly, nobody wants to pay too high a price, especially if it is not so clear what type of attacker is to be expected. But it is a commonplace insight that the price of security rises with the malicious abilities of the attacker. The difficulty is to realistically assess the abilities of the attacker, to not “pay” too little or too much. Finally and thirdly, it is notoriously hard to protect secrets in computing systems because of the many intricate ways that information can leak, be it through storage, timing or any other form of observable resource usage that cause unanticipated (covert) communication channels.

In particular, heterogeneous multicore systems with a high level of integration (MPSoCs) are a challenging environment for security research since the hardware/software integration creates many possibilities of covert channels. Consequently, such systems generally offer a large attack surface.

In this chapter, we survey selected advancements in the area of security in invasive computing systems from recent years. These contributions necessarily cover all main architectural layers: the hardware layer (i.e. isolating
applications running on the same core against system level attackers), systems software layer (i.e. isolating memory abstractions against application level attackers), and the application layer (i.e. integrating security requirements into programming languages).

The overall goal is to ensure the classical security properties of confidentiality, integrity and availability of an invasive computing system in the presence of untrustworthy programs that compete for resources and can contain malicious functionality. To take account of the computing environment of heterogeneous MPSoC architectures, security mechanisms must focus both in intra-tile as well as inter-tile security. This means that we have to protect applications running on the same tile against a set of increasingly stronger attacker assumptions, and at the same time extend these protection guarantees to applications running on multiple tiles. For these applications, our mechanisms aim to reliably enforce security properties in invasive computing.

Chapter outline: We begin with a description of the attacker model underlying our work in Section 12.2. It comprises attackers at different architectural levels and with different abilities. We then develop a set of comprehensive security requirements for invasive computing in Section 12.3 and present a classification and overview over previously established security mechanisms in Section 12.4. We then present several novel contributions:

- a combination of static quantitative information flow control and control flow attestation (Section 12.5),
- novel techniques for quantitative information flow control for \( \varepsilon \)-confidentiality (Section 12.6),
- a heuristic run-time-monitoring approach to detect covert channels (Section 12.7),
- a novel application of memory encryption on inter-tile virtual shared memory (Section 12.8), and
- a mutual attestation mechanism to check and (re-)enforce the integrity of computations on remote tiles (Section 12.9).

We summarise and conclude in Section 12.10.

12.2 Attacker Model

Our attacker model consists of four hierarchic levels at which an attacker can operate and execute software [14]. The first level concerns the possibility
to execute own code written in a high-level programming language like X10 [33]. Consequently, an X10-attacker corresponds to an attacker who can run programs written in X10. These programs can be statically checked through a trusted X10 compiler, strongly inhibiting malicious behaviour, e.g. by the type system of X10 (see Chapter 11). In contrast, the binary attacker may execute arbitrary (binary) code that runs with privileges associated with normal applications (usually user level privileges). With the OS-level attacker, we allow an attacker to take over control of the operating system. Finally, physical attacks are the most powerful ones which are considered to be technically difficult to perform, but also to defend against.

Note that the above attacker models are strictly increasing with respect to the behaviours the attacker is allowed to exhibit. For instance, a binary attacker can mimic an X10-attacker (i.e. reproduce all behaviours the X10-attacker may generate), but since the binary attacker is not restricted by the type system of the programming language, that attacker might also exhibit additional behaviours. In this sense, the described attacker models form a strict hierarchy

$$X10\text{-level} \subset binary \subset OS\text{-level} \subset physical$$

regarding subsets of possible attack behaviours.

In certain contexts of invasive computing, such as robotics (see Chapter 13) or High-Performance Computing (see Chapter 14), it can be assumed the invasive hardware can be sufficiently well protected and not expecting physical attacks can be considered a reasonable assumption [36]. The literature in this area has therefore concentrated on the dangers of code-based remote exploitation instead of exploitation through physical access. Overall, the strongest attacker model considered in the context of invasive computing is the OS-level attacker which comprises an absent or untrusted system layer accessing shared resources.

12.3 Security Requirements and Isolation Concepts

The well-known “CIA taxonomy” [12, 39] states that security properties can be classified into three classes: Confidentiality, Integrity and Availability. Roughly this can be formally approximated with the property classes of safety, liveness and information flow (also known as hyperproperties) [5]. In the literature on invasive computing, the focus has been on confidentiality and integrity, as we explain shortly. To establish these security properties in invasive computing systems under a given attacker model, security mechanisms must be employed. No matter whether integrity or confidentiality require-
ments are concerned, the general concept of *isolation* can be used to achieve these properties. We will use different isolation concepts to classify the security mechanisms presented in the remainder of this chapter.

### 12.3.1 Confidentiality Requirements

Regarding confidentiality, a new class of properties was developed that can quantify the amount of information leakage that a given system is known to have in the worst case: \( \epsilon \)-confidentiality \([14]\). The difference between classical confidentiality and \( \epsilon \)-confidentiality is delicate: For most practical situations, requesting classical confidentiality suffices as it corresponds to the “classical” understanding of confidentiality as provided by standard run-time protection techniques in operating systems (memory protection). The requirement of \( \epsilon \)-confidentiality intuitively protects at least as much as classical confidentiality, but furthermore provides “state-of-the-art” protection against side-channel attacks. If requesting \( \epsilon \)-confidentiality, the programmer has to specify a value \( \epsilon \) which is the maximum acceptable rate of information flow in bits per second (bps). The formal definition is as follows.

**Definition 15 (\( \epsilon \)-confidentiality).** The invasive software \( S \) satisfies \( \epsilon \)-confidentiality for attacker \( A \) and environment \( E \) iff there exists evidence of attacks on \( S \) by \( A \) in \( E \) that lead to unauthorised information leakage of at most \( \epsilon \) bits per second (bps).

In the definition, \( S \) refers to a concrete hardware/software configuration of an invasive application, \( A \) refers to the different types of software-only attackers from the attacker model, and \( E \) specifies all other environmental circumstances outside of \( S \). Intuitively, the definition can be rephrased as follows: “\( S \) satisfies \( \epsilon \)-confidentiality iff the best published attack on \( S \) achieves \( \epsilon \) bps.” This is how the definition takes the “state-of-the-art” into account, it is therefore relative to the knowledge of existing attacks on \( S \).

Theoretically, if nobody has attacked the system yet, it satisfies \( \epsilon \)-confidentiality for \( \epsilon = 0 \) bps. This shows that the concept is only meaningful if many people have tried very hard to attack \( S \) and have published their results. Our definition therefore corresponds to security definitions of cryptographic primitives that also depend on how well researched a primitive is. Note that the value \( \epsilon \) can also be understood as an inverse metric to the “effort” that is currently considered to break the system. It does not answer the question how many bits of leakage are acceptable in a given application context.
12.3.2 Integrity Requirements

In addition to the “classical” definition of integrity [9], a specific variant of integrity for invasive applications has been defined: integrity with attestation (Iwa) [14]. Requesting Iwa implies requesting classical integrity, i.e. a program can be sure that code and data have not been modified in an unauthorised manner. The difference between this and Iwa refers to the possibility of proving to a third party that integrity has been satisfied. As discussed above, this is generally known as attestation in the literature [8]. Iwa is interesting for applications that are deployed on remote computing centres in which the physical environment or the system maintainers might be untrustworthy (as is the case in the application scenarios of invasive computing, see Chapters 13 and 14). Using Iwa, the application receives a special token (the attestation or proof) which it can send to a third party that can verify integrity.

12.3.3 Isolation Concepts

In general, isolation means that two domains of activity do not interfere with each other. Integrity requirements generally can be understood as attackers not being able to write whereas confidentiality as not being able to read [4]. We briefly explain several isolation levels that can be used to distinguish security mechanisms (not only) in invasive computing.

**Basic Isolation** The concept of basic isolation ensures basic read/write protection for applications. Isolation concepts like horizontal isolation of the system layer against applications as well as vertical isolation of applications against each other. For example, such solutions come in the form of dynamic memory protection at the operating system level (see Chapter 10).

**Strong Isolation** To guarantee the correct execution of intended behaviour of an application, not only in the face of competing or malicious programs, but also in view of an untrusted system layer or in the absence of a system layer, stronger degrees of isolation are required. Predictability of invasive programs in terms of correct execution must involve a trusted computing base (TCB) that guarantees the confidentiality and integrity of a program’s code and data. Confidentiality and integrity can both only be provided if there exists a trust anchor for applications. This can be assumed to exist in software, but to resist attackers who have gained system level privileges, a zero-software TCB must exist on top of an immutable hardware trust anchor.
**ε-Isolation**  TCB-based strong isolation concepts, however, may still allow information to leak from an application through side channels. The concept of **ε-isolation** additionally requests that the best known attack attains information flow up to ε bits per second, i.e. ε-confidentiality (see above).

**Spatial Isolation**  In general, ε-isolation aims to limit information flow over shared resources up to a certain bound. A related concept is **spatial isolation** [40] which is based on the observation that the amount of shared resources used by different applications concurrently running on modern MPSoCs decreases with the physical “distance” they have in the system. Spatial isolation is a concept that has to be taken into account when performing the design-time/run-time mapping of applications [40]. Spatial isolation is then achieved through a mutually-exclusive allocation of resources to applications in the MPSoC. To this end, Weichselgartner et al. [40] first compute compact and connected application mappings (called *shapes* at design time. In a second step, run-time management uses this information to map multiple spatially segregated shapes to the architecture.

### 12.4 Previous Work

We now summarise the main previous work on security in invasive computing. It achieved different levels of isolation at different architectural layers against attacker models with different assumption coverage. As can be seen from the overview in Table 22, the choice of a particular isolation concept depends on the attacker model. While X10 attackers can be counteracted on the compiler level, binary attacks can be thwarted with basic isolation concepts. System level attackers, however, must be counteracted with new concepts for strong isolation. To achieve ε-isolation, additional measures must be taken. For sake of comparison, the research advances that are later described in Sections 12.5 to 12.9 are also included in Table 22.

#### 12.4.1 Application and Systems Software Layer

Early foundational work on security in invasive computing systems [14] proposed to establish security requirements of applications as *constraints* that could be requested before invasion (see Chapter 1). Security requirements could then be handled during resource allocation in a systematic way and even be declined, for example if the requested level of ε-confidentiality could not be guaranteed by the system. Still, malicious applications still can act in a greedy fashion and claim an unfair share of resources. This, however,
Table 22: Previous and current work on security in invasive computing systems categorised according to attacker model and achieved properties which are abbreviated as $C$ (confidentiality), $I$ (integrity), $\varepsilon$-$C$ ($\varepsilon$-confidentiality), $IwA$ (Integrity with Attestation).

<table>
<thead>
<tr>
<th>attacker</th>
<th>result</th>
<th>properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>X10-level</td>
<td>security as invasive constraints [14]</td>
<td>$C$, $\varepsilon$-$C$</td>
</tr>
<tr>
<td></td>
<td>quantitative information flow control [19, 20] (Section 12.6)</td>
<td>$C$, $\varepsilon$-$C$</td>
</tr>
<tr>
<td>binary level</td>
<td>dynamic memory protection (RAMcrypt) [18]</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td></td>
<td>identity-based encryption for inter-tile communication [42]</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td></td>
<td>combining information flow control with control-flow attestation (Section 12.8)</td>
<td>$C$, $\varepsilon$-$C$, $I$, $IwA$</td>
</tr>
<tr>
<td></td>
<td>heuristic covert channel detection (Section 12.7)</td>
<td>$C$</td>
</tr>
<tr>
<td></td>
<td>inter-tile virtual shared memory encryption (Section 12.8)</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td>OS-level</td>
<td>Soteria [17]</td>
<td>$C$, $I$, $IwA$</td>
</tr>
<tr>
<td></td>
<td>Atlas [27]</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td></td>
<td>SOFIA [11]</td>
<td>$I$, $IwA$</td>
</tr>
<tr>
<td></td>
<td>Hypercrypt [16]</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td></td>
<td>Exzess [41]</td>
<td>$C$</td>
</tr>
<tr>
<td></td>
<td>SGX-Kernel [32]</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td></td>
<td>TEEshift [25]</td>
<td>$C$, $I$</td>
</tr>
<tr>
<td></td>
<td>SEVGuard [31]</td>
<td>$I$, $IwA$</td>
</tr>
<tr>
<td></td>
<td>mutual attestation and integrity enforcement (Section 12.9)</td>
<td></td>
</tr>
</tbody>
</table>
may also happen accidentally and handled using classical resource allocation mechanisms (see Chapter 10).

Other work on the programming-language level focused on ensuring confidentiality through static information flow analysis using JOANA [35], a tool that can handle full Java with arbitrary threads, scales to 250000 LoC, and guarantees to discover all possibilistic and probabilistic leaks [7, 19, 20]. It also uses precise program analysis to ensure context-, flow- and object-sensitivity. In recent work, JOANA was extended to address quantitative information flow control for $\epsilon$-confidentiality, as well as timing side channels (see Section 12.6).

Previous work focusing on the systems software layer initially targeted basic isolation, such as solutions relying on transparent memory encryption both in software [16, 18] and hardware [14, 41]. The challenges of memory encryption for inter-tile communication have been previously also tackled using identity-based encryption [42].

### 12.4.2 Hardware Layer

Memory encryption is also a feasible solution to protect confidentiality of code and data against compromised operating systems. The hardware-based memory encryption solution Atlas [14, 27] has especially been designed to provide encryption for intra-tile accesses using an encryption unit (Figure 142) between the cache and main memory to guarantee confidentiality even in the event of a complete software compromise. An analysis of existing trusted-computing architectures [26] has shown that similar hardware protection solutions scale for manycore systems and can deal with heterogeneous memory. But while strong isolation guarantees can be given on intra-tile level, at least confidentiality is maintained on inter-tile level, e.g. for global memory. Atlas relies on a zero-software TCB to protect against system-level attackers. Prototype solutions have been implemented based on the LEON3 softcore processor, including toolchain extensions for developers.

Apart from memory encryption, the basis for establishing security properties in systems like Atlas [27] is the protected-module architecture (PMA) which is solely implemented in hardware to protect against OS-level attackers. An abstract view of a PMA is presented in Figure 143. PMAs, such as those developed by Atlas, have been influential to trusted computing architecture in academia and industry [26].

Strong isolation and attestation guarantees can also be achieved in a more static way [17]. Besides just maintaining the confidentiality and integrity of code and data, control flow integrity solutions have also been developed [10].
12.4 Previous Work

Figure 142: Encryption unit inserted between cache and main memory to guarantee confidentiality even in the event of a full (software-based) system compromise [27].

Figure 143: A state-of-the-art Protected Module Architecture (PMA) runs multiple Software Modules (SM) side by side, along with one or more unprotected applications. The Trusted Computing Base (TCB) ensures that the internal state of an SM is protected from any other software running on the system. The measurement of an SM establishes a Dynamic Root of Trust (DRoT). The result can be used to attest the state of the module to a remote verifier. By sealing data, the SM can send it securely to untrusted storage [26].
These provide some level of protection against OS-level software vulnerabilities. SOFIA [10] is a software integrity and control flow integrity architecture that allows the processor to defend against a large number of attacks, including code injection, code reuse, and fault-based attacks on the program counter. In addition, the architecture also defends against software copyright infringement and reverse engineering. All protection mechanisms are enforced in hardware using cryptographic techniques at the finest possible granularity. SOFIA has been implemented on the LEON3 microprocessor. Several benchmarks were compiled and executed on the modified core running on an FPGA, showing an average total execution time overhead of 106% compared to an unmodified LEON3 core [10].

In the same direction as Atlas [27], other solutions have been developed that allow unmodified processes to transparently work on encrypted data. RAMcrypt [18] can be deployed and enabled on a per-process basis without recompiling user-mode applications. In every enabled process, data is only stored in cleartext the moment it is processed, and otherwise remains encrypted in main memory. In particular, the required encryption keys do not reside in main memory, but are stored in CPU registers only. Hence, RAMcrypt effectively thwarts memory disclosure attacks, which grant unauthorised access to process memory, as well as physical attacks. In the default configuration, only up to 4 memory pages are exposed in cleartext at the same time which still allows proper execution of the system.

As seen with RAMcrypt [18], the spectrum of solutions targeting OS-level attackers is not flat. Established techniques often base some trust in parts of the systems software. For example, Hypercrypt [16] is based on a trusted hypervisor that encrypts the entire kernel and user space to protect against attacks on main memory. This solution is fully transparent to the guest operating system and all applications running on top of it. At any time, only a small working set of memory pages remains in clear while the vast majority of pages are constantly kept encrypted. By utilising CPU-bound encryption [30], the symmetric encryption key is never exposed to main memory. With the default configuration of 1024 cleartext pages, successful attacks are rendered highly unlikely due to large caches on the invasive platform. In some scenarios not even the hypervisor can be trusted. Exzess [41] is a hardware-based mitigation method against memory disclosure attacks based on the idea of a memory proxy. Exzess effectively protects against attacks on main memory, while being transparent to applications and the operating system after initialisation.

Other approaches protect systems-software and application code by using existing hardware-based trusted execution environments [26]. For example,
SGX-Kernel [32] uses Intel’s Software Guard Extensions (SGX) to isolate operating system components despite the fact that many existing trusted computing solutions have not been designed to work in kernel mode. SGX-Kernel wraps kernel functionality within secure containers by moving parts of it to user space. Kernel components are strictly isolated from each other such that a vulnerability in one kernel module cannot be escalated to a full kernel compromise. The solution protects integrity and the confidentiality of the secure container against all software level attacks as well as physical attacks. In a similar approach, TEEshift [25] protects the confidentiality and integrity of application code by shifting selected functions into trusted execution environments like Intel SGX. Only after attesting to a remote party that the loading SGX enclave behaves with integrity, the functions are decrypted, but remain inside the enclave protected against reverse engineering. Similarly, SEVGuard [31] allows to secure one application instead of an entire VM based on AMD’s technology of Secure Encrypted Virtualization (SEV).

After surveying previous work, we now turn towards several recent results that are described in the following sections.

12.5 Information Flow Control via Control Flow Attestation

The approach of non-interference [15] specifies the absence of information flow in a very abstract way. A (secret) system $X$ does not interfere with a (public) system $Y$ if $X$ does not influence or change the possible behaviours of $Y$. The property of non-interference disallows any form of information flow whatsoever. Techniques to constrain or analyse all possible behaviours at compile time are collected under the heading of language-based information flow control [28], in which the absence of information flow is enforced by, e.g. static analysis of the source code. One such tool is JOANA [19], which can verify non-interference for Java programs.

Non-interference is particularly relevant in scenarios of remote computation, i.e. where a client wishes to execute a critical computation on a remote server (e.g. in a computing centre or “in the cloud”). If the computation operates on sensitive data like company secrets or personal data, the client has an evident interest that not only the computation behaves as expected but that also the sensitive data is protected from unauthorised access. This is particularly hard if the client cannot physically guard the servers.

For remote computations it is usually assumed that the attacker has full physical access to the server and can interact with it in arbitrary ways, e.g. for
remotely running Java programs the attacker could extract memory contents by listening on system buses on the remote server. To be able to ensure any form of security, it is usually assumed that the server contains an inherently trustworthy software subsystem (the TCB). For remote computations, next to software the TCB usually contains some hardware providing a trusted execution environment [26].

One way to protect the integrity and confidentiality of remote executions is to move the entire computation (together with its runtime environment) into the TCB [13, 24]. There are also other approaches like Uranus [23] that try to minimise the size of the TCB by only running annotated Java functions inside the SGX enclave. A critical factor in security assessment, however, is the size of the TCB: Small TCBs usually provide less attack surface and are therefore considered more secure.

Tools like JOANA ensure the absence of information flow as long as the program executes as expected, i.e. as long as it does not deviate from its allowed control flow. Violating control flow is a deliberate goal of attackers like in return-oriented programming (ROP) [34]. To ensure the integrity of a binary during execution, the concept of Control Flow Integrity (CFI) [1] can be used. Briefly spoken, CFI aims at guaranteeing that the concrete control flow of an application does not deviate from a set of specifically allowed control flows.

A common approach to guarantee control flow integrity during runtime is called control flow attestation [2]. This means that the verifier compares the actual execution path of the running application with a previously generated set of control flow sequences that define its allowed behaviour. The verifier can therefore check sequences of states (executions) rather than just a single state. The published mechanisms for control flow attestation (like C-Flat [2]) focus on the execution of binary code in embedded systems, not on executions of high-level languages like Java.

We developed a solution that combines static information flow control with remote control flow attestation to yield a system that guarantees confidentiality and integrity for Java programs running on a remote computer. We combine the information flow control provided by JOANA with a novel dynamic control flow attestation system for Java programs which is based on a relatively small TCB implemented using Intel SGX. By guaranteeing (and attesting) the integrity of control flow to an external observer, the information flow guarantees given by JOANA extend to the runtime of the application. This provides, for the first time, the enforcement of strong non-interference guarantees against
for r1: [0,1,...,15]
    r2 := state[r1]; // state depends on key and plain text
    r3 := sbox[r2]; // sbox is a constant array
    state[r1] := r3
end

Figure 144: An AES implementation.

system level attackers at run time for remotely executing instrumented Java bytecode programs.

12.6 Language-based Information Flow Control

Language-based information flow security is a subfield of IFC, in which information flow control is enforced by the syntax of the programming language, by the compiler or by the type system of the programming language, or by static analysis of the source code. For that we assume that only the program itself causes information flow and the system environment is secure and free of side channels.

12.6.1 Timing-Sensitive Information Flow Control

A very common information flow side-channel is the time taken by the program. Even if the output itself does not give any clues to the attacker, the run-time might depend on the secret, for example if additional instructions might be executed depending on the secret, or the secret influencing cache timing. From a control flow graph (CFG), timing dependencies can be calculated automatically, and the correctness of this calculation has been demonstrated by a machine-checked proof [21]. A simple CFG can also be extended with cache information, which makes it possible to find timing leaks due to cache effects.

As an example, consider the implementation of the encryption algorithm AES in Figure 144. The array sbox holds constants which typically spans multiple memory blocks, while r1, r2, r3 are registers. Thus the value of r2 in one iteration may influence whether the read of r3 in a later iteration is served from cache (namely if, earlier, the corresponding memory block was already loaded into the cache), or from main memory. This makes a difference in execution time and can be observed by an attacker; who may thus be able to
infer the value of \( r_2 \). Such leaks can be discovered by timing-sensitive control dependencies: \( r_2 \) has a timing dependency to the end of the program, and \( r_2 \) depends on the secret, meaning that the secret is found to influence program runtime.

This analysis can also be extended to work with arbitrary CFGs, increasing the range of programs that we can analyse. A usual restriction is that CFGs contain a unique exit node, i.e. a node without successors reachable from every other node. But this makes it impossible to model programs with multiple exit nodes, e.g. for Java with exceptions, or loops that cannot be exited, e.g. reactive systems.

### 12.6.2 Information Flow Control for Parallel Programs

Parallel programs open up a new way for an attacker to learn secret values. In the example in Figure 145, all possible secret values have the same possible executions: Either it prints 0 and then 1, or it prints 1 and then 0, depending on scheduling decisions. However, the higher the secret value is, the more probable the first possibility is. Probabilistic non-interference guarantees that the secret value does not influence the probability distribution of the public output.

We developed a new algorithm for probabilistic non-interference [6] (see Figure 146) that is substantially more precise than existing ones. The soundness of our criterion is proven by a machine-checked proof. This algorithm was integrated into JOANA to provide IFC for full multithreaded Java with unlimited threads.

To increase the precision even more, the classification given in Figure 146 is split into two parts: one that describes the traditional classification of statements and one that describes timing classification between statements, with both of them influencing each other, leading to a mutual recursive relation.
1. \( \forall n \in N: cl(n) \geq \bigcup_{m \rightarrow n} cl(m) \)

2. \( \forall n, m \in N: MHP(n, m) \land c = cda(n, m) \land \exists c' \in \text{path}(c, n), cl(c') = H \Rightarrow cl(n) = H \)

3. \( \forall n \in I: ucl(n) = cl(n) \) and \( \forall n \in O: ucl(n) \geq cl(n) \)

Figure 146: The new RLSOD criterion, describing when a Program Dependency Graph can be correctly classified by \( cl \), which implies that the program is non-interferent. Rule 1 handles sequential dependencies, rule 2 handles non-determinism arising from scheduling and rule 3 ensures that the classification respects the input and output security levels \( ucl \) given by the user.

between traditional and timing classification. Using the results described in Section 12.6.1, we get a much smaller set of statement pairs whose timing classification needs to be checked, leading to less false alarms.

12.6.3 Quantitative Information Flow

Traditionally, IFC checks for non-interference as security property. This results in a binary answer: Either the secret inputs of the program cannot have any influence on the program output or the security analysis finds that there may be such an influence, no matter how small it is. In the latter case, the program is rejected as insecure. In certain cases, however, such influence cannot be prevented since it is part of that program’s functionality. With quantitative IFC, one can deduce bounds on how many bits a program can reveal, giving a much more fine-grained answer than traditional IFC. To calculate these bounds, we use a data flow analysis to calculate constant bits and dependency relations between different bits in the program. This results in a dependency graph on the bit level. A minimum vertex cut is then computed to obtain a minimum number of bits which determine the output of the program (see Figure 147).

Thus, we get a sound approximation of the leakage [3]. For if-statements, we increase precision in the then- and else-branch by using guarantees we get from knowing that the if-condition must have been true or false, respectively. By using techniques based on summary edges, the analysis is interprocedural, including arbitrary recursion. While traditional quantitative IFC only works for batch programs, our approach has been extended to deal with interactive programs, allowing user input and output at arbitrary program points. This novel algorithm has been integrated into JOANA.
12 Security in Invasive Computing Systems

\[
x = (h_1 \& 1 == 1) ? h_1 : 3;
\]

\[
y = (h_2 \& 1 == 1) ? h_2 : 3;
\]

\[
s = (x == y);
\]

\[
o = s ? x : 3;
\]

\[
p = h_2
\]

Figure 147: An example program and its bit dependence graph. A minimum vertex cut is marked in blue. Since the cut has size 3, the program leaks at most 3 bits.

12.7 Run-Time Monitoring for Side-Channel Detection

Heuristic approaches can be used to monitor system behaviour for suspicious behaviour. This is facilitated if distributed monitoring systems are deployed (see Chapter 8). While in general it is impossible to reliably detect information leaks, we exploit the fact that the attacker must move first and devise an approach that can dynamically choose the monitors, encodings, and statistical tests that it performs.

The isolation of applications in invasive computing already narrows down the number of possible side channels through a by-default spatial isolation of applications during invasion. However, Masti et al. [29] demonstrate the use of temperature sensors to build a side channel on an Intel Xeon server platform despite strong isolation techniques. Therefore, we also focused our research on thermal side channels.

We demonstrated that such a side channel exists in invasive computing in a proof-of-concept implementation. Current approaches employed frequency scanning to detect communication patterns [22]. In contrast, we identified the use of neural networks to test for deterministic communication patterns as a promising approach. To this end, we demonstrated the competing performance of neural networks on communication detection in our implementation. Here, the neural network (executed on a TCPA, see Chapter 6) evaluated the temperature values of specific applications within a predefined time window. We used known enforcement techniques [22] to counteract the attack if a malicious communication pattern is detected.

Figure 148 provides an overview of the proof-of-concept implementation. The proposed security mechanism has four steps. First, a security application buffers the temperature data of all applications on the tile. Second, the TCPA
Figure 148: Thermal covert channel detection in invasive computing. The security application gathers the temperature data of all applications located on the tile. An interface on the TCPA allows transferring temperature data and the results of the neural network. The security application applies enforcement techniques to the application identified as malicious from the neural network.

receives the buffered temperature data after the predefined time window. Third, the neural network evaluates the data and reports the results to the security application. Lastly, the security application reacts to the results if necessary with enforcement techniques.

The solution allows system operators to test both external suspicious applications and internal ones to establish confidentiality values of $\epsilon$.

### 12.8 Virtual Shared Memory Encryption, Integrity and Access

Confidentiality and integrity protection for intra- and inter-tile memory access can be guaranteed through MPU access control while the Atlas hardware-based memory encryption mechanism [14, 27] also provides encryption for intra-tile accesses. While a hardware-based approach such as Atlas is usually preferable due to its higher performance and smaller TCB, we considered a software-based approach to be a useful addition in case the OS is assumed to be trusted. Software-based virtual shared memory (VSM) encryption thus complements Atlas by improving portability to non-Atlas-capable hardware, better supporting inter-tile accesses and being more flexible as well as feature-rich while compromising on the strength of the attacker model and performance.

This extended the ability of OctoPOS for VSM (Section 10.2.1), which means transparent access to the different levels of the memory hierarchy as defined by an invasive architecture for PGAS systems, enabling a stronger memory model with memory consistency on demand while also providing an intervention
point to enhance page accesses via the system software. To ensure scalability in larger systems and improve the protection guarantees of non-local memory storage and DMA transfers, OctoPOS-based (Chapter 10) VSM encryption was implemented where remote pages are transparently de-/encrypted on inter-tile access, optionally also guaranteeing integrity and authorisation for access of the respective memory region. VSM security operations are performed on a set of memory pages, where the extent is determined by the application’s desired level of security, an assessment of the claim neighbourhood e.g. via CFA (Section 12.5) and the application’s other relevant characteristics such as memory access behaviour and performance demand. Our technique complements the continuum of security choices available in an invasive system, ranging from no security over language-based type-safety (see Chapter 11), MPU and MMU protection, VSM security, i-Core-specific security features (see Chapter 5 and Chapter 9) to Atlas.

12.9 Hardware-based Mutual Attestation

The concept of the trusted execution environment can be extended from one tile to another [38]. For this specific purpose, we introduced the idea of mutual attestation, which allows the tiles to evaluate each other directly and manage their cooperation accordingly. A tile’s boot or invasion enforces an immutable subroutine, which measures available resources and attests them to neighbouring tiles. The resources, in this context, refer to code and data files and predefined memory regions. Next, the involved tiles evaluate the attested resources of each other. This evaluation aims to decide which tile is trusted compared to the other based on the available resources. In fact, the evaluation even goes further and checks if a tile can find a relatively more trusted version of a specific resource at its neighbours. To aid this decision, we propose extending the resources with metadata for defining them. A substantial component within this definition is version information, for the assumption that a recent version of a resource is expected to be trusted compared to its older versions, as it should address the known vulnerabilities. After the decision is made using the comparison with the metadata, the attestation responses occur. These responses aim to update a tile’s untrusted resources with the trusted ones available at the neighbouring tiles. The responses extend to preventing even Denial-of-Service (DoS) attacks by restoring the trusted execution state of even exploited tile.

For reliably executing the attestations, evaluation, and response mechanisms, we designed a new TCB, as an immutable hardware Root-of-Trust (RoT) attached to each tile. The RoT modules are supported with a link to the net-
work adapter for direct communication to each other, and with a lightweight cryptography module. These prevent the RoT from relying on the system software, hence avoiding any disruption of its operation by potentially exploitable software.

To minimise the cost of equipping each tile with the proposed RoT module, we designed a compact elliptic curve implementation [37]. The module integrates Ed25519 digital signature and X25519 key establishment schemes in a single module. Our design opts for minimising resource use almost without sacrificing fast execution. Furthermore, the design even offers customisable countermeasures against timing and differential power analysis side-channel attacks and fault-injection attacks.

12.10 Summary and Acknowledgements

In this chapter, we surveyed security aspects of resource-aware programming in heterogeneous MPSoC systems and security research results related to invasive computing and respective MPSoC architectures. Our focus was on security mechanisms developed in Project C5 within CRC/Transregio 89, a joint effort between at least three research groups at FAU and KIT involving many people over the lifetime of the project. The chapter authors especially wish to thank past PhD and postdoctoral researchers involved in InvasIC: Johannes Bechberger, Davide Bove, Ruan de Clercq, Gabor Drescher, Johannes Götzfried, Pieter Maene, Tilo Müller, and Alexander Würstlein for their efforts that helped shape this relevant research domain.

References


D: Applications
13 Invasive Software-Hardware Architectures for Robotics
Tamim Asfour, Walter Stechele, Nael Fasfous, Felix Hundhausen, Fabian Paus

Abstract We have explored the specific benefits and restrictions of invasive architectures in challenging real-time embedded systems and in particular in humanoid robotics. To this end, we have investigated the implementation of a cognitive robot control architecture with its different processing hierarchies, both on invasive TCPA and RISC-based multiprocessor system on a chip (MPSoC). The goal has been to explore techniques of self-organisation to efficiently allocate available resources for the timely varying requirements of robotic applications. We have shown in multiple robotic application domains that less computing resources are needed to fulfil the application requirements compared to traditional resource assignment at compile time.

13.1 Introduction
In this chapter, we investigate the benefits and limitations of invasive computing in robotic applications, which combine methods and algorithms from the area of computer vision, motion planning, and scene understanding with concurrent processes and timely varying resource demands.

Resource-aware robotic algorithms were developed to consider the trade-off between quality of their results and the required resources. During an object localisation and tracking task, the number of features to extract and match in each frame is a crucial parameter. On the one hand, the higher the number of tracked features, the higher the accuracy of the tracked object pose. On the other hand, a higher feature number also increases run-time and memory consumption during execution. A resource-aware algorithm can model the impact of its parameters on its resource requirements and adjust them dynamically to meet resource or quality requirements.

We investigated how resource-aware algorithms can be used in the context of humanoid robots. By taking inspiration from the core principles of resource invasion and retreat (see Chapter 1), we developed novel resource-aware algorithms with self-monitoring capabilities to identify the difficulty of the underlying problem.
The following topics are addressed in this chapter:

- **Feature Detection**: Harris corner detection and SIFT feature extraction and matching
- **Object Classification and Segmentation**: Convolutional neural networks for object classification and localisation in resource-constrained embedded systems
- **Low-Latency Run-time Adaptable Binary Neural Network Classifier**: Optimised convolutional neural networks for object classification
- **Motion Planning**: A novel resource-aware algorithm for collision-free motion planning
- **Extracting Support Relations**: Understanding support relations between objects based on point clouds
- **Action Effect Prediction**: Learning to predict action effects for multiple interacting rigid and deformable objects
- **Network and Robot Co-Simulation**: Combining robot and network simulation to predict network bandwidth and latency

### 13.2 Feature Detection

We developed several strategies to map computer vision algorithms (Harris corner detection (HCD) and SIFT feature extraction and matching) to homogeneous/heterogeneous MPSoC and evaluated these algorithms on resource-aware programming models (see Chapter 1). Our results indicate that resource-aware programming helps to improve the throughput and worst observed latency of the application program along with better overall workload distribution within the heterogeneous MPSoC (see [18, 19, 20, 26]).

Executing multiple applications on a multicore system while the workload of all applications varies brings the challenge of dynamically adapting resource allocations and parametrisation with respect to constraints, e.g. timing limits of real-time applications. We investigated a hybrid approach which extracts a set of Pareto-optimal operating points during design time (see Chapter 3), which are used to dynamically parametrisate the periodic application during run time. The setup is done at the beginning of each iteration of the execution and exclusively allocates processing elements from the system depending on the current workload. The parametrisation is performed with the observed information about workload complexity and allocated resources. An oper-
ating point includes a task-to-core mapping, voltage/frequency setting, and application parameter settings, e.g. filter strength depending on the current workload. The workload is based on the complexity of the scene, e.g. uniformity of background and number of objects.

We investigated a case study based on the widely used object-detection algorithm class, specifically on a three stage algorithm, composed of HCD, SIFT feature extraction, and SIFT feature matching [26]. The various stages within the algorithm were analysed and mapped to different types of PEs based on their computational requirements. However, the presence of multiple processing elements with different characteristics raises issues related to programming and application mapping [20]. With our case study, we showed that the application may suffer performance loss and a higher worst observed latency (WOL) when the PEs on the heterogeneous processor are shared between different applications.

As a remedy, we proposed applying concepts of invasive computing as introduced in Chapter 1 to overcome these challenges. It enables the application program to adapt itself to the available resources at run time.

The impact and challenges of having an on-chip GPU was studied using TCPA, see Chapter 6, consisting of a two-dimensional array of processing elements. We demonstrated how the HCD algorithm can adapt to available resources at run time [19]. Based on the availability of resources, the application could be executed on a highly parallel TCPA (see Chapter 6), or on LEON3 CPUs (see Chapter 5). Instead of a static mapping scheme, our technique relies on the information provided by the run-time system to map the application to different types of PEs, based on their availability. The application is aware of available resources on the heterogeneous MPSoC and can adapt the workload if sufficient resources are not available. The enhanced corner detector can generate results within the specified search interval and avoid frame drops. Our experiments show that incorporating resource awareness into the conventional HCD can significantly reduce frame drops, improve its overall quality (precision and recall rates) and also constrain the worst-case latency.

The resource allocation and release happens once per frame and the additional overhead created by the resource-aware run-time system is negligible when compared to the time taken by the detector to process millions of pixels in every frame. A detailed evaluation was conducted on an FPGA-based hardware prototype to ensure the validity of the results. Feature detection is the basis for high-level robot vision tasks like object classification and segmentation, which we will discuss in the next section.
In the context of prosthetic hands, electromyographic (EMG) signals are often used to control the opening and closing of the fingers as well as the hand’s orientation. This process puts a mental burden on the user that has to manually switch between modes and learn complex muscle contraction patterns to achieve the desired grasps. To alleviate some of this burden, we have worked on semi-autonomous control schemes that utilise additional sensor information, e.g., images from an in-hand camera, to automate some of the manual processes during grasping with a prosthetic hand. In order to know which object the users intends to grasp, we need the ability to classify and segment objects in the perceived camera images. However, due to resource and battery constraints, the computing system inside a prosthetic hand needs to be both efficient and resource-aware.

In this realm, we have developed a resource-aware system for classification and localisation of objects based on visual information obtained by an in-hand camera of an intelligent prosthetic hand. The system combines an optimised object classification network with an adaptable semantic segmentation network. In case of a user confirming correct classification, segmentation can be performed class-aware using an encoder-decoder network trained on a particular class. For incorrectly classified and unknown objects the network uses weights trained on the set of known objects [10]. Highest accuracy results are achieved for object-aware trained segmentation. For unknown objects the accuracy drops but can still provide good accuracies. The major contribution of this work is the realisation of the related methods on a resource-limited embedded system. We present methods for optimisation of the network architecture and investigate resource-aware deployment enabling execution on constrained systems. Our methods enable the use of cost and energy-efficient hardware platforms, which is an important step towards the realisation of higher levels of autonomy in prosthetic hands. Figure 149 shows the KIT Prosthetic Hand and the different steps involved in the segmentation of a hammer image.

Classification is realised by a deep convolutional neural network (CNN) optimised towards inference on an embedded processor. For our application of object classification two aspects are relevant: A set of known objects must be recognised with high accuracy within given real-time constraints. We set a maximum of $\approx 150$ ms as an acceptable value for recognition.

To achieve real-time network inference, the classification network architecture is optimised using an evolutionary algorithm. The algorithm uses multiple
13.3 Object Classification and Segmentation

Figure 149: Top: In-hand camera in the palm of the KIT Prosthetic Hand. Bottom: Camera image, ground truth segmentation mask, predicted 8-bit and binary segmentation mask.

Evolution steps in which the algorithms evaluated the fitness of all networks, breeds offspring networks with crossover characteristics from two parent networks of high fitness and randomly mutates segment parameters. Since real-time constraints and given hardware resources allow a maximum number of operations at which the highest accuracy can be expected, we have modified the fitness function to target a given number of operations.

Since the convolution layers are responsible for most of the resulting operations, only the operations resulting from convolution are regarded. The number of multiply-accumulate (MAC) operations per convolution layer can be calculated as

\[ \beta_{\text{conv}} = \frac{I_H \cdot I_W \cdot I_C \cdot K_H \cdot K_W \cdot O_C}{S^2} \]  

where \( I \), \( K \), \( O \) as well as height, width and channels \( H, W, C \). \( S \) is the filter stride. We derived the fitness of a network combining number of operations and accuracy by designing a fitness function using a generalised logistic function, in detail

\[ F(\alpha_{\text{final}}, \beta_{\text{conv}}) = \alpha_{\text{final}} + (1 + e^{\beta_{\text{conv}}(1+\nu)})^{-\frac{1}{2+\nu}} \]

where \( \alpha_{\text{final}} \) is the converged network accuracy and \( \nu \) the target amount of MAC operations in millions, in our case, \( \nu = 2.0 \). The function was designed so that \( 0.0 < F(\alpha_{\text{final}}, \beta_{\text{conv}}) < 2.0 \) and strictly monotonously increases for
smaller $\beta_{conv}$ and $\nu$. Fitness strongly decreases for $\beta_{conv} > \nu$ to penalise a higher number of operations than the preferred $\nu$.

For semantic segmentation, the captured camera image as well as the user evaluated result of the classification network is used as input. The output is a binary pixel-wise segmentation of the object in the input image. The network follows the structure of an encoder-decoder network. Encoder and decoder are connected by a residual connection. This connection short-cuts a downsampling layer and a convolution-layer with reduced layer sizes. The decoder network is using deconvolution layers realised by an up-sampling layer which is concatenated by the layer before down-sampling. The concatenated layers are merged by using depth-wise convolution. The final pixel-wise result is generated by applying a binary threshold to the output image. The encoder-decoder CNN for semantic segmentation and its acceleration on a TCPA are also part of a prosthetic hand demonstrator (see Chapter 15).

![Figure 150: Average accuracy and MAC operations in millions per generation. The evolutionary algorithm successfully generates networks that have increased accuracy and lower operation count. The convergence of operations to 2 million MAC operations is a result of the chosen fitness function [10].](image)

Results of the evolutionary algorithm used for the design of the optimised classification network architecture, as average accuracy and MAC per generation, are depicted in Figure 150. The results show, that evolution as expected decreases the operation count while increasing accuracy. The operation count, according to the target count defined in Eq. (44), converges towards $\nu$. Since accuracy and amount of operations improved evenly, the fitness function does not overrate either one and is therefore beneficial. We have observed that with increasing number of generations the variance in the models is decreasing, as
network architectures tend to converge and share similar layer setups. After 10 generations, the progress significantly slows down and first generations do not increase the best fitness. This suggests that a local extremum is reached after a short evolution time. The results indicate that the chosen breeding function in fact carries on favourable characteristics. Results show that evolutionary architecture synthesis does not reach an optimal architecture, but provides a good result without manual hyperparameter optimisation. The algorithm allows reducing execution time of networks which can be created automatically, allowing the creation of user-specific network architectures for individual requirements. To optimise CNN implementations further, we will investigate binary neural network next.

13.4 Low-Latency Run-Time Adaptable Binary Neural Network Classifier

To push the limits of compute efficiency of the CNNs which enable the semi-autonomous prosthetic hand function, we investigated an implementation using a binary neural network (CNN) with run-time adaptable frequency scaling [7]. The classification task was offloaded to an efficient CNN accelerator which performs high-throughput XNOR operations on digital signal processing (DSP) blocks of an embedded FPGA. To tailor the classifier’s performance to the current application scenario, the frequency scaling approach dynamically switches between two modes of operation, high-performance and power-saving. At high-performance, classifications are performed with a low latency of 0.45ms, high-throughput of 4999 FPS (Frames Per Second), and power consumption of ~2.15 W. This enables functions such as object localisation and batch classification. Switching to power-saving mode, a latency of 80 ms is maintained, with up to 19% improved classifier battery-life. Our prototypes achieve a high accuracy of up to 99.82% on a 25 class problem from the YCB graspable object dataset [4].

The weights and activations are binarised using the CNN method proposed by Courbariaux et al. [9]. The network parameters are represented by full-precision latent weights $\mathbf{W}$ at training time, allowing a smooth training process. The input and output layers in this implementation are not binarised. This mitigates significant drops in classification accuracy.

Without loss of generality, the activation feature map $\mathbf{I}^{l-1} \in \mathbb{R}^{I_H \times I_W \times I_C}$ is considered as the input to a convolutional layer $l \in [1, \ldots, L]$ for an $L$-layer BNN. $\mathbf{I}^0$ and $\mathbf{O}^L$ are the input image and the prediction of the BNN, respectively. The latent weight matrix $\mathbf{W}^l \in \mathbb{R}^{K_H \times K_W \times I_C \times O_C}$ is composed of the trainable
parameters of layer \( l \). During the forward-pass, the weights \( w \in W \) are transformed into the binary domain \( b \in B \subset B^{KH \times KW \times IC \times OC} \), where \( B = \{-1, 1\} \). In the hardware implementation, the \(-1\) is represented as \( 0 \) to perform multiplications as XNOR logic operations. The weights and input feature maps are binarised by the \( \text{sign()} \) function

\[
b = \text{sign}(w) = \begin{cases} 
1 & \text{if } w \geq 0, \\
-1 & \text{otherwise}.
\end{cases}
\] (45)

The \( \text{sign()} \) function restricts the flow of gradients during back-propagation due to its derivative, which is zero almost everywhere. To overcome this problem and facilitate the learning process, the \( \text{sign()} \) function is approximated during back-propagation by the straight-through estimator (STE). In the simplest case, the estimated gradient \( g_b \) could be obtained by replacing the derivative of \( \text{sign()} \) with the hard \( \text{tanh} \), which is equivalent to the condition \( g_w = g_b \) when \(|w| \leq 1\) [9].

Particularly for BNNs, it is of crucial importance to normalise the input elements \( i^{l-1} \subset I^{l-1} \), before the approximation into the binary representation \( h^{l-1} \subset H^{l-1} \in B^{IH \times IW \times IC} \), by means of batch normalisation. The result of the batch-norm operation in the considered BNN architectures is followed by \( \text{sign()} \). Since the result after applying both functions is simply \( \{-1, 1\} \), the precise calculation of the batch-norm is wasteful on embedded hardware. Based on the batch-norm statistics collected at training time, a threshold point \( \tau \) is defined, wherein an activation value \( a^{l-1} \geq \tau \) results in 1, otherwise -1 [24]. This allows the implementation of the typically costly batch-norm operation as a simple magnitude comparison operation on hardware.

We evaluated the classifier on 25 objects from the YCB dataset [4], improving upon previous work by 12 objects [10]. The dataset is augmented through scale, crop, flip, rotate and contrast operations. The masks provided with the dataset are used to augment the background with random Gaussian noise. The dataset is expanded to 105K images for the 25 classes. The images were resized to \( 32 \times 32 \) pixels. The BNNs were trained up to 300 epochs, unless learning saturates earlier. Evaluation is performed on a 17.5K test set. BNN architectures shown in Table 23 were trained and implemented on hardware. Each convolutional (Conv) and fully-connected (FC) layer is followed by batch-norm and activation layers except for the final layer. Conv groups 1 and 2 were followed by a max-pool layer. The PE count indicates the number of hardware processing elements allocated per layer, while the SIMD lanes refer to the degree of data parallelism within each processing element of each layer.
The target system-on-chip (SoC) platforms that have been used for the experiments were the XC7Z020 (Z7020) for \( v \)-CNV and \( m \)-CNV prototypes, and XC7Z010 (Z7010) for \( \mu \)-CNV. All prototypes were finally deployed on a Z7020 SoC. Power, latency, and throughput measurements were taken directly on a running system. The power was measured at the power supply of the board (includes both processing system and programmable logic). Latency measurements were performed end-to-end on the accelerator covering the classifier’s total time for an inference, while throughput was determined as the classification rate when the accelerator’s pipeline is full. Note that throughput is higher than the latency rate due to the streaming architecture working on *multiple images* concurrently in different parts of its pipeline when it is full.

In Table 24, the CNV network with (1,2) and (2,2) bits for weights and activations can be compared with the fully binarised CNV (1,1) network, which achieved an accuracy of 99.82% on the YCB graspable object dataset. This shows that BNNs are well-suited for this task in terms of complexity and learning capacity.

In the bottom half of Table 24, the hardware utilisation for the prototypes proposed in [7] is shown. For the \( v \)-CNV network, a reduction of 2386 (9%)
LUTs can be observed from the regular CNV [24]. For the constrained Z7010, such reductions can make a previously non-synthesisable design realisable after moving XNOR operations to DSPs. The increase in DSP usage can be justified as they are not the bottleneck for synthesisable designs in this case. It is important to note that $\mu$-CNV was synthesisable on the Z7010 only after moving the XNOR operations to the DSPs, as proposed in [6].

Prosthetic devices used on a daily basis must offer high performance for safe and convenient use, while minimising power dissipation to increase the continuous usage time before charging. In Table 24, two values are reported (↕) for power, latency, and throughput per prototype, for high-performance and power-saving modes. At 2 MHz, $v$-CNV achieves a reduction of up to 16% in power consumption with run-time frequency scaling compared to standard CNV [24]. This translates to an improvement in battery-life of up to 19%. In high-performance mode, a latency of only 0.45 ms is consumed by the $m$-CNV network at 125 MHz. This reduces latency by 99.7% compared to the work in [10]. Considering the performance/watt efficiency metric, $m$-CNV achieves 2318 frames/watt compared to 20 frames/watt in [10]. With an optimal controller delay for myoelectric prostheses of 125 ms [5], all the classifiers in [7] consume $<1\%$ of the total time, leaving more slack for post-processing, actuators and other parts of the system. In power-saving mode, the prototypes run at 0.7–2 MHz and achieve an $\sim$80 ms latency, still leaving more than 36% of the allocated delay for the controller. It is important to note that in all the reported power measurements, roughly 1.65W of power is consumed by the Z7020’s ARM-Cortex A9 processor and the board. This leaves the isolated accelerator’s power at roughly 0.2W in power-saving mode for all configurations, making it very energy-efficient. However, we report the overall power since the accelerator is still dependent on processor calls and preprocessing. In future work, the power consumption of the board and the A9 processor can also be optimised to further reduce the classifier’s overall power requirement.

In addition to the low latency of the high-performance mode, the high throughput of up to 4999 FPS (Frames Per Second) can be used to improve the quality of the application. Instead of providing a single classification, the accelerator can pipeline many images, potentially from different sensors, and perform batch-classification. The batch classification result will represent the highest class over all classifications, which in practice compose of slightly different angles, lighting and distance to the object, improving the chances of a correct classification. Multi-camera prostheses proposed in [11] can benefit from the high throughput, as more data is gathered through the multiple camera setup.
13.4 Low-Latency Run-Time Adaptable Binary Neural Network Classifier

The high-performance mode can also be utilised to perform object localisation in multi-object scenes. A large input image can be sliced into several smaller images and reclassified [24]. The image can be reconstructed with bounded high-confidence classifications. Figure 151 demonstrates the described function.

In Figure 152, a frequency sweep is performed on the ν-CNv prototype, identifying different points of operation for different application requirements. The low-power region is considered to be below 1.90 W, while localisation would require classification rates of above 2250 FPS for an input resolution of 320×240. Batch classification can be triggered in critical scenarios where a latency of <10 ms is needed.

Run-time frequency scaling is demonstrated in Figure 153. The total power of the chip has been measured for a duration of 80 seconds. At time $t=15$, we introduced a stimulus representing the motion of the hand. The event triggers the classifier to high-performance mode for an observation period of 35 seconds. If no further event occurs, the classifier winds down to low-power mode at $t=50$. Naturally, the intermediate frequencies shown in Figure 152 can all be triggered for other scenarios or operating modes.

In summary, our contributions (see also [7]) demonstrated that BNNs have the potential to bring cutting-edge classification performance to semi-autonomous prostheses. The proposed prototypes achieved over ~99% accuracy on a 25 class problem from the YCB dataset, and a maximum of 4999 FPS and latency of 0.45 ms. The low-power mode can potentially improve the battery-life of the classifier by 19% compared to an equivalent accelerator running continuously.
at full-power. This improvement in battery-life is considered with the Z7020’s ARM-Cortex A9 processor and the board consuming a constant \(\sim 1.65\)W of power from the total reported \(\sim 1.85-2.15\) W. This leaves a lot of room for further optimisation in future work, to capitalise on the proportionally large power-savings achieved in the isolated accelerator with frequency scaling.

### 13.5 Motion Planning

In the following, we propose a novel resource-aware algorithm for collision-free motion planning. We developed a novel motion planning algorithm that employs self-monitoring concepts in order to identify the difficulty of a given planning problem and dynamically adapt resource usage based on problem difficulty and current planning progress (see [16]). We showed how dynamic adaptation of resource allocation on algorithmic level can reduce the system workload as compared to static resource allocation while meeting Quality of Service (QoS) measures such as average workload or efficiency. We evaluated our algorithm in several synthetic setups with varying problem difficulty in the context of collision free motion planning for the humanoid robots of the ARMAR family.

Figure 154 shows the implemented architecture. A user defined robot component sends a motion planning request to the distributed resource-aware motion planner. Once a collision-free motion is found, the planner sends the result back to the component. The planner consists of a central manager which spawns workers up to a defined maximum number to iteratively solve
the motion planning problem. The workers communicate their progress asynchronously to each other and the manager. In the following we briefly describe the process of dynamic adaptation to problem difficulty under the constraints of limited resource availability.

The planning algorithm is based on an improved version of RRT* [12], a modified version of the original Rapidly-Exploring Random Trees (RRT) which provides asymptotic optimality and probabilistic completeness. RRT starts with a tree consisting of one node, the start configuration, and expands the tree with configurations as nodes and collision-free paths as edges until the goal configuration is reached, i.e. is connected to the tree. The tree expansion samples a random valid configuration, finds the nearest neighbour in the tree and tries to expand the tree into the direction of the random sample. Expansion can fail if no collision-free path can be found. We define $\varphi$ as the rate of failed expansions. If $\varphi$ is low, the algorithm has no problem of exploring the configuration space. A high value of $\varphi$, on the other hand, indicates a hard planning problem where difficult paths between obstacles and free space, e.g. in narrow passages of the free space, need to be computed.

Our algorithm starts each motion planning task with an initial number of workers $\text{workerCnt}_{\text{initial}}$ and can use up to $\text{workerCnt}_{\text{max}}$ workers. The algorithm uses different strategies to determine whether more computational power is required to solve the current problem. Every strategy to be used with the planner must have the following properties: independence of and thus not posing restrictions on the planning algorithm and short computation time to only cause minimal overhead. To this end, we investigated, implemented and evaluated two strategies.
Table 24: Hardware results of design space exploration. Power is averaged over a period of 100 seconds of operation.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(8,8) - [10]*</td>
<td>400</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.446</td>
<td>115</td>
<td>9</td>
<td>96.51*</td>
</tr>
<tr>
<td>(2,2) - CNV**</td>
<td>100</td>
<td>35718</td>
<td>140</td>
<td>32</td>
<td>2.217</td>
<td>4.87</td>
<td>860</td>
<td>99.91</td>
</tr>
<tr>
<td>(1,2) - CNV</td>
<td>100</td>
<td>40328</td>
<td>131.5</td>
<td>26</td>
<td>2.241</td>
<td>1.63</td>
<td>3049</td>
<td>99.89</td>
</tr>
<tr>
<td>(1,1) - CNV [24]</td>
<td>100</td>
<td>26060</td>
<td>124</td>
<td>24</td>
<td>2.212</td>
<td>1.58</td>
<td>3049</td>
<td>99.82</td>
</tr>
<tr>
<td><strong>DSP XNOR + Frequency Scaling [7]</strong>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1,1) - v-CN V</td>
<td>111</td>
<td>23675</td>
<td>124</td>
<td>72</td>
<td>1.857</td>
<td>2.172</td>
<td>61</td>
<td>3388 99.82</td>
</tr>
<tr>
<td>(1,1) - m-CN V</td>
<td>125</td>
<td>21972</td>
<td>44.5</td>
<td>66</td>
<td>1.879</td>
<td>2.157</td>
<td>28</td>
<td>4999 98.99</td>
</tr>
<tr>
<td>(1,1) - μ-CN V</td>
<td>100</td>
<td>11738</td>
<td>14</td>
<td>27</td>
<td>1.824</td>
<td>2.028</td>
<td>80.64</td>
<td>16 1646 90.58</td>
</tr>
</tbody>
</table>

*: Running on ARM Cortex M7 (CPU frequency reported), accuracy for 13 classes, 72×72 input

**: Less PEs and SIMD lanes to fit the SoC

- The ΔTy-strategy which is solely based on the run time of the algorithm and can be parametrised by ΔT. If the condition \( T_{last} + \Delta T \leq T_{now} \) evaluates to true, additional resources are requested, i.e. a new worker process will be created. We indicate the chosen value for ΔT by replacing \( y \), e.g. ΔT5 is a strategy where ΔT = 5s.

- Since the ΔTy-strategy needs an accurate guess for the value of ΔT, the \( N\sigma\Delta Ty \)-strategy modifies ΔT depending on the rate of failed tree expansions \( \phi \). The modified time \( \Delta T' = \frac{\Delta T}{1 + \sigma \cdot \phi} \) is then used to define the threshold for a new worker \( T_{last} + \Delta T' \leq T_{now} \). The added parameter \( \sigma \) determines the influence of the rate \( \phi \) on \( \Delta T' \).

The strategies were evaluated on four synthetic setups and one realistic setup with the humanoid robot ARMAR-4 [2]. The synthetic test cases consist of four artificial motion planning problems with increasing difficulty called SerialWallsN where \( N \in 1, 2, 3, 4 \). The motion planner has to find a collision free path from a specified start configuration to the goal. Let \( t_{solve} \) be the execution time until the first solution is found and \( t_{total} \) the accumulated execution time of all workers during this period. We define the total number
of workers $W_{\text{total}}$, the average workload $W_{\text{avg}} := \frac{t_{\text{total}}}{t_{\text{solve}}}$ and use $t_{\text{total}}$ as a metric for efficient use of computation power. $W_{\text{avg}}$ measures how many workers would be necessary to solve the problem in $t_{\text{total}}$, if no thread management was involved. We compared the two strategies with simple fixed resource allocation. The results show that a dynamic adaptation of resource allocation on algorithmic level can reduce the system workload as compared to static resource allocation.

Figure 155 shows $W_{\text{total}}$ and $W_{\text{avg}}$ for all combinations of SerialWalls test cases and executed strategies. Lower values for $\Delta T$ lead to more workers being started. The $NNx\Delta y$-strategy compensates for high values of $\Delta T$ depending on the additional parameter $\sigma$. The execution time $t_{\text{solve}}$ of different strategy can be seen in Figure 156. Static resource allocation is the fastest method if enough resources are available. The dynamic resource allocation strategies run slower but show improved efficiency (see Figure 156).

Furthermore, we proposed a combined approach for robot placement and coverage path planning while considering constraints like collision avoidance and static stability [23]. In this approach, we focus on mobile manipulation tasks that require a fixed placement for executing coverage trajectory segments. The approach was evaluated in two scenarios that exemplify the broad range of possible applications: The coverage of a building facade using a robotic manlift and the coverage of an industrial conveyor belt for maintenance tasks.
Figure 155: The values of $W_{\text{total}}$ (light colour) and $W_{\text{avg}}$ (dark colour) for the $\text{SerialWalls}$ test cases and the used strategies. New resources are only allocated for harder test cases. The value of $W_{\text{avg}}$ can be less then zero because it considers work done by the manager process [16].

Figure 156: Execution time $t_{\text{solve}}$ and efficiency $t_{\text{total}}$ for the test cases $\text{SerialWalls2}$ and $\text{SerialWalls4}$ per evaluated strategy. For each test case, the values are normalised to the execution time with a single worker [16].

using the humanoid robot ARMAR-III [1]. In particular, we investigated the trade-off between run-time and achieved surface coverage, showing a linear correlation between them. This enables the prediction of run-time resources depending on the desired surface coverage and vice versa.

### 13.6 Extracting Support Relations

Given a point cloud of the scene, we want to understand the support relations between the objects in the scene, i.e. which object is physically supported by other objects [13, 21]. To achieve this, we first segment the point cloud and extract geometric primitives, e.g. boxes, cylinders, and spheres, using an approach based on RANSAC (RA nondom SAmple Consensus). By randomly choosing points from the point cloud segments, parameters for the pose
13.6 Extracting Support Relations

Figure 157: An input point cloud of the scene is segmented into object hypothesis. Then, the geometry of each object is estimated. Based on the estimated geometry, support relations between object pairs are extracted.

and shape of geometric primitives can be determined. Furthermore, the probability distribution over the parameters can be estimated by calculating the mean and covariance over all geometric primitive parameters generated by RANSAC. Using a Monte-Carlo simulation, we can estimate the support relations from the distribution over geometric primitive parameters. Figure 157 gives an overview over this process.

The different steps of the pipeline include segmentation, geometric primitive fitting and support relation extraction. Each of those steps is a parametrisable algorithms whose resource consumption depends on the input (e.g. the point cloud size) and the algorithm parameters (e.g. number of RANSAC iterations). Relevant resources include the run-time, CPU usage, and memory usage. We wanted to develop an algorithm-agnostic framework for learning resource prediction models of the different steps in the pipeline. The developed framework is able to generate different sets of input data and algorithm parameters and record the resource consumption during execution. Furthermore, the quality of the output can be measured by predefined error metrics (e.g. mean squared error to the ground truth or accuracy). This is necessary, since some algorithm parameters might reduce resource consumption but increase the error in the output (e.g. a lower number of RANSAC iterations).

Figure 158 shows our proposed framework for algorithm-agnostic resource prediction. During resource recording different sets of inputs and parameters are fed to the algorithm. Resource consumption is monitored during algorithm execution. Since the input often is multi-dimensional (e.g. a point cloud or an image), we first extract features from it. These features may be computed directly (e.g. number of points) or derived (e.g. number of objects in the scene). Finally, an error metric is computed for the produced output. The extracted features and the algorithm parameters form the training input, while the resource consumption and the error metrics form the output of the resource
prediction model. We train both linear and polynomial regression models to fit the generated data.

For evaluation, let us have a look at different trained prediction models. Figure 159 shows prediction results for Region Growing [8], a segmentation algorithm. We can see that there is a linear correlation between input point cloud size and the CPU time required for the algorithm. Furthermore, the average CPU usage stays constant independent of input size.

Figure 159: Left: Real and predicted values for point cloud size (input) and CPU time (resource). Right: Real and predicted values for point cloud size (input) and average CPU usage (resource).

Figure 160 illustrates the resource and error prediction for the geometric primitive fitting. The number of points is a feature of the input while iterations and downsampling ratio are algorithm parameters. Run-time is one of the
resources we are interested in and mean squared error is an error metric which reflects how well the geometric primitives fit to the point cloud. In the left plot, one can see that the run-time of the algorithm increases both with the number of points as well as the number of iterations as expected. In the right plot, the mean squared error decreases when the number of iterations and the downsampling ratio are increased. However, above 30 iterations and a 0.5 downsampling ratio diminishing returns seem to settle in. Support relations can be used to predict which objects might fall. In the next section, we will investigate a more general way to predict action effects.

Figure 160: Left: Prediction model that relates the number of points and iterations to the run-time of the algorithm. Right: Prediction model that relates iterations and downsampling ratio to the error in the generated output.

13.7 Action Effect Prediction

A robot needs to be able to predict the effect of its action not only on its own resources but also on the environment. To this end, we developed a data-driven, explicit action effect prediction, which can handle multiple interacting rigid [22] and deformable objects [25]. In Figure 161, the concept for action effect prediction is shown.

The core idea is to use a graph-based representation, where vertices represent objects or, in the case of deformable objects, parts of objects, and edges represent spatial relations between objects. Action parameters are encoded in the global graph attributes and used to transform the scene into an action-centric representation. For example, a push is parametrised by a direction and an end-point of the action. By using a local coordinate system, whose origin is the end-point and whose $x$-axis is aligned with the push direction. This action-centric representation is invariant to translational and rotational transformations.
Since scenes can contain a variable number of objects, an action effect prediction model is needed that is invariant to the number and order of involved objects. We identified graph neural networks [3] as a suitable model candidate, since they enable the use of graphs as both input and output of neural networks, which inherently support an arbitrary number of vertices and edges.

For data generation, different actions like pushing, lifting and opening a bag are executed in a physics simulation. The initial scene states and action parameters are randomised. The trained models have been evaluated on both simulated and real data collected on the robot. The evaluation shows that the model’s prediction accuracy is suitable for predicting action effects on real data.

### 13.8 Network and Robot Co-Simulation

Predicting and assigning run-time and memory resources as described until now is still limited to resources on a single machine. However, a humanoid robot usually consists of multiple computers, on which different software components run and communicate via the network. Traditionally, the assignment of components to computers was done manually, which required expert knowledge about communication patterns, bandwidth, and latency. Instead, we aimed at analysing automating the assignment of components to host computers in a robot system by co-simulating the robot and the network during task execution [14]. We proposed a coupling between the robotics framework ArmarX and the network simulator OMNeT++ [17] to support the evaluation and optimisation of robot architectures. Our approach allows unmodified robot components to communicate across simulated network interconnects. In a case study, we examined the influence of different hardware assignments...
of software components on task execution times. It was shown that the timing information present in the simulation-based evaluations enables more efficient hardware assignments when compared to static graph partitioning.

We created a coupled simulation in which the messages exchanged between robot components are delayed according to the topology and conditions of the simulated network. Thus, the virtual time between ArmarX and the network simulation must be synchronised. The virtual time in ArmarX proceeds in a time-stepped fashion, whereas OMNeT++ is based on the discrete-event paradigm, i.e. virtual time advances to the earliest of a set of events scheduled in the simulated future. Thus, the proposed system is a hybrid simulation. In our implementation, the network simulation manages the progress of the coupled simulation system. The simulation proceeds as follows: the network simulation executes all events up to the current virtual time of ArmarX. Now, the network simulation sends a message to ArmarX to request calculation of the next time step. Subsequently, the network simulation resumes event execution up to the new ArmarX time. During this process, messages are exchanged between the network simulation and ArmarX to trigger the simulated transmission of a message, or to signal the completion of a simulated message transmission.

In Figure 162, a typical simulation of an Remote Procedure Call (RPC) within an ArmarX time step is shown. After simulating the transmission of an RPC request, the actual RPC is performed within ArmarX. Subsequently, the calculation of the RPC and the routing of its response can be simulated.

To evaluate the accuracy of the co-simulation, we investigated cycle times of different control loops. One example is the TCPControlUnit, which controls the position and orientation of end-effector (i.e. hand or tool of the robot). If the co-simulation is accurate, we expect the cycle time distribution to show similar patterns in the real execution as in the simulated execution. In Figure 163, we show the empirical distributions of the measured cycle times aggregated over all runs for the distributed execution in comparison to the cycle times obtained in the simulation.

13.9 Conclusion and Acknowledgement

In this chapter, we have seen applications of invasive computing incorporating resource awareness, prediction, and run-time adaptation into robotic algorithms. Applications included image and point cloud processing, motion planning, action effect prediction and network simulation. Common among these aspect where the need to quantify uncertainty in run-time, memory
usage and other resources, adapting the external quality requirements and prediction. Invasive computing could directly be applied for relatively focused applications (feature extractions, CNN). But invasive concepts have been proven to be useful in an even broader context (motion planning, scene understanding and action effect prediction).

Finally, we would like to acknowledge the contributions of the following people to the topic of invasive software-hardware architectures in robotics: Dirk Gabriel, Manfred Kröhnert, Raphael Grimm, and Johny Paul.

Figure 162: Timeline of the event scheduling between ArmarX components and the OMNeT++ network simulation for a single simulation step [14].

Figure 163: Histograms with cycle times of the TCPControlUnit (left: distributed execution, right: simulation runs) [14].
References


14 Invasive Computing and HPC
Michael Bader, Hans-Joachim Bungartz, Michael Gerndt, Jophin John,
Santiago Narváez Rivas, Isaías A. Comprés Ureña

Abstract Dynamic resource management opens up a plethora of opportunities in the High Performance Computing ecosystem. The node-focused static resource management still predominant hinders the inherent dynamism embedded in applications using techniques like adaptive mesh refinement. With invasive computing, we demonstrate the advantages of using dynamic resource management in HPC application development, power management, and fault tolerance. We showcase how invasive computing improved shared memory and distributed parallelism with invasive OpenMP and invasive MPI. Additionally, we demonstrate the impact of our invasive power-aware resource manager in system-level power management. In fault tolerance, our invasive checkpointing system shows how the notion of invasive computing improves the checkpointing infrastructure.

14.1 Challenges and Opportunities

Today’s supercomputers have very static resource management. Jobs are submitted via batch scripts to the resource manager, then scheduled on the machine with a fixed set of nodes. Other resources, such as power, network bandwidth and storage, are not actively managed and are provided only on a best-effort basis. This inflexible, node-focused and static resource management will have to change in the future due to many reasons, some of them listed below.

First, applications are becoming increasingly more dynamic. Techniques such as adaptive mesh refinement, e.g. as used in tsunami simulations, lead to changing scalability properties. Furthermore, only some application phases might benefit from specialised accelerators, and I/O phases might even run best with a limited number of compute resources.

Additionally, the execution environment of applications is also becoming dynamic. Modern processors change the clock frequency according to the instruction mix as well as power and thermal envelopes. Heavy use of the vector units, for example, can lead to a lower clock frequency to stay within the thermal power budget.
As an independent concern, due to the sheer number of components, failure rates are expected to increase thus slowing down computation or even leading to an increased number of node failures.

Finally, the upcoming machines will be power-constrained, which means that the power will have to be carefully distributed among all running applications. The resulting power capping will impact the application's performance due to adaptation of the clock frequency and manufacturing variability.

These challenges in HPC will only be solvable by using a more adaptive resource management approach. For example, compute nodes need to be redistributed among running applications to adapt to changes in the application's resource requirements either due to a varying number of grid points or interspersed algorithmic phases that profit from certain accelerators; network and I/O bandwidth will have to be assigned to applications to avoid interference caused by contention of concurrent communication and I/O phases; power needs to be dynamically redistributed both within an application and across applications to enable increased efficiency. Dynamic redistribution of resources will also give more flexibility to the resource manager to schedule jobs on the available resources and thus reduce idle times and efficiency lowering contention scenarios, e.g. in the situation of big jobs waiting for execution.

Enabling the required dynamic resource management in future HPC systems requires a holistic, layered approach. It starts with the system layer being responsible for scheduling applications on the machine and dynamically allocating resources to the running applications. The overall job management including power management and checkpointing need to be adapted to the dynamic resource management. At the programming level, applications need to be programmed in a resource-aware style such that they can adapt to resource changes and can make most efficient usage of the resources. On top of the programming interfaces, programming tools have to be available that allow the application developers to analyse and tune the applications for the varying amount of available resources. At the application level, applications have to be redesigned to enable significant gains in efficiency and throughput, e.g. adaptive mesh refinement, approximate computing, and power-aware algorithms are a few aspects to mention here. Although inspired by the embedded world, the invasive computing paradigm, as introduced in Chapter 1, can be used to unlock the potential of dynamism in HPC systems, significantly contributing to its progress.
Our research group investigated the integration of dynamic resource management into HPC systems in such a holistic approach. For enabling invasive applications, novel programming concepts on the node level as well as across nodes were developed. Invasive applications are also called elastic, since they can adapt to changing resources (invade new or retreat from existing resources), either triggered from within the application or from the machine’s resource management.

As a first step towards bringing invasive computing in HPC, major HPC programming models (OMP [10, 22] and MPI [18]) were extended by designing and implementing iOpenMP (Section 14.2) and iMPI (Section 14.3). Afterwards, several applications were written in these invasive programming models. These applications needed to cooperate with the invasive resource management and thus an integration with the Slurm job management system was developed. Slurm is widely used on HPC systems for the management of jobs submitted via job scripts. The Slurm integration allows coexisting invasive and classic HPC jobs. It manages resources adaptively, i.e. changing resources of running invasive applications, to enable a better system utilisation via job scheduling, a more efficient computation via an efficiency-based resource distribution, and a novel power management. The opportunities for invasive HPC application are demonstrated for HPC proxy applications and production applications.

On top of iMPI, EPOP (Section 14.3.4), a high-level invasive programming model that facilitates the programming of invasive applications, was developed. One of the major advantages is, that the integration of new resources in form of application processes is significantly simplified by an automatic management of application phases. New processes are automatically directed to the application phase of the existing processes.

Finally, novel invasive system services (Section 14.4) that profit from the invasive resource management were developed. With the integration of interactive workload support, new application domains especially in the field of data analytics are enabled on HPC systems. The invasive HPC resource management was integrated with the dynamic resource management of Apache Spark1 and resource decision are taken according to the resource requirements of both sides and the priority of HPC or data analytics jobs.

The adaptive resource distribution is also the basis for the new invasive power corridor management. The background for this service is that HPC systems are a big manageable resource for the electricity providers. In case of a shortage or

---

1 https://spark.apache.org
an overprovisioning of carbon-free energy, the power corridor could be reduced or extended, the HPC system would have to adapt to this. The adaptation is implemented by redistributing resources between more or less power hungry applications with the goal to guarantee the power corridor.

The final invasive system service developed by us is an invasive checkpointing system. Checkpointing is the major technique in HPC for fault-tolerant execution of long-running applications. These applications save their state periodically and can restart at a specific checkpoint in case of a fault. The invasive checkpointing system iCheck is special in the way that it dynamically allocates nodes of the system for in-memory buffering of checkpoints and that it supports invasive applications with dynamically changing node numbers. Besides enabling efficient checkpointing, it also helps in a big challenge of invasive application developers, namely the data redistribution. Due to the distributed memory architecture of current HPC machines, any dynamic change of the number of nodes of an application requires to redistribute the data among the remaining nodes. This is to be implemented in application-specific code for each iMPI application. With the help of iCheck, redistribution is automated for regularly distributed rectangular domains.

While most of the work concentrated on enabling dynamic resource management and invasive programming for state-of-the-art HPC systems, iMPI was also integrated with the OctoPOS runtime system, described in Chapter 10, to demonstrate the synergistic advantages of concepts for invasive programming for the embedded world and invasive HPC applications written in iMPI. Especially the low overhead execution of run-to-completion i-lets in OctoPOS can be combined with the invasive resource management of iMPI applications on commodity parallel systems.

The remainder of this chapter is organised as follows: Section 14.2 and Section 14.3 explain in more detail iOpenMP and iMPI. In Section 14.4 we discussed how we applied the invasive computing concept to different data centre level services. Finally, Section 14.5 discusses our efforts to port our iMPI library to OctoPOS.

14.2 Invasive Computing within an HPC Node

14.2.1 iOpenMP Concepts

iOpenMP was developed as an extension of the OpenMP 3.1 application programming interface (OMP) [10, 22], which has a directive-based approach for parallelising application for multiple threads. The OpenMP runtime system is
14.2 Invasive Computing within an HPC Node

responsible for managing the set of threads, the underlying OS is responsible for mapping the threads to the cores of the processors.

In iOpenMP, the application cannot only create and destroy threads but it can be programmed in a resource-aware way and communicate the required resources. This is done via a claim object that represents the number of cores allocated to the iOpenMP application. The claim provides means to invade additional cores or to retreat from cores. In iOpenMP the number of threads is always equal to the number of cores in the claim and, at run time, the threads are pinned to the cores. The invade operation provides means for specifying constraints, e.g. a range for the number of cores, and hints, e.g. specifying the scalability of the application for different core numbers.

Furthermore, iOpenMP provides functions to specify the start and end of compute phases. Most HPC applications consist of a global progress loop simulating the time steps where the loop body consists of the numerical simulation for that time step. The functions marking the start and end of each iteration of the progress loop communicate the application’s phase structure to the runtime system and enable, for example, performance measurements. In iOpenMP, these functions are also used to automatically adapt the application’s claim to resource decisions of the resource manager.

iOpenMP comes with a resource manager that distributes the cores among the running applications. It considers the constraints in invade operations but also global optimisation objectives, such as the best utilisation of the resources. It might thus prescribe resource changes which are then implemented when the application ends a phase as described above.

14.2.2 iOpenMP Implementation

The implementation of iOpenMP is based on the iOpenMP library and the resource manager. The library interacts with the resource manager, which is a separate process in the shared memory system. The claim class is implemented as a singleton that always represents the cores allocated to the application and provides invade and retreat operations. The library also provides the start and end phase marker functions and implements the measurement of the phase execution time and other metrics of the phase which are then communicated to the resource manager. The resource manager keeps track of the running applications’ scalability and takes distribution decisions based on that information.

The interaction with the underlying Linux Fair Share Scheduler is based on the concept of CPU sets. A CPU set of an application determines the cores
14 Invasive Computing and HPC

Figure 164: Left image: Two different scenarios of several competing applications. Invasive computing leads to improved performance (in terms of overall execution time) compared to OpenMP and TBB for the execution of concurrently running shallow water simulations. Right image: Four concurrently executed shallow water simulations. The resources are dynamically scheduled.

(llogical cores for hardware multithreaded architectures) where threads can run. The iOpenMP resource manager implements the resource decisions by modifying the CPU set of the application. The parallel regions in the phase computation are all annotated such that the team of threads always equals the number of cores in the claim. In combination with thread pinning, the number of threads is constant and equal to the number of cores, and threads will always run on the same core allowing for best cache usage.

14.2.3 Applications

To demonstrate the benefits of invasive computing on a shared memory HPC node, a collection of tsunami simulations were run and the overall makespan compared. Two scenarios were used. Scenario A consists of larger simulations and longer delays between the enqueuing of individual simulations and Scenario B consists of smaller simulations and only a few seconds delay between enqueuing.

Figure 164 visualises the performance results. Enqueued applications are started in the sequential policy only when the cores are free. When using invasive computing, the jobs are immediately started on a single core and the resource manager is dynamically redistributing the resources as described above. The iOpenMP strategy demonstrates the best makespan. The right figure visualises the dynamic resource distribution decisions. White spaces represent the invasion overhead. The bar labelled with TBB in the left figure represents an implementation of the tsunami simulation with Intel’s Threading Building Blocks library. It also provides support for explicit resource management to reduce resource conflicts for concurrently started applications but the application parallelisation requires a full reimplementation. The makespan is also much better than for the sequential strategy, but not as good as exploiting resource awareness in the invasive strategy.
14.3 Invasive Computing for Large Scale HPC Systems

After adapting the invasive computing paradigm to in-node computation, the next logical step was to bring the invasive computing model to distributed-memory systems. The most widely used API for those applications is the Message Passing Interface (MPI) [18]. Thus it was decided to extend an existing MPI library implementation with new operations to support the invasive computing model, as well as develop simulations to exploit these new operations [5, 12].

14.3.1 iMPI Concepts

The development of the Invasive HPC software stack has been divided into two software components: the Invasive MPI (iMPI) and the Invasive Resource Manager (iRM). The main difference between iMPI and the MPI standard is the inclusion of 4 additional methods:

1. MPI_INIT_ADAPT: Initialises an invasive application (i.e. an application that might change its resources at runtime).
2. MPI_PROBE_ADAPT: Checks whether an adaptation is required.
3. MPI_COMM_ADAPT_BEGIN: Starts the so-called ”Adaptation Window”
4. MPI_COMM_ADAPT_COMMIT: Marks the end of the ”Adaptation Window”.

The resource management infrastructure consists of a fork of the Slurm workload manager [24] and extensions to support resource-aware applications developed with the iMPI library. The new invasive MPI operations serve as an alternative to the standard dynamic processes support in MPI. The general flow of an MPI application is different with the use of these extensions. A code snippet of the expected structure of an iMPI application that uses the proposed extensions is provided in Listing 14.1.

First, the application initialises itself in adaptive mode by calling the new initialisation operation. What follows is the application’s progress loop. There the application must check at certain intervals for adaptation instructions from the resource manager with the probe operation. In the event of an adaptation, the application needs to reach a safe location (typically at the beginning or end of a computational loop or phase), where it can begin the adaptation. Once an adaptation is completed, the application resumes making progress on its computations.
To perform an adaptation, applications rely on the creation of adaptation windows with the new begin and commit adaptation operations. The begin adaptation operation provides the application with helper communicators. Once an adaptation is complete, the application calls the commit operation, where the MPI_COMM_WORLD communicator is transformed permanently.

```c
int main(int argc, char *argv) {
    MPI_Init_adapt(&argc, &argv, &local_status);
    for (...) {
        MPI_Probe_adapt(&adapt, ...);
        if (local_status == MPI_ADAPT_STATUS.JoinING || adapt == MPI_ADAPT_TRUE) {
            MPI_Comm_adapt_begin(...);
            // data redistribution
            MPI_Comm_adapt_commit(...);
        }
        // Compute and other MPI code
    }
}
```

Listing 14.1: Expected structure of an iMPI application.

### 14.3.2 iMPI Implementation

The Invasive Resource Manager is an extension to the scalable workload manager Simple Linux Utility for Resource Management (Slurm). Slurm is a popular open-source resource manager in HPC systems. Slurm contains a collection of binaries. `SLURMCTLD` or the Slurm controller is the centralised component concerned with scheduling, tracking the nodes, and allocating the resources. `SRUN` is in charge of launching the job in the nodes, and it takes jobs from the user and asks the `SLURMD` daemon in the nodes to create processes. `SLURMD` daemon is present in each node of the partition. `SLURMD` starts a `SLURMSTEPD` daemon in each node, creating and tracking the local part of the processes. Users can submit a job through the `sbatch` command by specifying several job-specific options such as required number of nodes, wall clock time, partition etc. in a job script.

Slurm was extensively modified to support adaptation operations defined in the above section. The default `SLURMCTLD` was extended and replaced by two components, i.e. the Invasive Runtime Scheduler (iRS) and the Invasive Batch Scheduler (iBS)[4], as shown in Figure 165. The iBS has global knowledge of the available system resources and closely interacts with the iRS to request performance and power data and communicate expand/shrink decisions for running jobs.
After the iBS decides to expand/shrink a running invasive job, it is the responsibility of the iRS to enforce the operation while maintaining a consistent system state. It is responsible for changing the context of the running job step, launching new processes for an expand operation, and destroying the preexisting processes for a shrink operation. Runtime reconfiguration of an invasive application is a six-step process, each of which is shown in Figure 165. In the first step, a reallocation message is generated by the iRS and sent to the SRUN instance, which invokes the developed handler. If the adaptation operation is an expansion, then SRUN notifies SLURMD daemons of all expansion nodes to launch the required number of processes. The SLURMD daemons in the expansion nodes notify SRUN after the newly created processes are ready at the beginning of the adaptation window. In the case of a shrink operation, SRUN sends instructions to the SLURMD daemons of nodes that are
to be retreated from. Following this, each SLURMD daemon updates its local 
\texttt{MPI\_Probe\_adapt} metadata. During both expand and shrink operations, the 
state of the job is changed from \texttt{RUNNING} to \texttt{ADAPTING}. After completion of 
The adaptation operation, the leader node notifies \texttt{SRUN}. Following this, \texttt{SRUN} 
notifies the \texttt{iRS} that the adaptation was completed by sending a reallocation 
complete message. Finally, \texttt{iRS} sends \texttt{SRUN} the updated job credentials and 
updates the job state back to \texttt{RUNNING}.

### 14.3.3 Applications

To investigate the advantages of invasive computing on HPC systems, we 
selected and implemented multiple large-scale applications that can potential-
tially benefit from runtime resource changes [12, 13]. Here we present one of 
them.

**Large-scale invasive 2D tsunami simulation**

Our implementation of an invasive tsunami simulation is based on \texttt{sam\(\text{(oa)\text{\textsuperscript{2}})}\)}, an existing framework for parallel simulation with adaptive mesh refinement 
(AMR) [16, 17]. This fluid-dynamics application is an ideal candidate for 
the invasive paradigm, because it is a real-world scientific application repre-
senting the class of tightly coupled parallel applications that typically have 
spatially-dependent data and a non-trivial communication topology [7]. Its 
simulation domain is discretised with a structured triangular grid that refines 
and coarsens in various local regions at every simulation time step, resulting in 
a changing computation workload at application runtime. The dynamic nature 
of the adaptive mesh is accomplished with the use of Sierpiński Space-filling 
curves, which map from a 1D to 2D domain created by recursive substructuring 
of right, isosceles triangles [1, 23].

The tsunami wave propagation can be modelled using the Shallow Water 
Equations. This is due to the fact that, although a three-dimensional domain 
is considered, the wavelength of a tsunami typically largely exceeds the water 
height. Thus, in the end, the whole system can be expressed in the general 
form of a balance law as

\[
q_t + f(q)_x + g(q)_y = \Psi(x), 
\]  

(46)

where
Equation (46) is a hyperbolic system which describes the conservation of mass and momentum. The water height $h$ represents the mass, and $hu$ and $hv$ the momentum in the two horizontal dimensions. $g$ represents the gravitational acceleration and $b$ the bathymetric data.

There are several methods for solving the unknown $q$, especially for SWE [2, 8, 9]. The tsunami simulation in sam(oa)$^2$ uses the finite volume approach by [15].

A high-level invasive algorithm to implement the previous model is shown in Algorithm 3. In case of resource expansion, the joining processes bypass the initialisation phase and directly join the resource adaptation block, in which they are initialised with necessary data and objects, and synchronised with the preexisting processes. After returning from the adaptation block, they reach the tsunami simulation loop and join for the next iteration. In case of resource reduction, data must be transferred out from the leaving processes to the staying processes.

Figure 166a and Figure 166b show the simulation of the Tohoku 2011 tsunami and the underlying grid at two simulation time steps. One can observe that the underlying grid varies significantly at these time steps. Indeed, it adapts according to the propagation of the wave. Different colour blocks in the grid denote different processes.

In order to analyse the impact of resource adaptation on the application runtime, we made test runs with a single instance of the invasive tsunami simulation in a 32-node environment on SuperMUC Phase I. The test always starts with the minimal 1 node (16 processes), and can maximally expand to 32 nodes (512 processes). We use a random scheduler that makes a random resource change decision every 60 seconds. We break down the execution time into components and make measurements of each one. The resource profile of one of the test runs is shown in Figure 167a. Four resource adaptations took place in this test run—two resource expansions and two reductions. The total time spent on resource adaptation in this run is measured to be 5.7% of the total execution time. All test runs have an average total adaptation time of 5 to 10% of the total execution time.
Invasive Computing and HPC

Figure 166: Results of the benchmark simulation of the Tohoku tsunami from 2011. The colour blocks in the grid correspond to the colour of the executing nodes, e.g. the blue portion of the grid is computed by the node in the same colour. Nodes in grey colour are not utilised. Taken from [12].

**Algorithm 3** Invasive tsunami simulation.

```java
function main
2: MPI_Init_adapt(…)
3: if local_status == JOINING then
4: resource_adaptation()
5: else
6: Initialise and distribute grid
7: Read input data
8: while … do
9: Grid refinement
10: Load balancing
11: Time step computation
12: if time to probe iRM then
13: resource_adaptation()
14: MPI_Finalize()
15: function RESOURCE_ADAPTATION()
16: MPI_Probe_adapt(…)
17: if adapt_flag == TRUE then
18: MPI_Comm_adapt_begin(…)
19: Data redistribution
20: MPI_Comm_adapt_commit(…)
21: Update MPI info
22: 
```

Figure 167b shows the number of MPI ranks versus execution time for two runs on the same 32-node-environment. In one, the number of ranks was fixed at 512, while in the other dynamic adaptations were allowed. The integral of this curve over time represents the CPU-hours of the run. One can see that even though the iMPI run took longer to completion, its CPU-hours is almost reduced by 50% of that of the static MPI run. This shows that, although the static MPI application finished earlier, the invasive simulation used its resources more efficiently, reducing the CPU-hours almost by 50%.

### 14.3.4 Elastic-Phase Oriented Programming Model

The development of iMPI applications can be challenging for programmers. One needs to consider the possibility of new processes, their interaction with the existing processes, sharing the application data between the existing and new processes, and finding the appropriate entry point for the new processes.
in the application. The Elastic-Phase Oriented Programming model (EPOP) [14] simplifies this application development process by providing the concept of application phases. An **Init** phase represents the initialisation part of an application, an **Elastic** phase represents a compute-intensive part of the application that can benefit from resource adaptation, a **Rigid** phase represents parts of an application that does not need resource adaptation, and a **Branch** phase to switch between different phases.

A simple EPOP version of the application is in Listing 14.2. The EPOP driver, which is in charge of control flow in EPOP applications, will call iMPI routines in the background (not shown in the listing) to make it invasive. During the application run, the EPOP driver will call these phases in the order specified by the programmer. Additionally, EPOP will perform the iMPI calls associated with resource adaptation such that the application developer does not need to concern about the complex control flow created by the iMPI application.

```
1 void init_block(...) {/* initialisation code*/}
2 void elastic_block(...) {/* compute block*/}
3 bool loop(...) {/*loop condition elastic block*/}
4 void adapt(...) {/* data redistribution*/}
5 void finalize_block(...) {/* finalisation*/}
6 extern "C" phase_vector epop_program()
7     setInit(init_block);
8     setElastic(elastic_block, loop, adapt);
9     setRigid(finalize_block, ...);
10 }
```

Listing 14.2: Pseudocode of a sample application using EPOP.
14.4 Data Centre Level Invasive Services

High Performance Computing (HPC) systems are primarily used to run computationally intensive scientific applications. A critical component in modern HPC systems is a middleware called Resource and Job Management System (RJMS), which is responsible for efficiently distributing the computing resources among the applications and mapping the submitted jobs to the underlying hardware. Every application would have different node requirements, and the scheduling often results in multiple nodes remaining idle. Many research works are out to improve system utilisation by developing better resource managers and scheduling strategies. Nevertheless, applications may not utilise all idle nodes due to scalability limits. This complex issue can be tackled using the application and system-level dynamism elaborated in the coming sections. Section 14.4.1 shows how dynamism allows combining high performance analytics with MPI for better system utilisation, Section 14.4.2 shows power management using dynamic techniques. The last Section 14.4.3 demonstrates the usage of dynamism in our fault tolerance infrastructure.

14.4.1 Enabling Interactive Workloads

Apart from improving the job scheduling strategy for HPC systems, another opportunity to optimise node usage is to assign idle nodes to data analytics applications.

Concepts

Scientific applications running on HPC platforms generate a huge amount of data as output. This data often needs to be cleaned, sorted and indexed before being used for research. Furthermore, this data needs to be analysed manually by domain experts or processed by data analytics applications to obtain significant observations and derive conclusions. Consequently, these conclusions often lead to improvements in the scientific applications that can generate better data in the future. This indicates that data analytics applications and scientific applications are highly co-dependent. Therefore, it would be beneficial to run data analytics applications on the same HPC systems where scientific applications are also being executed. The presence of idle nodes while executing scientific applications and the advantages of running data analytics applications on HPC platforms will also result in a better utilisation of HPC resources.
To demonstrate the effectiveness of such a strategy, we have integrated the iRM and Apache Spark, thereby exploring the possibilities of a new application area, dynamic high performance analytics [3].

**Implementation**

iRM and Apache Spark work in tandem to maintain efficient system utilisation. We have extensively modified iRM, extended Apache Spark, and created two new scheduling strategies in iRM for dynamic resource management between MPI and Spark applications.

**Idle Node Filling scheduling strategy:** iRM will schedule MPI applications based on the existing scheduling strategy in this technique. Then it identifies the idle nodes and allocates them to Spark applications. The number of idle nodes can vary over time based on the number of resources used by MPI applications. A higher priority is assigned to MPI applications, and the Apache Spark is forced to use only the idle nodes.

**Performance-based scheduling strategy for Spark:** Spark sends the status information (for example, the number of drivers that are not launched due to lack of resources and the number of pending tasks) of currently running Spark applications to the iRM. iRM decides the resources (number of nodes) needed for the spark applications based on this information. For example, more resources are allocated for Spark if the number of waiting drivers increases over time or if an application’s pending tasks are above a threshold (selected based on multiple runs). If all the Spark applications are complete and there are no more waiting drivers, iRM takes the nodes back. Spark applications are assigned a higher priority in this scheduling strategy, and MPI applications can only run on the idle nodes not used by Spark.

The iRM communicates with the Spark master to exchange resource information and performance data. Based on the selected scheduling strategy, iRM identifies the nodes that can be assigned to Spark and MPI applications. A list of hostnames corresponding to the allocated nodes and the number of nodes assigned to each Spark application is then sent to Spark.

**Application Results**

Results from the Idle node scheduling strategy can be seen in Figure 168a. We can observe that the resource allocation for Apache Spark is modified dynamically. Initially, a spark application that performs generalised linear model regression and classification tests was launched on 34 nodes. Later, we launched an MPI application (Laplace solver) with 4 nodes. When the
MPI application starts, the resources are taken away from Spark and given to MPI. The number of nodes allocated to Apache Spark shrunk to 30. When we launched another MPI application (Poisson solver) on 32 nodes, the resources allocated to the Spark application were further shrunk to 2 nodes. We can also observe that Apache Spark resources are expanded whenever idle nodes are available. As we can see from the figure, the nodes are always occupied by either MPI applications or Apache Spark, improving the system utilisation.

### 14.4.2 Power Corridor Management

Electricity contracts between energy companies and compute centres are written in accordance with the so-called power corridor. Thus, the power consumption must be bounded by certain upper and lower limits. If the computing centre goes beyond those limits (i.e. if it consumes less or more than what is stipulated in the contract), the energy company could apply some penalties. Alternatively, the computing centre can also be a power stabiliser for the grid load. This means that dynamic adaptations of the power corridor might be part of the electricity contract and could be requested by the electricity company. As a result, the computing centre will obtain economic incentives such as reduced electricity costs. It is possible to use well-known techniques such as power capping to enforce the upper power limit. Nevertheless, these cannot be used to enforce the lower limit (i.e. increase the system power consumption). In such scenarios, we can utilise invasive computing for system-level power corridor management.
Concepts

The core tenet of employing dynamic resource management for power corridor management lies in the power usage characteristics of the applications. Power usage of the applications varies based on the application characteristics. We can loosely categorise the applications into high-power or low-power applications. During runtime, we can redistribute the resources among high and low-power applications to manipulate the overall power usage of the system. The primary requirement for such a system is to have a power-aware resource manager and invasive applications. Therefore, we have aggressively extended iRM with power-aware scheduling and resource management capabilities [4, 14, 19].

\[
\begin{align*}
\text{Minimise :} & \quad f(k_{idle}) = k_{idle} \times p_{idle} \\
\text{Subject To :} & \quad l \leq \sum_{i=1}^{K} k_i \times p_{(i)}^{min} + k_{idle} \times p_{idle} + m_j \times p_{(j)}^{min} \\
& \quad u \geq \sum_{i=1}^{K} k_i \times p_{(i)}^{max} + k_{idle} \times p_{idle} + m_j \times p_{(j)}^{max} \\
& \quad k_{\text{min}} \leq k_i \leq k_{\text{max}}, \quad k_i \in \mathbb{N} \setminus \{0\}, \quad i = 1, \ldots, K \\
& \quad 0 \leq k_{idle} < N, \quad k_{idle} \in \mathbb{N}
\end{align*}
\]

The redistribution of nodes among the applications to maintain the power corridor by the resource manager can be formally expressed in Eq. (47). The objective of this Linear Programming (LP) model is to improve system utilisation, i.e. reduce the number of idle nodes in scheduling scenarios. The LP model uses running and waiting jobs for analysis.

In Eq. (47), \( K \) represents the number of running applications, \( k_i \) represents the nodes required for the \( \text{job}_i \), \( N \) represents total number of nodes, \( m_j \) represents the nodes required by the \( \text{job}_j \) from the waiting queue in the order of priority, \( k_{\text{min}} \) and \( k_{\text{max}} \) represent the minimum and maximum values possible w.r.t. the --node-constraints of the invasive job, \( p_{\text{min}} \) and \( p_{\text{max}} \) represents the minimum and maximum power consumption of the job, and \( k_{\text{idle}} \) represents the number of idle nodes in the system consuming \( p_{\text{idle}} \) power. Finally, \( l \) and \( u \) represent the lower and upper power boundary respectively.
Implementation

In the beginning, the iRM obtains information about the current jobs and available resources in the system. Initially, all jobs are launched according to the minimum and maximum power values per node mentioned by the user. After the launch, the iRM periodically updates the power values associated with each running job and looks for a power corridor violation.

In case of a power corridor violation, we utilise the LP model shown in Eq. (47) to enforce the power corridor. The LP model uses running and waiting jobs to generate a new resource configuration that satisfies the power corridor. We iterate over the waiting jobs in priority order and use them along with the running jobs as an input to the LP model. The first feasible resource redistribution configuration generated with the waiting job is chosen, and the running jobs are reconfigured (expand or shrink) with this new configuration. Additionally, the selected waiting job is launched.

We check the available power budget and start the appropriate jobs (jobs that maintain the system-level power budget) in priority order if there is no power corridor violation.

Application Results

We utilise two iMPI applications with different power requirements to analyse our dynamic power corridor management techniques in SuperMUC. The first application repeatedly calculates the value of PI and the second solves the 2D heat equation using the Jacobi iteration. The average power consumed by the PI and the heat applications is 170 and 250 watts per node. The nodes required by the applications range from 1 to 4. Our workload consists of 20 jobs with equal distribution among the two applications and is submitted to the iRM alternatively with an inter-arrival time of two seconds.

We consider three scenarios to analyse our power corridor management strategy. In the first scenario, we use the static backfill scheduler in Slurm with no resource redistribution. We utilise the LP model to generate node redistributions for scenarios 2 and 3. However, in scenario 2, only running jobs are considered for the redistribution, while in scenario 3, waiting jobs are also taken into account. We dynamically change the values of the power corridor during the evaluation.

We can observe the number of power corridor violations in different scheduling scenarios in Figure 168b. Scenarios 2 and 3 are successful compared to scenario 1 in system-level power management. Initially, both scenarios (2 and
3) maintained the power corridor through resource reconfigurations. When the power corridor was decreased dynamically, scenario 3 maintained the power budget by generating a new feasible resource reconfiguration using the LP model and launching a waiting job from the queue. On the other hand, no such feasible resource reconfiguration was found for scenario 2 using only the running jobs.

14.4.3 Invasive Checkpointing

Fault tolerance is vital in High Performance Computing. As the peak system performance breaks the exascale barrier, the expected mean time between failures is minutes. Thus, it is essential to be able to restart the application from the state before the failure. This requires saving the application state periodically (checkpointing), restarting the application, reading the saved state and continuing computation from the saved state. This led to the development of the invasive checkpointing system (iCheck). It combines fault tolerance and dynamism to create an adaptive fault tolerance interface for standard MPI and iMPI applications.

Concepts

iCheck is an adaptive software system dedicated to providing checkpointing services to applications. Its design objective focuses on providing resources (memory, endpoints) for checkpointing based on the demand from the applications. iCheck can scale horizontally and vertically as per the dynamic needs of the system. iCheck has a core system and a Checkpoint/Restart library (See Figure 169). iCheck runs on compute nodes of the HPC system and communicates with the application on the machine via the high performance network. Using the provided library, the application contacts iCheck for checkpointing.

Figure 169: Components of iCheck.
The core system consists of three components: Controller, Manager, and Agent.

**Controller:** The controller performs multiple roles in the iCheck ecosystem. The primary role is to perform the manager selection and agent scheduling tasks. The monitoring framework in the controller analyses several metrics (e.g. bandwidth, available memory, checkpoint frequency) and aids in deciding the number of agents along with managers in which those agents will be launched. The secondary role is to interact with the iRM. iCheck can be configured as a standalone system or as being tightly coupled with iRM. If configured, the controller can request additional nodes or relinquish the existing nodes to iRM.

**Manager:** It manages the node-level activities of the software, such as launching the agents, monitoring and predicting the node usage parameters (e.g. memory usage, bandwidth usage), and interacting with the controller. The manager frequently updates the controller with the node status information from the monitoring infrastructure.

**Agent:** An agent is the most elementary module in the architecture. The agent performs the core functionality of checkpoint read/write. The communication between the agents and an application is performed through Remote Memory Access (RMA) operations and TCP/IP. An agent supports both, blocking and non-blocking checkpoint operations. After a checkpoint operation, the agents send status information to the manager using shared memory.

**Implementation**

iCheck provides a minimalistic API for application-level coordinated checkpointing (see Listing 14.3). A minimum of six iCheck API calls are needed for successfully writing and reading a checkpoint, two for configuration, two for checkpointing, and two for restarting. The function *icheck_init()* registers the application with the iCheck ecosystem while *icheck_finalize()* will deregister it. *icheck_add()* records the data, the type, and size to be checkpointed. The *icheck_commit()* will trigger the RDMA operations and transfer the checkpoint to the agent. *icheck_restart()* and *icheck_restore()* are used during the application restart. Additionally, iCheck provides special API calls to support dynamism and performance improvement. For example, *icheck_probe_agents()* will query the controller for any change in the agent configuration.

iCheck uses RMA for checkpoint data transfers. There are two RMA operations, namely read and write, and iCheck uses a combination of these to perform
14.4 Data Centre Level Invasive Services

checkpoint storage and retrieval. The two strategies that iCheck provides are push and pull. In the push technique, the application transfers the checkpoint to the agents, while in the pull technique, the agent initiates the checkpoint transfer from the application. iCheck uses the libfabric [21] library for RMA operations. Additionally, iCheck can also use OpenMP to parallelise the checkpoint data transfer process.

Non-Blocking Checkpointing: iCheck, with its agent-based checkpoint retrieval model, easily integrates the asynchronous checkpoint retrieval capability in its library. Since the agents use RMA, the application does not need to use its computing resource for data transfer, but it can rather continue the execution immediately after notifying the agents about the checkpoints. The agents can remotely retrieve the data without affecting the application performance.

```c
#include<iCHECK.h>
int main() {
    MPI_Init(NULL, NULL);
    float data[SIZE];
    iCheck_init("appname", ...);
    iCheck_add("Array", data, SIZE, float);
    if(checkpoint_available) {
        iCheck_restart();
        iCheck_restore("Array", data, SIZE);
    }
    for(i = 0; i<N; i++) {
        /*Read/Write data[]*/
        iCheck_commit();
    }
    iCheck_finalize();
    MPI_Finalize();
}
```

Listing 14.3: Pseudocode of a sample application using iCheck.

Multilevel Checkpointing in iCheck: iCheck stores the checkpoint in agents memory in the compute nodes as well as in a parallel file system. The parallel file system acts as a second level in our multilayered checkpointing approach. iCheck writes the checkpoint data into the parallel file system during its idle time. This data can be used in case of an agent failure or checkpoint corruption inside the agent. Additionally, iCheck agents will perform the data compression before writing into the parallel file system.

Dynamism in iCheck: One of the key characteristics of iCheck is that it is an adaptive checkpointing system. iCheck is adaptive in two aspects. Firstly, it can reconfigure the resources (add/remove agents) associated with the
connected application. If enough resources are available, the system can dynamically increase the number of agents assigned to the application. This functionality makes iCheck compatible with iMPI and can cater to dynamic applications’ needs by allocating new agents during a resource expansion. Also, an application does not need an invasive MPI to utilise the dynamism offered by iCheck. For example, the applications can use the icheck_probe_agents() function to query for change in its agent configuration. Secondly, it can horizontally scale its resources (available memory, number of agents) by attaching more manager Nodes. Both of the above reconfigurations can be made dynamically during the life-cycle of the application and the iCheck system, thereby making it a fully adaptive system. Hence, it provides the opportunity to dynamically improve and optimise the checkpointing process.

Application Results

The evaluation of iCheck is performed on SuperMUC-NG. The iCheck system is assessed with the highly scalable molecular dynamics application ls1 mardyn [20]. For comparison, we used the in-house checkpointing using MPI-IO available in the ls1 mardyn. We ran the ls1 mardyn application on one to twelve nodes (48 process per node). Each process checkpoints information about molecules like id, position, momentum, velocity, etc. For data transfer tests (comparison with MPI-IO), we ran ls1 mardyn for 100 time steps with a checkpoint interval in every tenth-time step. For all the tests, the simulation system was initialised with 65536 molecules.

Figure 170a and Figure 170b depict the average time taken for a checkpoint write operation performed by iCheck and MPI-IO during the execution of ls1 mardyn. Figure 170a shows the time (in log scale) taken by iCheck with agents varying from one to twelve and MPI-IO during the application run on twelve
nodes, while Figure 170b plots the iCheck and MPI-IO time while running the application on one to twelve nodes with a single agent and manager. It can be observed from both figures that the iCheck performs better than MPI-IO versions in all the configurations. A comparison with the worst-performing iCheck agent combination in the iCheck with the best-performing MPI-IO in Figure 170a shows that iCheck is 400 times faster in writing checkpoints than MPI-IO. The best iCheck configuration performs 5000 times faster than MPI-IO. This considerable performance improvement can be attributed to the adaptive RMA-based in-memory checkpointing model in iCheck.

14.5 Integrating iMPI with the x86 Native OctoPOS Platform

The iMPI extensions described in Section 14.3 allowed us to run invasive applications on top of HPC systems. iMPI was also integrated with the OctoPOS runtime system to demonstrate the advantages of combining iMPI with OctoPOS. This integration reduces the gap between the invasive HPC and the more standard invasive computing approach which mainly uses OctoPOS as development platform. A description of OctoPOS can be found in Chapter 10.

14.5.1 Approach

OctoPOS was selected for three main reasons. 1. It is an operating system conceived for invasive applications, which makes the iMPI concepts fit naturally. 2. It is a system that supports multiple hardware architectures, including x86. 3. There was already a basic implementation of MPI.

The biggest difference in terms of design of this port with respect to the original iMPI is that in the latter all the responsibility on when and where an adaption shall occur lies with the resource manager. On the OctoPOS version we have introduced MPI_Resizer_request, which allows applications to explicitly request more or fewer resources. Besides that, the API is exactly the same as exposed in Section 14.3.1. Additionally, an initial prototype of a resource manager has been created for OctoPOS.

As mentioned earlier, OctoPOS already had a basic non-feature complete MPI implementation. Nonetheless, it was originally designed to run the NAS Parallel Benchmarks (NPB)\(^2\), a small set of programs designed to evaluate the performance of parallel supercomputers. This implied that the library was designed considering only static applications.

\[^2\] https://www.nas.nasa.gov/software/npb.html
14.5.2 Implementation

We have added the four iMPI function calls defined in Section 14.3 and a new function MPI_Resize_request to transform MPI into iMPI in OctoPOS. Through the MPI_Resize_request function, an application can ask for more resources to the iMPI library. Therefore, in OctoPOS iMPI, the iMPI library must perform the necessary process creation and management tasks. Since the library cannot have a global overview of the system resources (for example, how many applications are running, number of available cores) to make the adaptation decision, the application’s responsibility is to request the dynamic resources via the MPI_Resize_request function.

The control flow of standard MPI and iMPI calls remains the same (as in Listing 14.1), except for the additional call placed by all the processes to request the resource. Upon receiving a resize request call with a new size, the iMPI library will mark the application for adaptation. This, in turn, will trigger the MPI_Probe_adapt and later the creation of adaptation windows using MPI_Comm_adapt_begin and MPI_Comm_adapt_commit.

iMPI uses the claims under the hood to perform the dynamic resource reconfiguration in OctoPOS. Unlike standard iMPI, new MPI processes are created inside the MPI_Comm_adapt_begin method during a resource expansion. The main process of the application will acquire the claims for the new set of processes from OctoPOS and launch new processes in those claims. During a reduction, upon reaching the MPI_Comm_adapt_commit, the processes marked for reduction exit from the application, and the main process will destroy the claims associated with those processes.

14.5.3 Applications

MiniMD

MiniMD is a proxy C++ molecular dynamics simulation code developed in the Mantevo project at Sandia National Laboratories [6]. It has a relative small size compared with other Molecular Dynamics (MD) simulation codes, as LAMMPS or lsl mardyn. It is as well mostly self-contained, reducing the dependencies from third party libraries. All of that made it a good candidate to be ported to OctoPOS.

Molecular dynamics is a type of $N$-body problem, in which $N$ bodies with positions $x_i$, velocities $v_i$ and masses $m_i$ for $i = 1, \cdots, N$ interact. Newton’s equations of motion are used to advance the system in time. The accelerations $\vec{a}_i$ are determined from the forces acting on each particle via Newton’s second
law. Time integration schemes need to be used to solve this system. MiniMD uses the so-called Leapfrog Algorithm [11].

MiniMD defines two different types of interactions between particles, the Lennard-Jones and the Embedded Atom Model (EAM). These determine how the forces are calculated.

The Lennard-Jones potential is composed of a steep repulsive term that acts at close distances, and a smoother attractive term at long distances. Equation (48) shows the 12-6 Lennard-Jones potential, one of the most widespread variants. There \( r_{ij} \) denotes the distance between the two particles, \( \sigma_{ij} \) and \( \epsilon_{ij} \) denote parameters of the potential, which depend on the type of particles i and j.

\[
U_{ij}^{12} = 4\epsilon_{ij} \left( \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left( \frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right)
\]  

(48)

To be efficient MiniMD uses spatial decomposition, so the simulation domain is divided into subdomains, and each one is assigned to a different processor. Each processor is then in charge of calculating the new position and forces of the particles that belong to its respective subdomain, as well as interchanging values required by its neighbours processors.

Several modifications were performed in order to adapt MiniMD to OctoPOS. For example, the original code used a Cartesian topology via MPI_Cart_* functions. Since those are not available on the current MPI implementation, it was necessary to adapt the code to use custom functions. Additionally, we use the TCP functionalities provided by OctoPOS to gather and visualise the results.

**Heat**

A second application to simulate heat transfer on a 2 dimensional rectangular system was developed. This simulation is based on a diffusive process, as expressed in Eq. (49). There \( \Phi(x) \) represents a quantity which diffuses (i.e. temperature \( T \)), and \( k \) is a diffusion constant.

\[
\frac{\partial \Phi}{\partial t} = k \nabla^2 \Phi
\]  

(49)

We focus on the analysis of the steady-state, for which \( \frac{\partial \Phi}{\partial t} = 0 \), leading to the Laplace equation \( \nabla^2 \Phi = 0 \). It is then possible to find a numerical solution for this system via Finite Differences.
The application developed uses a rather simple Jacobi solver to find the solution of the previous problem. Jacobi was chosen for two main reasons: it is simple to implement, and it can be solved in parallel. The latter was important to us, as we wanted an embarrassingly parallel problem to pose as best-case problem for iMPI.

On each timestep the application checks whether an adaptation is bound to happen, and if so the data is redistributed among the leaving or joining processors. Since at the moment of writing there is no equivalent to a resource manager on OctoPOS, the adaptation is triggered with a call to MPI_resize_request. For visualisation, we transmit the data in a similar fashion as with MiniMD.

At the time of writing, both the Heat application as well as MiniMD are not yet fully functional. We hope to discuss the obtained results once the development is finished in a future publication.

### 14.6 Summary and Outlook

This chapter introduced the concepts and tools for enabling dynamic resource management for HPC systems. Since this is a disruptive change for HPC, novel techniques are required on all levels of the HPC software stack. The work presented in this chapter covers all the layers from the system level up to the application level.

The next steps are to integrate the concepts of iOpenMP and iMPI into the OpenMP and MPI standards. Furthermore, production-ready implementations of the extensions and the system-level support in Slurm are required. Significant work will also be required to facilitate the transformation of current static MPI applications into invasive applications. This not only requires new algorithms, but also to make the resource changes easier.

A further area of research is to enable not only the flexible redistribution of entire nodes, but to integrate into the concepts support for co-scheduling of application on the nodes. This would then enable to flexibly, for example, distribute entire accelerators or parts of accelerators of the node to the co-scheduled applications for certain program phases.

An area that our research did not cover yet is the impact of invasive resource management on the programming tools. All performance analysis and tuning tools used in HPC are based on a static set of resources allocated for exclusive use of the application. When this resource set becomes dynamic, the major assumption of reproducibility of the performance of application runs is broken.
This will make jitter more frequent, and differences between runs will be a common scenario. At least the techniques developed in our research are based on changing, but at the same time dedicated, resources for the application. Hence, the undesirable impact of jitter, namely creating performance problems, is eliminated.

References


15 Validation and Demonstrator

Jürgen Becker, Frank Hannig, Thomas Wild, Marcel Brand, Oliver Lenke, Fabian Lesniak

Abstract This chapter describes concepts and implementations linking all invasive computing contributions including theory, language and compiler design, operating system, and hardware architecture design together on an FPGA-based prototyping platform. After introducing a multi-FPGA prototyping platform, the concept of a modular invasive multi-tile architecture that is configurable in terms of size and composition of tiles is presented, which is mapped to the modular architecture including the configuration of peripherals, memories, and debugging support. The remaining part of this chapter is dedicated to the introduction of different demonstration scenarios that have been developed including an interdisciplinary demonstrator for runtime requirement enforcement of performance and security, as well as power and thermal aspects of a soft real-time video object detection application as well as a demonstration of a prosthetic hand machine learning application.

15.1 Introduction

In the previous book chapters, the fundamental concepts of invasive computing have been introduced, and various contributions to system-level and application-level software as well as invasive hardware components have been described. This chapter covers the actions undertaken to show the advantages of invasive computing in general and to validate and demonstrate the effectiveness of the mentioned contributions in tailor-made invasive computing architectures in particular. The demonstration of invasive hardware architectures would not have been feasible with commercial off-the-shelf compute platforms. Therefore, a commercial multi-FPGA prototyping system was used as the underlying system. The integration of all contributions into an invasive processing architecture and its mapping to the prototyping platform is the major focus of this chapter.

Evaluating a new computing paradigm like invasive computing in a graspable manner is challenging. The improvements of invasive computing become apparent when running applications on a real system and not only via simulations (e.g. using the X10-based simulator InvadeSIM [4, 11]). The major difficulty comes from the fact that on a prototype all constituent parts, be
it hardware or software, have to actually interoperate in an error-free and efficient manner, which does not allow abstracting out details. Further, it was also a challenge to integrate contributions provided by multiple research groups from very different domains into an operational prototype, which covers the whole computing stack (Figure 171).

Starting from the hardware layer, it was necessary to integrate the developed compute resources, e.g. invasive tightly-coupled processor arrays (TCPAs [3], see Chapter 6) or i-Core extensions to the used LEON3 processor cores [1] (see Chapter 5), a new communication architecture to enable invasive inter-tile communication (see Chapter 9), as well as an appropriate memory hierarchy. For all of this, appropriate OS support was required, which has to cooperate with middleware layers like the X10 runtime system and the agent system. Not to forget the applications that have to be programmed in a way to comply with the lower levels of the stack and to provide a compiler framework to support the new programming paradigm. All mentioned components deliver specific features contributing to the invasive computing approach.

Combining all contributions from invasive computing into a common prototype requires well-established procedures for the integration in terms of the design flow as well as for verification purposes. Further, appropriate hardware and software interfaces must be provided to support the verification, the evaluation and the visualisation of measured metrics. Moreover, and most importantly, proper demonstrators have been set up that have helped to showcase the value of the different hardware and software contributions.

The following sections introduce the underlying FPGA prototyping system making up the basis for the integration and demonstration efforts. Then, the invasive computing architecture with its hardware constituents and the various configuration possibilities that can be realised on the FPGA platform.
is described. The flow of design integration and verification is covered in a subsequent section. Finally, the major demonstrators and the associated application scenarios running on the FPGA system are described.

15.2 FPGA Prototyping Platform

The invasive computing demonstrator platform is based on an FPGA prototyping system by ProDesign called proFPGA. At each site (FAU, KIT, TUM), an individual proFPGA platform instance was used. All three systems were operated consistently to ensure the coherent implementation of all the individual components, the proper integration of the contributions by the architectural research (see Part ), and to provide testbeds for experimentation and measurements of specific aspects as well as the whole compute stack. The resources for prototyping invasive multicore architectures consisting of up to 80 RISC processor cores are available by utilising the four interconnected FPGAs of the proFPGA platform. It consists of four Virtex-7 2000T FPGAs, providing a total capacity of up to 48 million gate equivalents [13]. All four FPGA modules are mounted on a mainboard [13], which provides the power supply for FPGAs and extension boards as well as a common interface for configuration and debugging. An FPGA module contains the FPGA itself and four FMC connectors on both the top and the bottom side. The top side connectors can be used to attach extension boards, if required. While the bottom connectors connect the FPGA module with the motherboard, most of the pins are passed through to connectors on the rear side of the motherboard. These connectors are used for ribbon cables to connect the FPGA modules among each other. The cables ensure high-speed communication between the partitions if a large design has to be mapped onto multiple FPGAs. Signals between these partitions are connected using pin multiplexing on the cables, as described next.

15.2.1 Inter-FPGA Pin Multiplexing

To house complex invasive computing architectures, which exceed the number of resources available in one of the high-end Virtex-7 2000T FPGAs, the design has to be partitioned into up to four parts (corresponding to the number of FPGAs). In case the architecture is partitioned to more than one FPGA, it should nevertheless act like a single-chip design to enable benchmarking and validation of test results.

We decided to connect each FPGA with its direct, non-diagonal neighbours using an inter-FPGA ribbon cable with 118 pins each. The proFPGA platform provides a mechanism called pin multiplexing to serialise signals on one FPGA
and transmit them to another FPGA using single-ended or differential signalling. By driving these pins in high-speed SerDes mode, the actual amount of transferred single-bit signals is the pin count multiplied by the *muxing factor*. The muxing factor can be chosen between 4 and 64, but care must be taken that the frequency is not set too high and that the transmission on single-ended pins is still stable. Therefore the number of logical connections between the design partitions should still be kept low—in our configuration, using a muxing factor of 4 running at 200 MHz allows transferring 236 signals per direction.

### 15.2.2 Memory and Interfacing

The prototype platform supports memory resources of different types. Each Xilinx XC7V2000T FPGA provides about 5.5 MiB of distributed block RAMs (BRAMs), which are useful for implementing various buffers and caches. Additionally, each FPGA carries two low-latency SRAM extension boards to mimic larger on-chip memories of future manycore architectures. Each board features three SRAM chips, adding up to a total SRAM capacity of 24 MiB per FPGA. As a third memory resource and the last level in the memory hierarchy, up to two DDR3 extension boards are utilised with a capacity of 4 GiB of memory each. To provide connectivity to other systems, an Ethernet and an
15.3 Invasive Architecture Prototype

Figure 173: proFPGA system with four FPGAs and several extension boards. The figure shows high-speed interconnect links between the FPGAs (blue) and various extension boards. HDMI extension board can be added. The respective controllers and interfaces have been developed and tested thoroughly for the different extension boards. Figure 172 shows a typical arrangement of the extension boards. The whole prototype is displayed in Figure 173.

15.2.3 Configuration and Debugging

Each proFPGA system is accompanied by a host PC used for configuration and debugging purposes. The proFPGA mainboard provides a common interface named MMI64 [12] to communicate with the FPGAs, especially to configure them. It can be accessed either using a dedicated Ethernet, USB, or PCI Express connection. The latter is preferred to achieve the highest performance possible.

Vendor tools can be used to program the FPGAs with their respective bit-streams. After programming, a host system may use an API to communicate to MMI64 components within the user design by using an API. The components accessible via MMI64 are connected in a tree topology and are identified by individual IDs. Arbitrary read and write operations can be performed on components to transfer data.

15.3 Invasive Architecture Prototype

Using the ProDesign proFPGA prototyping system, we support single-tile and 2×2 tile designs—mapped onto a single FPGA—as well as 4×4 tile designs
that span over all four available FPGAs. A tiled design itself is modular in the number and combination of different tile types, which are connected by the iNoC (see Chapter 9).

The existing tile types comprise normal processor tiles consisting of a configurable number of LEON3 processing cores, or more specialised tiles, i.e. i-Core (see Chapter 5), TCPA tiles (see Chapter 6), memory and I/O tiles. A fundamental component of all tile types is the memory-mapped AMBA v.2 AHB/APB bus, that connects the processors, the local memory, the custom modules and the network adapter that establishes the connection to the iNoC.
15.3 Invasive Architecture Prototype

15.3.1 Invasive Network-on-Chip (iNoC)

The iNoC (see Chapter 9) forms the global interconnect of a tiled invasive hardware architecture. It has been designed with scalability in mind and allows to configure differently sized, Manhattan-style architectures. Each iNoC router features a tile interface which is independent of the tile type, thus tiles can be replicated and attached to the iNoC at any location according to the current architecture configuration. The resulting tiled architecture uses the partitioned global address space (PGAS) model, where each tile’s local memory is mapped to the global address space. Section 15.3.7 provides details on the addressing.

If a configuration does not fit on a single FPGA, it is subdivided at the iNoC boundaries to keep the amount of connections between the individual partitions low. In case of a 4×4 configuration, iNoC connections between the second and the third row/column of tiles use the inter-FPGA pin multiplexing to communicate over FPGA borders, see Figure 174. Inputs and outputs on the respective FPGAs are registered to compensate for the much longer logic path, leading to a higher latency (two additional clock cycles) on NoC links between FPGAs compared to internal links. But this is necessary to compensate the delay of (de-)serialisation and does not have a significant impact on the overall performance.

15.3.2 Processor Tiles

Processor tiles are the main computing resource of the demonstrator system. One developed tile is a LEON3-based multicore system with up to 5 processor cores. An abstract view of a processor tile can be seen in Figure 175a. The LEON3 is a customisable SPARC V8 compliant processor provided by Gaisler’s GRLib hardware library targeted for FPGAs which can operate at a frequency of 100 MHz in the basic configuration with a level-1 cache, but without floating-point support.

The GRLib library also provides modules to implement the memory mapped AMBA2 AHB/APB bus—the necessary bus arbiter and interrupt controllers as well as common master and slave modules. Via this bus, the processors can access a local memory, the tile-local memory (TLM), consisting of 8 MiB SRAM memory that is used to store the tile-local operating system and applications, and which can also be used as a local shared memory. Further, we connect each tile to a CiC module, which is a FIFO-based i-let job scheduler (see Chapter 7), and to a network adapter (NA), which connects to the iNoC. Accesses to the memory of other tiles are cached in the shared level-2 cache. Figure 175a
Figure 175: An abstract architectural view of (a) a compute tile, (b) a TCPA tile, and (c) an i-Core tile. All tiles are equipped with a transactor (TA) and a debug support unit (DSU) to enable external access from the host PC, especially debugging.

shows the internal structure of a processor tile. All other tile types including memory and I/O tiles are variations of the processor tile and use the same bus and NoC interfaces.

**Programming a tile**  In addition to the compute modules, additional helper modules are connected to the bus of each tile, namely the transactor (TA), which establishes the connection to the host PC, and the Debug Support Unit (DSU). They are not supposed to be used during an application’s execution, but to interface to Gaisler’s GRMon monitoring tool. GRMon runs on a connected host PC and uses the MMI64 interface to connect to the transactor on the tile’s bus. Using this setup, a specifically developed management tool called *iman* is used to program each tile and run or debug applications. *iman* conveniently encapsulates loading a bitstream into each FPGA of the prototyping system, connecting a *GRMon* instance in its own *tmux* session to each tile via the transactor, using *GRMon* to copy the application binaries into each tile’s local memory, and starting the program execution. Using the transactor infrastructure, all tiles can then be monitored in parallel.
Local Memory Access  The tile-local memory (TLM) of each tile is accessed by a memory controller which has been designed and implemented. Each memory controller allows burst read and write accesses from and to one of three SRAM banks on a connected SRAM extension boards. The SRAM accesses are pipelined and clocked at 200 MHz. The low-level memory controller needs 4 cycles for initialising read accesses and at each edge of the clock, 18 bits can be read or written from/to memory. Exploiting the fast memory clock, the memory controller can load data faster than the running prototyped architecture and thus provide uninterrupted burst accesses up to the 1 kiB bursts allowed by the AHB/APB protocol. Using a small FIFO, the SRAM controller can pause, resume and preload data without any observable delay and thus continue burst copies larger than 1 kiB without having to reinitialise the data transfer.

Dynamic Voltage Frequency Scaling  In order to adapt to various performance and power requirements during run time, we integrated the emulation of dynamic voltage frequency scaling (DVFS) on core granularity into each tile. Based on a thermal modelling block, 20 predefined power modes, characterised by voltage/frequency pairs can be emulated. The thermal modelling block takes care for translating the selected power mode of a core into the corresponding power and temperature values (see Chapter 8). A custom-designed hardware-based frequency controller emulates the associated frequency of the core via pipeline manipulation, as illustrated in Figure 176. Each of the possible frequency values corresponds to different run and stall periods of the CPU pipeline. The power mode can be controlled either directly by the operating system or by a hardware thermal manager that ensures the compliance with dynamic latency and power constraints. Switching between different power modes using DVFS is one central mechanism for enforcing non-functional program requirements at runtime. Consequently, DVFS is the key component for showcasing different aspects within the enforcement demonstrator (see Section 15.5.1), such as latency corridor enforcement or thermal safety enforcement, see, e.g. Chapter 2 and Chapter 3.
15.3.3  **i-Core Tiles**

On an i-Core tile, as seen in Figure 175c, one of the LEON3 processing cores is substituted by an i-Core (see Chapter 5). An i-Core provides Special Instructions (SIs) that are loaded into reconfigurable containers to accelerate parts of an application. This can be simple instructions like a square-root operation up to whole loop bodies, as long as the logic fits into the predefined container. When an application (i-let) is scheduled to an i-Core tile, the application can check if it is actually running on the i-Core and whether the needed SIs are available. If they are, the application may benefit from the provided acceleration, and if they are not, the application can default to a normal LEON3-based execution.

15.3.4  **TCPA Tiles**

A TCPA (invasive Tightly-Coupled Processor Array, see Chapter 6) tile contains a TCPA and uses a single LEON3 processor core as a companion to perform configuration tasks, as seen in Figure 175b. A TCPA is a class of coarse-grained reconfigurable arrays that uses an array of tightly-coupled processing elements (PEs) that can be dynamically programmed and configured to accelerate unbound loop nests. The buffer memory surrounding the PE array provides an interface to the AHB/APB bus. Only the PEs at the border of the array have access to data residing in a buffer memory, which is streamed through the array while processing it in each passing PE. The output data is streamed to an adjacent buffer and written back to local or external TLM.

The flow of using a TCPA on a prototyping system is as follows: (1) Load or locate the configuration of the TCPA. Such a configuration contains the programs of the processing entities, the configuration of the interconnect structure, the configurations of the address generators and the global controller. (2) Define the TCPA’s input data. (3) Allocate memory in the TLM or globally shared memory for the TCPA’s output data. (4) Patch the addresses of inputs and outputs into the TCPA configuration. (5) Transfer the configuration’s address and size to the TCPA controller interface. And finally (6), the TCPA loads and applies the configuration, loads the input data from the patched addresses, and writes the output after successfully executing the configured algorithm to the patched output addresses. When the TCPA finishes execution of a loop nest program, it raises an interrupt.
**15.3.5 Shared Memory Tile**

The Shared Memory Tile (SMT) provides access to the connected 4 GiB DDR3 memory extension boards. There can be up to two Shared Memory Tiles on a 4×4 tiled architecture. Limited by the address space of the AHB/APB bus, 2 GiB of DDR3 memory distributed over the available SMTs can be used.

**15.3.6 I/O Tiles**

The I/O tile is a special tile that serves the prototyping architecture with interfaces. Our demonstrator system mainly provides two types of interfaces to let running applications input or output data, either by communicating with other systems such as host a PC or video source (e.g. a camera).

**Ethernet**  The I/O tile connects to a 1000 Mbps Ethernet port supporting the standard TCP/IP protocols, which are implemented in the OS. The Ethernet interface is special in the way that it can be accessed from any other tile by operating system routines. The OS reserves one of the LEON3 cores on the I/O tile to process Ethernet requests.

**Video interface**  The video interface connects an HDMI input and output port to a double buffering frame buffer. It can stream in- and output video frames at the same time, while also generating box overlays to the output stream on the fly. The double buffering is used to avoid stalls and backpressure when reading frames into the system takes longer than the frame dictates.

**15.3.7 Address Mapping**

We defined a specific address map for the multi-tile invasive architecture, supporting configurations with up to 16 tiles. This covers local resources including TLM and other tile-local components as well as remote TLMs and shared memory. The bus provides 32 bits for addressing and thus an address space of 4 GiB for the complete design, which we divided as shown in Table 25. Our default address mapping defines 2 GiB of shared memory, followed by tile-local addresses used for the tile-local memory and periphery, and then by a global address map. The global map defines an address space of 16 MiB to access a portion of the tile-local address space of each included tile. This address space contains the remote tile’s local memory as well as some peripherals that need to be accessible remotely, e.g. the cache coherency manager (see Section 9.5). With all 16 tiles enabled, the size of the global map sums up to 256 MiB.
Table 25: Default address map.

<table>
<thead>
<tr>
<th>Start Addr.</th>
<th>Size</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>2048 MiB</td>
<td>Shared memory (DDR)</td>
</tr>
<tr>
<td>0x80000000</td>
<td>8 MiB</td>
<td>Tile-local memory (TLM)</td>
</tr>
<tr>
<td>0x80800000</td>
<td>8 MiB</td>
<td>Local peripheral (remotely accessible)</td>
</tr>
<tr>
<td>0x81000000</td>
<td>8 MiB</td>
<td>Local peripherals (without remote access)</td>
</tr>
<tr>
<td>0x84000000</td>
<td>1 MiB</td>
<td>TCPA controller</td>
</tr>
<tr>
<td>0xA0000000</td>
<td>8 MiB</td>
<td>Core i-let controller (CIC)</td>
</tr>
<tr>
<td>0xB0000000</td>
<td>16 MiB</td>
<td>Remote mapping of Tile 0</td>
</tr>
<tr>
<td>0xB1000000</td>
<td>16 MiB</td>
<td>Remote mapping of Tile 1</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>0xBF000000</td>
<td>16 MiB</td>
<td>Remote mapping of Tile 15</td>
</tr>
<tr>
<td>0xD0000000</td>
<td>256 MiB</td>
<td>Local i-Core peripherals</td>
</tr>
<tr>
<td>0xF0000000</td>
<td>128 MiB</td>
<td>Debug support unit (DSU)</td>
</tr>
<tr>
<td>0xFFFF0000</td>
<td>64 kiB</td>
<td>AMBA plug &amp; play</td>
</tr>
</tbody>
</table>

15.3.8 Non-Intrusive Monitoring System

In order to analyse the system for collecting experimental results and for debugging purposes, it is essential to collect monitoring and result data from the system. This can be achieved using either hardware or software counters, which are present in most of the different modules in the prototype. Most of this data is accessible using the bus system within the tiles and can be read from benchmarking software. However, obtaining these values through the main system bus is not possible without a certain impact on the system. Depending on the measurement, this is undesired as it may alter the timing behaviour and possibly the result of the measurement itself. Therefore, a non-intrusive monitoring system has been developed as shown in Figure 177.

This monitoring system is directly connected to the counters of many different components in the system. Its concept consists in sampling these counters periodically, collecting their values and forwarding them to an external computer for further processing. This whole process happens without influencing the execution of the prototype, so that measurements are as precise as possible. Figure 177 shows the monitoring system architecture, including the tree structure to collect data from every tile. For transmission to the host PC,
15.4 Integration Procedure and Debugging

Testing and debugging complex prototyping systems is a challenging task. As numerous researchers from the four research areas contributed to the prototype, we set up a sophisticated procedure for integration and verification. In order to ensure the correct functionality, hardware and software designers were aided by a specific debugging environment. The highest level of system-level tests was made up of a test suite of real applications (in our case the NAS parallel benchmarks). This enabled to perform holistic tests of the components and their interplay in an automated fashion while running on top of iRTSS on the prototypes. Output traces of error messages helped to categorise them and give first hints where to focus further testing. This test suite verified not only the correct behaviour of all individual components, but also proved the correct interplay between all modules including the OS.
Validation and Demonstrator

Beside that, we also provided a so-called bare metal test environment without operating system to execute pure C code directly on the processors. These tests could easily be modified to investigate the functionality and performance of specific modules with respect to all corner cases. Directly accessible memory-mapped registers allowed a better control flow and easier debugging routines of the hardware components without the need of designing system-level test benches in VHDL or Verilog. These lightweight bare metal tests furthermore offered the possibility to run a low-level hardware simulation (QuestaSim) with very little time overhead. A compile script directly compiled the same code twice, one binary for the hardware prototypes and the other for the QuestaSim simulation environment. Several hardware components offered internal status information available through memory-mapped hardware registers. We furthermore used Chipscope (the on-FPGA logic analyser solution from Xilinx) for on-chip debugging.

Continuous Integration The hardware design was kept under version control using Git. To improve the stability of the source code of the hardware prototype, a comprehensive set of tools and scripts had been developed to automatically verify the hardware design. For each change, on the stable or any development branch, the hardware design underwent a compilation and synthesis check. The compilation check consisted of building the full design using a simulator as well as doing an elaboration run to verify the structural compatibility between components. The synthesis check ran the FPGA vendor tools, which was Xilinx Vivado in our case, to generate a bitstream. Developers were signalled automatically if any of their changes broke the simulation or synthesis checks. In addition, they could use the automatically generated bitstream to further test and validate their design changes. Setting up these continuous integration tasks helped to identify issues earlier and to improve the quality of our hardware design.

15.5 Invasive Computing Demonstrators

In the following, we present joint efforts centred around the FPGA-based demonstration platform to show and validate the findings of the entire collaborative research centre in its last funding phase. The first common demonstrator focuses on the most recent leitmotif of invasive computing run-time

---

1 Earlier demonstrators can be found on YouTube such as “Invasive Computing for Dummies” (https://youtu.be/4kOQYHhnZWo), “Invasive Computing for Experts” (https://youtu.be/p-TpiCES9cc), and “Robotic Hand Demo on TCPA” (https://youtu.be/Znfe-mbvUk, [5]), but also on our website https://www.invasic.de. In addition, several individual demos both
requirement enforcement. Presented are also several hardware accelerators, e.g. near-memory graph copy acceleration using i-Cores or the acceleration of CNNs on a TCPA as part of a prosthetic hand demonstrator.

15.5.1 Enforcement Demonstrator

Embedded systems and their respective applications are created for specific purposes and optimised to meet different non-functional program properties, such as performance, timing, cost, security, and power. Often these non-functional properties have to fulfil strict requirements, e.g. hard real-time constraints. The concurrent execution of several tasks on an MPSoC, respecting or optimising multiple non-functional program properties, faces a complex mapping problem. Even when isolating applications from each other either in space—as investigated in invasive computing—or time, the compliance with such program properties (e.g. by adapting the number of resources allocated to a task) is challenging; especially if the program execution is affected by other applications or inputs only known at runtime. Such variations may violate the requirements if one does not want to reserve system resources provisionally for the worst case. As a remedy, run-time requirement enforcement (RRE) was proposed in Chapter 2 and is investigated across all project areas within the CRC/Transregio. The main idea of RRE is to provide means to control the non-functional execution properties within acceptable bounds at runtime.

The goal of the enforcement demonstrator introduced in the following is to showcase the enforcement of different non-functional properties, including, throughput, energy, power, temperature, safety, and security by using both design-time application characterisation and RRE. Here, various RRE aspects and techniques for adapting control knobs at runtime are demonstrated. These include switching of precomputed operating points, adaptations in the application’s algorithm, task migration, voltage/frequency scaling, the usage of accelerators, such as i-Cores and TCPAs as well as near-memory computing. The whole stack, from applications to hardware (i.e. the prototyped tiled MPSoC architecture on the proFPGA platform), including the locations where RRE techniques are incorporated are sketched in Figure 178. Next, the different RRE aspects of the demonstrator are briefly described.

Throughput Corridor Enforcement  The first aspect demonstrates the capabilities of RRE of real-time and energy properties. For illustration, a video processing application is considered, where video frames from a camera (image on FPGA platforms and commodity hardware were developed. For these, we refer to the other chapters, e.g. x86-based use cases in Chapter 14.
source (IS in Figure 179) are captured continuously and streamed into the MPSoC architecture on the proFPGA platform. The task of the application is to detect objects in the video stream. The streaming pipeline was implemented in ActorX10 [10]. Figure 179 illustrates the pipeline of subtasks (actors) required for the object detection. Some actors could be analysed and mapped at design time, e.g. the compute-intensive preprocessing steps greyscale conversion, edge and corner detection are input-invariant and are mapped onto a TCPA tile. In contrast, the number of features within a captured frame as the result of the corner detection (CD) is only known at runtime, thus affecting the execution time of the following actors. Here, an enforcer $F$ is used to keep the execution time within a defined latency corridor and simultaneously minimise energy. The enforcer $F$ takes as input the number of features of the current video frame and determines the required number $n$ of cores (up to four parallel SIFT description (SD) instances, as shown in the figure) and one power mode $m$ out of 20 available modes (cf. frequency scaling in Section 15.3.2). For details on the FSM-based enforcer, we refer to Chapter 2. The demonstrator illustrates how enforcement can ensure a stable throughput, whereas without RRE, the object detection quality fluctuates significantly. Notably, frequency jitter is perceived as a very annoying and even headache-causing effect in many video applications, including virtual and augmented reality.

**Thermal Safety Enforcement** In addition to performance goals, the second demonstration aspect shows the enforcement of thermal constraints in a scenario with several applications. Object detection also makes up an application with soft real-time requirements, as described before. Multiple instances of this application, each with a different constraint, run concurrently on the invasive architecture.
The goal of this part of the demonstrator is to enforce thermal constraints in a composable manner. To achieve this, each operating point (characterised by the number and voltage/frequency levels of cores to be allocated) is additionally annotated with a *thermally save utilisation* (TSU), which indicates the number of the cores that can be active in the background in parallel to this application. When a new application is to be started in addition to already running applications, an operating point is selected whose TSU can be satisfied considering the number of active cores on the chip at the current time. Moreover, after mapping the new application, the TSU values of already running applications have to be still satisfied. The TSU values have been obtained by employing the novel power budgeting technique, TSP, presented in Chapter 7. TSP considers the floor plan and the characteristics of the cooling system of the targeted MPSoC. It selects operating points so that TSUs are satisfied, which guarantees avoiding thermal violations on the MPSoC.

**Covert Channel Detection and Security Enforcement**  Invasive computing has previously shown how security properties of applications can also be enforced, e.g. by ensuring the confidentiality of applications. This was achieved by encrypting critical parts of an application which were then decrypted on the fly by the so-called Atlas core [7].

Apart from such passive security measures, this part of the demonstration shows how to actively detect a malicious covert channel attack in this aspect of the demonstrator. In this scenario, an application is running on one or several tiles, and an attacker runs a malicious application on one of the unoccupied tiles to snoop data over temperature as the covert channel. Thus, the attack cannot be detected by conventional operating system measures.

To detect suspicious behaviour, heat data of each tile is monitored and processed by a pretrained convolutional neural network (CNN). If the CNN
detects temperature patterns between neighbouring tiles that indicate an attack, the security of our system is enforced by taking one of several countermeasures. The first method would be to abort the application and report the attack. This ensures maximal security until the attacker is identified. Other measures include migrating the application to a different tile or changing the frequency of the utilised cores. Both would change the temperature patterns of the running application and hinder the attacker from getting any useful information.

**Hardware Acceleration** With the trend towards distributed tile-based architectures novel challenges arise to utilise the nominal system performance at its best. In order to bridge the widening gap between performance increase of processors and memory, near memory computing can be a promising approach in distributed systems. For further performance improvements applying hardware accelerators for specific computational intensive tasks is a common approach. The goal of this part of the demonstrator is to showcase the potential of near-memory integration of graph-copy tasks and hardware acceleration by application-specific solvers in a real-world case study for tile-based manycore architectures.

Chosen is an application that calculates shallow water equations (SWE) on our platform [8] (for further details, see also Chapter 14). This application simulates the wave propagation in a tsunami scenario and spreads the whole computation over 15 places while each place corresponds to one compute tile of our prototype platform in this scenario. A computational grid provides a visualisation that divides the whole map into 15 Cartesian patches, each of them corresponding to one computational place. Figure 180 illustrates the visual output on the host PC.

A near-memory graph copy accelerator, as described in detail in Chapter 9 is able to visibly accelerate the inter-tile communication significantly by transferring arbitrary object graphs in an efficient manner [9]. As an addition to this, we equip all tiles with an extended version of the LEON3 core, the so-called i-Core, as described in Chapter 5. The i-Core enables the use of application-specific special instructions, which are executed on a reconfigurable fabric. The processing time of computationally expensive tasks is greatly reduced compared to standard LEON3 cores, which leads to an acceleration of the overall computation.
15.5.2 Prosthetic Hand Demonstrator

Over 12,000 people in the European Union experience work accidents, resulting in traumatic amputation each year [2]. The loss of a limb is a tragedy and comes with massive changes for the affected person. Prostheses can make an important contribution to enabling these people to regain autonomy in their daily life.

As such, our prosthetic hand demonstrator showcases how invasive computing can be used to control a prosthetic hand developed at the chair of High Performance Humanoid Technologies at KIT (also see Chapter 13). The hand is controlled by two DC motors that pull or loosen tendons connected to the fingers to form a grasping motion and is planned to be powered by a portable energy source. First, the object that is supposed to be grasped is recorded by an embedded camera and categorised by a classification convolutional neural network (CNN) running on an integrated FPGA. A segmentation CNN then tries to determine the specific object inside the identified object class. If an object is recognised, the hand’s grasping motion and strength are fine-tuned to it.

Our demo shows the benefits of invasive computing to accelerate the CNN inference time until the user can grasp the object. By exploiting the concept of run-time requirement enforcement (RRE) while running part of the CNN inference on a TCPA (see Chapter 6), the power consumption of the device decreases while the battery runtime increases. As seen in Figure 181, a TCPA
Figure 181: Prosthetic hand demonstration flow. See Chapter 13 for further details on the prosthetic hand.

runs on our proFPGA demonstrator platform, which is connected to the camera source located in the hand’s palm via USB. When the user triggers the compute pipeline, the image is classified on the device itself. However, as very accurate segmentation CNNs also come with high memory and computational power demands [6], the image and the classification are transmitted to the proFPGA system to accelerate the over 11 million MAC operations of our used segmentation CNN on a TCPA.

15.6 Conclusion and Acknowledgement

This chapter presented the effort and the results to validate and demonstrate invasive computing on a multi-FPGA platform. We realised modular invasive compute architectures, which are tile-based and configurable in terms of size and composition of tiles. To achieve this, we integrated several hardware components into that prototype system in order to fulfil all hardware requirements of invasive computing. This work involved the cooperation among many people working in a multi-disciplinary environment to develop specific contributions and integrate them into a fully functional system. Three generations of doctoral researchers worked together in a distributed environment, which stretched across FAU, KIT, and TUM, and brought this endeavour to final success.

We would like to acknowledge the valuable contributions of Srinivas Boppu, Christian Heidorn, Leonard Masing, David May, Shravan Muddasani, Sven Rheindt, Éricles R. Sousa, Akshay Srivatsa, and Aurang Zaib.
References


Invasive computing is a paradigm for designing and programming future parallel computing systems. For systems with 1,000 or more cores on a chip, resource-aware programming is of utmost importance to obtain high utilisation as well as computational, energy and power efficiency. Invasive computing provides a programmer explicit handles to specify and argue about resource requirements desired or required in different phases of execution; In an invade phase, an application asks the operating system to allocate a set of processor, memory and communication resources to be claimed. In a subsequent infect phase, the parallel workload is spread and executed on the obtained claim of resources. Finally, if the degree of parallelism should be lower again, a retreat operation frees the claim again, and the application resumes a sequential execution. To support this idea of self-adaptive and resource-aware programming, not only new programming concepts, languages, compilers, and operating systems were needed to be developed, but also revolutionary architectural changes in the design of MPSoCs (multiprocessor systems-on-a-chip) to efficiently support invasion, infection, and retreat operations. This book gives a comprehensive overview of all aspects of invasive computing.