Operating-System Support for Efficient Fine-Grained Concurrency in Applications

Betriebssystemunterstützung für effiziente feingranulare Nebenläufigkeit in Anwendungen

Der Technischen Fakultät der Friedrich-Alexander-Universität Erlangen-Nürnberg zur Erlangung des Doktorgrades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

vorgelegt von

Christoph Erhardt
Als Dissertation genehmigt von der
Technischen Fakultät der
Friedrich-Alexander-Universität Erlangen-Nürnberg

Tag der mündlichen Prüfung: 03. März 2020
Vorsitzender des Promotionsorgans: Prof. Dr.-Ing. habil. Andreas Paul Fröba
Gutachter:
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Prof. Dr.-Ing. Jörg Nolte
Abstract

The steadily advancing trend towards multi- and manycore computing architectures bears enormous challenges for developers of application software. To be able to make efficient use of the raw parallelism provided by the hardware, programs must explicitly cater for that fact. The classic programming model of a multithreaded application process, which consists of a number of control flows (threads) managed and scheduled by the operating-system kernel within a shared address space, is being increasingly stretched to its limits: on the one hand, creating threads and switching between them is not sufficiently lightweight; on the other hand, structuring a parallel application around threads is often cumbersome and puts needless obstacles in the programmer’s way.

A suitable alternative to multithreaded programming is the use of a so-called concurrency platform that supports developers in articulating applications as a conglomeration of fine-grained concurrent activities. Concurrency platforms come with a runtime system that is responsible for dispatching the lightweight work packages to the available computing resources. Such runtime systems generally build upon the abstractions provided by an underlying commodity operating system such as Linux – that is, upon threads as abstractions of processor cores. This construction results in a number of disadvantages: for instance, the operating system’s scheduler acts without consulting the runtime system, thus making decisions that are potentially unfavourable from the application’s point of view; the coexistence of multiple parallel application processes causes problematic reciprocal interference; blocking system calls cause a temporary loss of parallelism.

This thesis presents AtroPOS, the design of an atrophied parallel operating system that is specially geared towards supporting concurrency platforms on manycore systems. AtroPOS is a derivative of the OctoPOS operating system and has undergone comprehensive further development; it rests on the paradigm of invasive computing and adopts its fundamental concepts: resource-aware programming, exclusive allocation of processor cores to applications, tailoring and dynamic reconfigurability. The operating-system kernel provides a boiled-down set of essential low-level abstractions on top of which arbitrary runtime libraries can be placed. InvRT, the invasive runtime system that supports executing applications of invasive computing, was developed as a reference runtime library.

By default, AtroPOS makes the existing physical processor cores directly available to the application; their virtualisation is strictly optional and there is no notion of threads. The scheduling of user control flows is carried out purely on the user level by the runtime system without involving the operating-system kernel; this allows for the efficient handling even of very fine-grained concurrency within the application. System calls that may block within the kernel have asynchronous invocation semantics and return immediately upon blocking so that loss of parallelism during the waiting time is ruled out by design. Notification of completed system operations is carried out by means of a generic mechanism that passes user-defined data structures upward to the application and can be used by the runtime system to construct arbitrary synchronisation data structures such as futures. The same versatile mechanism is harnessed on tiled computing systems to allow parts of a distributed application to communicate with one another.

In addition, AtroPOS offers configurable vertical isolation: the strict separation of the operating-system kernel from the application can be enabled and disabled in a coarse- and fine-grained
manner, and both statically and dynamically. With this, type-safe applications can issue system calls as ordinary function calls and thus lower their direct and indirect costs.

The aforementioned concepts were implemented in the AtroPOS kernel and the InvRT runtime system in the context of this thesis; they were evaluated with the aid of micro-benchmarks and various application suites. Moreover, the runtime library of the parallel programming language Cilk Plus – an extension of C/C++ – was ported to the AtroPOS interface in order to showcase the versatility of the approach.
Zusammenfassung


First of all, I wish to thank my adviser, Wolfgang Schröder-Preikschat, for his guidance and for giving me the opportunity and freedom to pursue my interests during my years as a member of his research group. He provided a superbly pleasant work environment where it was possible without pressure to concentrate on research, on engineering and also on teaching.

I would also like to express my gratitude to Jörg Nolte for taking the time to review my thesis.

I thank all the members of the systems-software group in Erlangen I have had the privilege to work with during our time together. I have thoroughly enjoyed the past $n$ years with you all! Special thanks go to Jürgen Kleinöder and Daniel Lohmann, who pulled the strings to grant me the funding and time I needed to complete this dissertation. To my long-time office mate Tobias Klaus I say thank you for over five years of lively conversations about day-to-day politics, open messaging protocols and KDE bugs.

My octopoid colleagues deserve praise not only for their vast technical competence but also for being a bunch of excellent chaps. Benjamin Oechslein wrote large parts of the original implementation of OctoPOS and patiently answered all my questions on its internals. Jens Schedel was the master of infrastructure and organisation and inspired me through his enormous passion for teaching. Gabor Drescher laid crucial groundwork by getting the x86_64 port into shape, and by implementing proper memory management and the multi-application model. Sebastian Maier and Florian Schmaus have grown into leading roles and have taken up the mantle; both have since shaped the further development of OctoPOS and have contributed countless excellent ideas to the project.

Thanks go to all my other invasive colleagues from Erlangen, Karlsruhe and Munich who have made me feel at home in the project – not least through countless games of Werewolves. If I had to pick one group in particular, it would be the compiler crew from Karlsruhe: Matthias Braun, Sebastian Buchwald, Andreas Fried, Manuel Mohr and Andreas Zwinkau. They were extremely committed to the success of the overall project, and we were always on the same page and spoke the same language.

The students from whose outstanding work I benefited a lot should not go unmentioned: Clemens Lang and Simon Kuhnle enhanced and improved the KESO compiler; Florian Schmaus, whom I have already mentioned, heroically built a Cilk Plus compiler and a work-stealing scheduler for OctoPOS; Tobias Langer and Martin Gumbrecht developed and implemented dynamic memory protection; Jonas Rabenstein made countless contributions to the OctoPOS code base; Lukas Senger built the Obinex tool for batch-executing operating-system binaries and collecting their output; Felix Sauer wrote the prototype implementation of many of the ideas on which this thesis is based.

I appreciate the work done by my proofreaders – Sebastian Maier, Manuel Mohr, Benjamin Oechslein and Jens Schedel – who provided valuable comments and suggestions that have helped refine the quality of this treatise.

I thank my family, especially my parents for granting me the freedom to go my own way and for selflessly supporting me in all my endeavours. Although my father is no longer with us, I know he would be very proud. It is to him that I dedicate this work.

Finally, I thank 金玮 for her endless and unconditional love and support, for believing in me even in times when I would not, and for making me a better person. ♥
## Contents

1 Introduction ........................................... 1
  1.1 Problem Statement .................................. 5
  1.2 Contributions of this Dissertation .................. 7
  1.3 List of Publications ................................ 8
  1.4 Thesis Structure ................................... 10

2 Fundamentals and Background .......................... 13
  2.1 Overview of Invasive Computing .................... 13
  2.2 Invasive Hardware Platform ........................ 14
  2.3 The OctoPOS Operating System ..................... 16
  2.4 Interim Conclusions ................................ 31

3 State of the Art ....................................... 33
  3.1 Threading Models .................................... 33
  3.2 Fine-Grained Concurrency ............................ 34
  3.3 Concurrency Platforms ................................ 39
  3.4 Avoiding the Thread-Blocking Anomaly .............. 48
  3.5 Kernel Execution Models ............................. 53
  3.6 Survey of Related Operating Systems ............... 54
  3.7 Summary ............................................. 59

4 Revisiting the Invasive Execution Model and Its Implementation .................. 61
  4.1 Context-Management Improvements ................... 61
  4.2 Wait-Free Synchronisation ............................ 67
  4.3 Handoff Scheduling .................................. 74
  4.4 Summary ............................................. 75

5 The Design of AtroPOS and InvRT ....................... 77
  5.1 Design Principles .................................... 78
  5.2 Overall Architecture ................................ 78
  5.3 Configurable Privilege Isolation .................... 80
  5.4 Control-Flow Model .................................. 86
  5.5 Inter-Tile Communication ............................. 94
  5.6 Invasive Runtime System .............................. 96
  5.7 Applicability to Other Runtime Systems ............ 102
  5.8 Summary ............................................. 103

6 Implementation Details ................................. 105
  6.1 Kernel Interface ...................................... 105
  6.2 Implementation of the Kernel Execution Model .... 108
  6.3 System-Call Mechanism ............................... 113
  6.4 Event-Notification Interface ........................ 120
  6.5 Implementation of Inter-Tile Communication ....... 126
  6.6 Dealing with Buffer-Space Exhaustion ............... 128
  6.7 Security Considerations ............................... 131
  6.8 Implementation of the Invasive Runtime System .... 132
  6.9 Summary ............................................. 141
## Contents

### 7 Evaluation and Discussion
- 7.1 Evaluation Environment and Methods .................................................. 143
- 7.2 Memory Footprint .................................................................................... 144
- 7.3 Micro-Benchmarks .................................................................................. 146
- 7.4 Application Benchmarks ......................................................................... 163
- 7.5 Discussion and Summary ........................................................................ 174

### 8 Conclusion
- 8.1 Summary ................................................................................................ 177
- 8.2 Outlook .................................................................................................. 179
- 8.3 Closing Remarks ..................................................................................... 180

### Appendices
- A List of System Calls .................................................................................. 183
- B Security Recommendations for Invasive Hardware Components ............. 185
- C Evaluation Results .................................................................................... 187

### Acronyms and Abbreviations

### Bibliography
You can’t just place a LISP book on top of an x86 chip and hope that the hardware learns about lambda calculus by osmosis.

James Mickens, ‘The Night Watch’ [Mic13]

Introduction

The free lunch is over [Sut05] for good.

In the decades following the invention of the semiconductor integrated circuit in 1958, the design of microprocessors was largely dictated by ever-proceeding advances in manufacturing technology. Performance improvements were mainly achieved by increasing clock frequencies, which let processors execute more and more instructions per second. The prediction known commonly as Moore’s law [Moo65] – stating that the continual downscaling of transistor sizes causes the density of components on a silicon chip to double every one to two years\(^1\) – has proved true until today and is expected to hold for at least a few more years. At the same time, Dennard scaling [Den+74], which asserts that power density stays roughly the same while transistors shrink, allowed microprocessors to be operated at decreasing voltages and currents.

As it became clear that the exponential development was continually widening the existing performance gap between logic and memory, processor designers added caches to keep data closer to the computation. Since then, caches have constantly become larger and are now organised hierarchically with complex prefetching and eviction strategies. Simultaneously, the internal architecture of microprocessors has undergone successive refinements that aim at increasing the number of instructions executed per clock cycle. Instruction pipelining [RL77], superscalar execution, register renaming and out-of-order execution [SS95b] exploit instruction-level parallelism while speculative execution and branch prediction keep the processor’s execution units from idling needlessly. All these techniques have made chip designs more and more complex\(^2\), but they guaranteed regular performance enhancements for sequential programs with each new processor generation.

Which brings us back to the free lunch for application programmers, and to how it ended.

Until about 2005, software developers could rely on the fact that the ongoing processor development was automatically going to benefit their applications. Even if a program brought a processor to its limit at the time, it was clear that it would run well on subsequent hardware generations. Essentially, programmers could sit back and enjoy a free meal while processor architects were taking care of

\(^1\)Moore originally spoke of a doubling every year for the next ten years; in 1975 he revised his prediction to a doubling every two years for the following decade.

\(^2\)As the recent Meltdown [Lip+18] and Spectre [Koc+18] vulnerabilities have starkly demonstrated, excessive hardware complexity is a problem in itself, albeit not one that will be covered extensively in this thesis.
1 Introduction

Figure 1.1: Trends in microprocessor characteristics since the 1970s.

The kinks in clock frequency and single-thread performance around 2005 are clearly discernible, as is the simultaneous advent of the multi- and manycore era. Figure taken from [Rup18] under the Creative Commons Attribution 4.0 International Public License.

the rest. A few years after the turn of the millennium, however, Dennard scaling began to collapse. With ever-decreasing feature sizes, leakage currents became increasingly problematic; at the same time, reductions in supply voltage got more difficult to achieve. The power density of a processor chip had long surpassed that of a kitchen hotplate [DB99]. Consequently, clock frequencies – until then the biggest contributor to performance improvements – became stagnant. As Moore’s law continued to hold, processor architects had to find a new way to put the ever-increasing transistor numbers to good use. They turned to designs that integrate multiple processor cores onto a single chip with a common bus interface and partly shared caches, giving processors the capability of executing several instruction streams in parallel. This sparked two simultaneous trends: one towards multicore processors with up to about a dozen complex general-purpose cores, each of which follows a mostly traditional design, and one towards manycore chips such as the Intel Single-Chip Cloud Computer [Mat+10] or the Xeon Phi [Sod+16], with several dozen or even hundreds of simplified cores, sometimes with specialised or heterogeneous instruction sets and weakened cache-coherence guarantees. The historic development of the characteristics of microprocessors is visualised in Figure 1.1, with a noticeable turnaround around the year 2005.

The presence of multiple cores marks a fundamental turn in the way software needs to be written. Although a processor with $N$ cores can generally be used to run $N$ separate sequential programs at the same time, its real benefits can only be exploited if an application itself spreads its workload to multiple cores. In contrast to most forms of instruction-level parallelism, this kind of parallelism – called thread-level parallelism – cannot be extracted automatically by the hardware; it must be explicitly expressed in software. Now it is the programmer’s job to write concurrent applications able to exploit the parallelism available in the system they run on; after the complimentary meal comes the time for labour.

3Sutter [Sut05] jokingly remarks that the primary benefit of a second core might be that it can be used to run all the malware accumulated on the end user’s system, leaving the first core available for actual work.
Parallel programming is by no means a new development that started in the early 2000s; it goes back several decades to the early distributed-memory multiprocessor systems. However, on the one hand, it is a methodology that used to be largely confined to scientific high-performance computing and has now permeated the mainstream; on the other hand, the sheer amount of parallelism at the programmer’s disposal has hugely increased. Moreover, programmers increasingly have to deal with hybrid manycore systems that expose a mixture of both shared- and distributed-memory parallelism.

The influence that a parallel program’s structure has on its theoretically achievable speedup given a fixed-size workload is described by Amdahl’s law [Amd67]. If we assume a fixed number of processors \( N \) and know the fraction \( 0 \leq p \leq 1 \) of the application’s work time that will be spent in parallelisable code, then the overall speedup \( S \) is bounded as follows (equality would be achieved in case of perfect linear speedup):

\[
S \leq \frac{1}{p N + (1 - p)}
\]

In this inequality, \( \frac{p}{N} \) is the proportionate time spent executing parallel sections of the program on \( N \) processor cores, whereas \( 1 - p \) is the proportion of serial execution time. The resulting scalability curves are depicted in Figure 1.2. If we hypothetically suppose that \( N = \infty \), we get the following hard upper bound (marked by horizontal dashed lines in Figure 1.2):

\[
S \leq \frac{1}{1 - p}
\]

Hence, if only half of the program’s instructions could be executed in parallel \((p = 0.5)\), then the program’s run time could at best be halved – even with unlimited computing resources at its disposal! Gustafson [Gus88] gives a somewhat more reassuring assessment by focusing not on how long it takes to perform a fixed amount of work, but on how large a problem can be in order to be solved within a given amount of time. He makes the point that programmers usually dimension
the problem size in a way that it fully exploits the computing power available. Either way, two important points stand out for programming parallel systems:

1. It can pay off to parallelise even small sections of code.

2. The overhead of creating and managing parallelism should be as small as possible.

As writing code for multi- and manycore systems is far from trivial, there has been a growing trend towards so-called concurrency platforms: library- or language-based frameworks that support developers in writing concurrent software able to exploit the available parallelism. Examples of such platforms include libraries such as HPX, the Java Fork/Join framework, Intel Threading Building Blocks (TBB) or Microsoft TPL; language extensions such as Cilk Plus or OpenMP; and dedicated programming languages such as Chapel, Go or X10.

These systems share the common trait that they allow programmers to express concurrency in relatively simple and straightforward terms – in the shape of loop annotations, explicit fork/join statements or the declaration of asynchronous blocks. Their goal is not only to facilitate writing concurrent software, but also to make it easy to understand and reason about [SL05]. Using such concurrency hints, the developer can express in a descriptive manner that certain parts of the program are supposed to be concurrent, but is not required (and usually not able) to prescribe exactly how this is put into effect. The declaration of concurrency is either translated into executable code with support from the compiler, or it is converted into runtime-library calls by a thin library wrapper. The decision which computations to run how, where and when is made at run time by the concurrency platform’s runtime system. Different applications based on different concurrency platforms can be run side by side on top of the same operating system, which allows programmers to choose the tools and environment that best fit their needs.

**Concurrency and Parallelism**

Although the terms concurrency and parallelism are often conflated or used sloppily, they are not the same thing.

Concurrency of two events – or computations, for that matter – in a system means that there is no causal connection between the two [Lam78]. That is, concurrent computations may be executed in any order, potentially overlapping or even simultaneously. Concurrency does not dictate anything about the actual execution of a given set of mutually independent computations: it merely states that unordered execution is possible and legal. Programs can be decomposed into sets of concurrent computations; as such, concurrency is a structural property of a software system.

Parallelism, by contrast, states that several computations are performed at the same time on different computing resources. It stands to reason that, in order to ensure the correctness of the overall result, only concurrent computations are allowed be executed in parallel. Hence, parallelism pertains to a system’s actual execution scheme.

In summary, concurrency offers a way to structure a program in order to solve a problem that may potentially be parallelisable.

Another distinction has to be made between concurrent and parallel programming. Concurrent programming is seen as more general because it can involve arbitrary patterns of communication and interaction between computations, whereas parallel programming typically implies a more rigid and structured communication pattern.
1.1 Problem Statement

Most existing concurrency platforms are implemented fully in user space and use some kind of M:N threading: they resort to the coarse-grained concurrency abstractions provided by the operating system – usually kernel threads – and implement a specialised user-level scheduler that manages fine-grained activities – often in the form of fibres – on top of these threads.

However, kernel threads are not the right kind of abstraction of computing resources. Threads are designed to be vessels for concurrency – but what a concurrency platform actually needs are vessels for parallelism that lets it implement concurrency on top of them. The following fundamental characteristics are of particular concern.

Processes, Threads and Fibres

Unfortunately, the terminology concerning schedulable concurrent entities in the existing literature is an extensive melange. In this dissertation, we will refer to these entities as different kinds of threads.

The term thread was originally coined by the authors of the Mach operating system. Accetta et al. define a thread as ‘the basic unit of CPU utilization [...] roughly equivalent to an independent program counter operating within a task’, with a task being ‘an execution environment in which threads may run. It is the basic unit of resource allocation. A task includes a paged virtual address space and protected access to system resources’ [Acc+86]. The POSIX specification uses the term process instead of task, defining it as an ‘address space with one or more threads executing within that address space, and the required system resources for those threads’ [Ope18]. In this thesis, we adopt the POSIX nomenclature and consistently refer to such a system entity as a process.

We distinguish between two fundamental types of threads.

Kernel threads (also called lightweight processes) are entities that are managed and scheduled by the operating-system kernel, which makes their creation, switching and destruction a privileged operation. They were first introduced in the Thoth operating system, unfortunately under the name processes (as part of a team) [Che+79]. On a multiprocessor system, kernel threads can be scheduled by the kernel to run in parallel.

User threads (also called featherweight processes), on the other hand, are managed purely in user space by a runtime system that multiplexes them on top of a kernel thread. They are significantly cheaper than kernel threads. However, as the kernel is unaware of them, user threads of the same process cannot be executed in parallel. Fibres are specialised user threads that are scheduled only cooperatively, which makes them particularly lightweight. They are closely related to coroutines.

A mixed model called M:N threading attempts to combine the advantages of both techniques by multiplexing many user threads or fibres on top of a smaller number of kernel threads. In this case, thread scheduling happens on two levels in the system: both in the kernel and in user space. Conceptually, kernel threads can be regarded as a 1:1 specialisation of this model while user threads can be seen as an M:1 specialisation. More details on threading models, especially on M:N threading, are given in Section 3.1.
**Resource abstraction.** Threads provide a level of abstraction that is too high for the purposes of a concurrency platform, and features that are unneeded or even obstructive. For instance, thread migration between cores may counteract decisions made by the runtime system. How many kernel threads should we create for the runtime system and to which cores of the central processing unit (CPU) should we pin them? How do we deal with multiple competing applications without having to suffer from frequent context switches due to time sharing and oversubscription? As Knauerhase et al. remark: ‘Essentially, the runtime and the programmer work hard to avoid many of the services the OS provides’ [KCT12]. Similar questions arise in the context of memory management in system with non-uniform memory access (NUMA): a uniform virtual address space is easy to use, but it hides all locality information from the programmer; the operating system often bends over backwards to compensate this, for example by implementing a first-touch allocation policy [Mar+95].

**Scheduling issues.** Since the operating system usually only sees the scheduling entities managed by itself and has no notion of user-level control flows, it may make decisions that adversely affect the latter. For example, it may preempt a kernel thread at a time that is inopportune for the user-level activity associated with that thread.

**Blocking behaviour.** As a particular consequence of the operating system’s ignorance of user-level activities, performing a blocking system call such as an input/output (I/O) operation can cause the kernel-level thread to be tied up – resulting in the temporary loss of an entire virtual processor from the user space’s point of view. This phenomenon, called *thread-blocking anomaly* [SPH98], is sometimes worked around by runtime libraries through the (eager or lazy) creation of additional kernel threads, which solves the original problem but is expensive and can cause other scheduling-related headaches in turn.

In short, an approach based on kernel threads is less than optimal in terms of performance and scalability. As support for a fine-grained concurrent execution model is a cross-cutting concern that cannot be viewed in isolation, it should be accommodated in all layers of the software stack, including the operating system and the abstractions provided by it. Thus, a conventional general-purpose operating system with a monolithic, relatively heavyweight structure and a threaded execution model in particular may hinder or thwart our stated efficiency goals.

Invasive computing [Tei+11] is a systems approach that aims to provide support for fine-grained concurrency on massively parallel hardware, tackling the fundamental challenge from a somewhat different angle. OctoPOS, the parallel operating system for invasive computing, gives applications exclusive control over their computing resources, including CPU cores, and allows expressing concurrency in the form of lightweight control flows called *i*-lets. A detailed introduction to invasive computing is given in Chapter 2. The execution model of OctoPOS centres around *i*-lets as first-class citizens and mostly avoids by construction the issues outlined above. However, OctoPOS itself suffers from shortcomings concerning two aspects.

**Protection.** There is no clear distinction or separation between user and kernel space because OctoPOS currently has no notion of privilege levels – essentially, OctoPOS trades some degree of security for efficiency. A malicious application is theoretically able to take over the operating system and to interfere with other applications at will. A naive implementation of privilege isolation could be achieved by erecting a system-call boundary at the interface between application code and operating system. However, this would turn every operation related to scheduling and dispatching into a system call, at a price tag that would be prohibitive – effectively robbing OctoPOS of one of its core strengths, namely its very efficient handling of *i*-lets.
1.2 Contributions of this Dissertation

• **Flexibility.** OctoPOS gives users no freedom to choose their own custom scheduling strategies for their programs; they have to make the best of what the system gives to them. Moreover, the execution model of OctoPOS is fairly rigid in that it requires all user code to be implemented in the form of i-lets, making it hard to port existing concurrency platforms whose constructs do not map well onto the i-let model.

Finally, the following overarching general aspects are worth keeping in mind and should flow into the operating-system design.

• **Tailoring.** As Parnas states, “‘less demanding’ applications should not be forced to pay for the resources consumed by unneeded features’ [Par79]. The operating system should offer a feature set tailored to the requirements given by applications; to that end, it should be able to selectively adapt itself in a fine-grained manner, both statically and dynamically. This includes the ability to relax at the request of applications the security guarantees given to them.

• **Scalability.** Going back to Amdahl’s law, the creation and management of concurrency and synchronisation should be as lightweight as possible. This is first and foremost a task for the runtime system, but the operating system should not get in the way by creating unnecessary bottlenecks.

1.2 Contributions of this Dissertation

This dissertation presents AtroPOS, an operating-system design that offers efficient support for fine-grained concurrency in user applications and aims to address the aforementioned issues. Its name, short for *atrophied parallel operating system*, alludes to the fact that it emanates from OctoPOS, but in shrunken form with many features factored out into a separate user-space runtime system. A special focus lies on the interplay between the runtime and operating system, and on the interfaces facilitating that interplay. AtroPOS shares its name with the Greek goddess of fate and destiny, whose job is to cut the thread that symbolises a person’s lifespan – while AtroPOS itself cuts out the notion of threads as an abstraction of control flows.

Even though Atropos is also referred to as ‘the inflexible’ in Greek mythology, AtroPOS is designed to be a generic and versatile foundation capable of supporting the runtime systems of arbitrary concurrency platforms. The default runtime system is InvRT\(^4\), the *invasive runtime system*, whose name signifies an inversion of responsibilities: functionality that used to be part of the OctoPOS kernel is now in user space.

The following technical contributions are made in this thesis:

1. The invasive programming model as implemented by OctoPOS is reviewed; selective advancements and improvements for better efficiency, scalability and predictability are proposed and applied.

2. A minimal operating-system kernel is constructed that specifically supports fine-grained concurrency. With OctoPOS as a starting point, the boundaries between kernel and user space are redrawn and a structured reassignment of responsibilities between the operating system and runtime system is conducted.

\(^4\)InvRT is supposed to be pronounced like the word *invert*.
1 Introduction

3. A cross-layer execution model is designed for both user and kernel control flows, with a focus on the interactions between them. In order to couple the two layers with each other, a non-blocking system-call interface and a generic event-notification scheme are developed.

4. A tailoring mechanism for enabling and disabling privilege isolation per application and per CPU core, both statically and dynamically, is proposed and implemented.

5. To demonstrate the design's generality, a second concurrency platform, the Cilk Plus runtime library, is ported to the ArroPOS kernel interface.

6. Finally, the merits of the mechanisms and design decisions proposed above are evaluated and discussed with the help of both micro-benchmarks and real-world application programs.

1.3 List of Publications

The ideas on which this thesis is based have partly been presented in the following peer-reviewed publications.


1.3 List of Publications


In Erhardt et al. [Erh+11] and Erhardt et al. [Erh+14] I demonstrated as primary author in the context of embedded systems that ahead-of-time compilation of programs written in a type-safe language can yield efficient machine code that can run unprotected while still guaranteeing memory safety. Aspects of that idea were picked up in Drescher et al. [Dre+16], where we argue that memory protection and privilege isolation can be enabled and disabled on demand, depending on the requirements imposed by applications and on the guarantees given by them. I co-authored that publication’s part on on-demand security mechanisms in systems software and provided the introduction and overview of the context of invasive computing.

In a number of joint papers [Pau+13; Pau+14; Pau+15a; Pau+15b] the benefits of the dynamic invasion of computing resources were demonstrated using the example of computer-vision algorithms. I contributed code to OctoPOS, the operating system used in the evaluation, and co-authored the papers’ sections on OctoPOS and the general context.

The fundamental concepts and design ideas behind OctoPOS – its layered structure, the lightweight control-flow model and the use of dedicated acceleration hardware – were presented in Oechslein et al. [Oec+14], to which I made major contributions as co-author.

In Mohr et al. [Moh+15], a joint publication with the compiler group from KIT, we describe the co-design and close coupling between the operating system and the runtime system of a type-safe PGAS language. I served as the primary author from the operating-system group. A number of ideas from that paper are picked up and developed further in this thesis, albeit in altered form.

As co-author, I contributed ideas to Schmaus et al. [SES18], which presents an extensible runtime system for micro-parallelism that makes use of locality hints.
1.4 Thesis Structure

The remainder of this dissertation is structured as follows.

**Chapter 2 – Fundamentals and Background** Chapter 2 gives an overview of the context of invasive computing in general, and of OctoPOS and its design principles in particular. A number of interim conclusions are drawn with respect to the problem statement outlined in this chapter, motivating the redesign of the system-software stack that has led to the creation of AtroPOS and InvRT.

**Chapter 3 – State of the Art** Chapter 3 conducts a survey of related work that constitutes the state of the art in support for fine-grained concurrency. It summarises the principles on which concurrency platforms are built, and discusses a number of relevant platforms. Afterwards, related operating-system concepts are examined and an overview of existing systems in the field is given.

**Chapter 4 – Revisiting the Invasive Execution Model and Its Implementation** In this chapter, a number of aspects of the invasive execution model and its implementation in OctoPOS are re-examined. Selective efficiency improvements are proposed and the synchronisation primitives are redesigned for better temporal predictability.

**Chapter 5 – The Design of AtroPOS and InvRT** This chapter presents the design considerations that have flowed into the creation of the AtroPOS kernel and of InvRT, the invasive runtime system. Aspects highlighted include the dynamic configurability of privilege isolation, the kernel’s control-flow model and its interaction with user-level activities, and the communication facilities offered by AtroPOS. Moreover, the construction of the invasive runtime system is illuminated and the applicability of the overall approach to other concurrency platforms is discussed.

**Chapter 6 – Implementation Details** Chapter 6 provides details on a selection of relevant implementation aspects in AtroPOS, among them the kernel’s interface and execution model. Furthermore, the mechanisms for system calls, configurable privilege isolation, event notification and communication are covered. The internals of the invasive runtime system are also described, with a focus on InvRT’s interplay with the AtroPOS kernel.

**Chapter 7 – Evaluation and Discussion** In Chapter 7, the methods developed in the previous chapters are benchmarked on a number of machines. The test hardware and the evaluation scenarios, comprising both micro-benchmarks and real-world applications, are introduced. Measurement results are presented, discussed and interpreted with respect to the original objectives.

**Chapter 8 – Conclusion** Chapter 8 summarises the work and the results achieved, and concludes with an outlook.
Notation and Conventions

In this document, capacities of memories and sizes of objects are given with the units and prefixes defined by IEEE 1541-2002 [IEE09]. The design of flowcharts is largely based on the ISO 5807:1985 standard [ISO85].

Program code is typeset in a monospaced font with coloured syntax highlighting. Most code listings are written in C++, C or x86_64 assembly (in AT&T syntax). It is assumed that the reader is somewhat acquainted with these languages, including a selection of constructs introduced in the C++11 and C++14 revisions: lambda expressions, function attributes, constexpr and nullptr. For better readability, some code listings are slightly simplified through the omission of assertions, casts, namespace prefixes et cetera.

The digital version of this document is compliant with the PDF/A-3u standard [ISO12]. References, citations, URLs, DOI names, acronyms and abbreviations are clickable hyperlinks.
Fundamentals and Background

The thesis at hand originated from, and is based on, research that was done in the context of the German Transregional Collaborative Research Centre 89 "Invasive Computing". Sections 2.1 and 2.2 give an overview of that context, with a focus on the aspects relevant for this work. OctoPOS – the invasive parallel operating system – constitutes the germ cell from which AtroPOS has spawned. The design principles of OctoPOS are presented in Section 2.3. In Section 2.4, a number of interim conclusions are drawn from the invasive programming model and the design of OctoPOS, setting the agenda for the remainder of this dissertation.

2.1 Overview of Invasive Computing

Invasive computing is a systems paradigm that aims to leverage the computing power provided by future multi- and manycore systems. Driven by the trend towards processors that integrate several dozens or even hundreds of cores onto a single chip, invasive computing is an effort to master the challenge of remaining scalable in a world of huge potential for parallelism.

The name *invasive computing* stems from the notion that application programs dynamically acquire (invade) hardware resources according to their needs for processing power, memory or communication data rate: CPU cores, memory, communication channels, accelerator units and others. An application is given exclusive ownership of the resources it invades. This guarantee of exclusivity enables the program to act in a *resource-aware* manner: to exploit its application-specific system knowledge in order to enhance its efficiency by adapting its algorithms and data structures according to the resources it is holding. Resource awareness grants more control over the system to programmers, but also imposes bigger responsibilities on them. Teich et al. [Tei+11] define invasive programming as follows:

> 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 [current] computing environment, to then execute in parallel the given program using these claimed resources, and to be capable to subsequently free these resources again.

1https://invasic.cs.fau.de/
Hardware resources are assumed to abound. In contrast to conventional contemporary systems, which virtualise the hardware and perform temporal multiplexing, invasive computing provides only spatial multiplexing of hardware units and thus avoids the inherent costs of resource virtualisation. Applications compete with one another for a share of the available resources, with a system authority that aims to maximise the system’s overall efficiency deciding who gets what. As a consequence, resource awareness also means that an application may not always get all the resources it originally requested, and that it must be capable of acting accordingly.

Figure 2.1 shows the life cycle of an invasive program. In the *invade* phase, the application requests a set of resources such as CPU cores from the operating system. Based on the global system state and an optional set of constraints given by the application, the system decides which resources to assign to the application, and grants it a *claim* for these resources. In the subsequent *assort* phase, the application may adapt itself, for example by selecting the algorithm that is best suited for the number and topology of CPU cores claimed\(^2\). The claim’s CPU cores are then *infected* with the corresponding program code, thus executing the parallel algorithm. Finally, after the execution has finished, the application may either resize and reuse the claim for further computations or fully *retreat* from it, releasing the associated resources.

Invasive computing is a comprehensive approach that covers all layers from the application software via the compiler, runtime and operating system down to the hardware. While this thesis concerns itself mostly with the runtime and operating system and their interplay, it is influenced by certain aspects of the hardware architecture nevertheless. The following part first gives a rough overview of the invasive hardware design and then describes in detail the software architecture and components on which this work rests.

### 2.2 Invasive Hardware Platform

One of the primary questions that invasive computing attempts to address is how to design a manycore system that is scalable even with hundreds or thousands of cores. One key insight from existing systems is that a centralised bus that connects all hardware components quickly becomes a major scalability bottleneck. The NUMA approach commonly seen in contemporary multi-socket systems, where multiple NUMA domains consisting of a limited number of cores that share a common bus and part of the main memory are interconnected via a communication network, still keeps cache contents coherent across domains. While some expect that on-chip

---

\(^2\)Note that *assort* is not necessarily an explicit operation performed by the runtime system – in the simplest case, it may be implemented statically by the application.
cache coherence is not going away any time soon [MHS12], others argue that it hampers the system’s scalability. Invasive computing is situated in the latter camp: its prototype builds upon a tiled [Mat+10] hardware architecture and guarantees cache coherence only between CPU cores that are located within the same tile [Hen+12]. A tile constitutes the system’s basic building block; one processor chip is made up of multiple – possibly numerous – such tiles. The backbone of a tile is a communication bus that is attached via a network adapter [Zai+15] to a meshed network-on-chip (NoC) that forms a two-dimensional grid structure [Bec+12; Hei+14]. Communication latencies between tiles are dictated by the topology of the grid.

An invasive system comprises a number of heterogeneous tiles of different types. Figure 2.2 shows an example architecture consisting of $3 \times 3$ tiles. An ordinary compute tile contains a number of CPU cores sharing the bus with a slice of fast on-chip memory. Particular CPU cores called $i$-Cores [Hen+11] possess reconfigurable special-purpose instruction-set extensions. Certain tiles host no CPU cores for use by regular application programs but provide interfaces to external I/O devices such as an Ethernet controller, or to off-chip double-data-rate (DDR) memory modules. Another special tile holds a tightly coupled processor array (TCPA), a programmable hardware accelerator for parallel loop processing [Mud+12; Han+14].

All tile-local and global memories are mapped into a single shared physical address space. To uphold the scalability of the overall system even with a large number of cores and tiles, no hardware-based cache coherence is provided across tile boundaries. The software may choose to partition the global memory among the tiles, assigning by convention a disjoint slice of global memory to each tile. This way, all cores within the same tile have a coherent view on their tile-local memory and on the respective global-memory partition.

If an application spans more than one tile and wishes to access data from a remote part, it must resort to message-based communication. The NoC’s network adapter offers special hardware primitives for efficient sending of active messages [Eic+92] and for asynchronous data transfers
between tiles. Alternatively, but less preferably, accesses to remote memory can make use of explicit cache flushes and invalidations to keep the contents of caches and random-access memory (RAM) consistent.

Both the invasive programming model and hardware architecture place demands on the system software that cannot easily be fulfilled by a conventional operating system. Hence, a custom operating system called OctoPOS was developed with the goal of supporting the invasive programming constructs *invasive*, *infect* and *retreat* as first-class citizens on the one hand, and of allowing applications to efficiently exploit the invasive hardware platform on the other hand. An introduction to OctoPOS is given in the following section.

### 2.3 The OctoPOS Operating System

OctoPOS [Oec+14], the parallel operating system for invasive computing, was designed specifically to facilitate the use of fine-grained parallelism on an invasive hardware architecture. Its objective is ‘to let applications manage the available computing resources on a local scope and to provide means for a dynamic and fine-grained expansion and contraction of parallelism’ [Oec+11]. The name OctoPOS was inspired by the octopus, an intelligent creature with a highly developed nervous system, able to act in parallel using its tentacles and to quickly react to changing environmental conditions. For efficiency, scalability and predictability, the internal state and data structures of OctoPOS are mostly synchronised in a nonblocking manner [Her91].

OctoPOS is designed as a multikernel [Bau+09]: each tile runs a separate operating-system instance holding its respective private system state. Individual instances communicate with one another over the NoC via message-based remote procedure calls (RPCs) [BN84], forming an overall construct that bears similarities to a closely coupled distributed system. Accordingly, user applications whose claims are spread over more than one tile must be aware of their locality. Following the notion of a distributed system, when we speak of remote resources, hardware components, application parts et cetera in the remainder of this thesis, we refer to entities that are located on another tile.

Claims are represented as hierarchically structured, tree-like entities that describe not only a given set of computing resources but also reserved amounts of memory, along with their localities [Oec18]. The top level of a claim’s hierarchy (level zero) denotes which tiles contain resources belonging to the claim. Accordingly, each first-level subnode encompasses one subset of the claim’s resources that all share a cache-coherent view on memory. The second hierarchy level represents the (possibly heterogeneous) structure of computing resources within a tile; each of its leaves contains only resources of the same type plus the associated memory. These leaf nodes are called elementary claims; they are the tile-local data structures that each OctoPOS instance handles for bookkeeping and managing the resources in its own tile. OctoPOS offers a low-level interface for the *invasive* and *retreat* operations that allows the direct creation and manipulation of elementary claims. These system operations take as argument a concrete request for a type and quantity of resources. An application program can either make direct use of the low-level resource-management interface or it can task a so-called agent with bargaining for resources on its behalf (see below).

Unlike conventional operating systems such as UNIX derivatives, OctoPOS does not offer threads as its basic control-flow abstraction. Instead, it opts for a significantly more lightweight execution model based on *i*-lets: short snippets of code that can be executed concurrently – possibly in parallel – with one another. Technically speaking, an *i*-let is represented as a four-word tuple consisting of a function pointer, up to two word-sized arguments and a control word containing scheduling hints. A concurrent application written for OctoPOS splits up its work into individual packages, creates an
i-let for each such work package, and tells the operating system to execute the i-lets on a given claim. Essentially, i-lets share similarities with fibres – but they are treated as first-class citizens by the operating system, not implemented as a user library on top of long-running traditional preemtpible kernel-level threads. OctoPOS is designed to create, activate and switch between i-lets as efficiently as possible. An in-depth explanation of the execution model is given in Section 2.3.1.

OctoPOS is not a singular piece of system software but a statically configurable family of operating systems [Par76] that can be tailored both to the properties of the underlying hardware and to the requirements of applications. Inspired by the CIAO project [Loh+09] and reusing parts of CIAO’s tooling infrastructure, the code base of OctoPOS is implemented primarily in AspectC++ [SGS02; SL07], an aspect-oriented extension of C++. Configurability is achieved through the combination of aspect-oriented programming [Kic+97] with a family model and a feature model. The latter, written in the Kconfig language [SB10], specifies all configuration points along with their allowed values and interdependencies. The family model maps each feature to a selection of source-code files. To produce a concrete variant of OctoPOS, the build system performs the following steps:

1. Merge a predefined configuration skeleton with a hardware-dependent platform-description file into a fully fledged feature configuration.
2. Transform the source code base – which comprises regular C++ code as well as aspect headers – by picking a subset of the source files. The list of files is obtained by applying the aforementioned feature configuration to the family model.
3. Invoke the AspectC++ compiler to weave the selected aspects into the transformed sources, then translate everything into machine code.

The result is a tailored library operating system that can be linked together with one or more applications to yield a bootable Executable and Linkable Format (ELF) image.

The extensive configurability of OctoPOS allows the creation of a specialised operating-system variant for each type of tile. For example, an I/O-tile kernel image has to include drivers for DVI and Ethernet interfaces but may forego a user-facing system-call interface, whereas a normal compute-tile image needs no I/O-device drivers but must provide the complete infrastructure to run application programs. The advantage of an operating-system variant tailored for the very tile it runs on is twofold: on the one hand, its memory footprint is no bigger than absolutely necessary; on the other hand, the a-priori omission of unused code paths allows for more aggressive optimisation of the remaining relevant code portions by the compiler.

Figure 2.3 gives an overview of the invasive software stack and its layers. The OctoPOS kernel exposes a native low-level C application programming interface (API) and is bundled together with a partial C standard library (libc) based on Newlib as well as a minimal C++ standard library. Hence, applications for OctoPOS can be developed in C, C++ or any other programming language that has C bindings. However, writing invasive programs at such a low level of abstraction comes with the caveat that it requires the programmer to be intimately familiar with the distributed nature of the system, its memory hierarchy and other pitfalls. In particular, C and C++ offer no direct support for the notion of a partitioned global address space (PGAS) – placement of objects, correct handling of pointers and explicit access to remote data are all left to the developer.

Writing applications for OctoPOS can be eased in two ways: either through the use of a runtime library such as wpeX [Sch18] that wraps the operating-system API into a higher-level interface, or by employing a specialised programming language with built-in PGAS support. One example of such a language is X10 [Cha+05], a type-safe language for distributed programming that draws

---

3https://sourceware.org/newlib/
Figure 2.3: Overview of the invasive software stack.

The OctoPOS kernel and the agent system form iRTSS, the invasive run-time support system. Runtime libraries exist for developing applications in C, C++ or X10, with support for MPI. OctoPOS has been ported to various hardware platforms, including a Linux guest-layer implementation.

Inspiration from Java and Scala. An extension of X10 called invadeX10 was developed specifically to support the constructs of invasive computing, and its runtime system was ported to OctoPOS.

Invasive applications are expected to have varying resource demands over time. For a good utilisation of the overall system, hardware resources should neither sit unused for a longer period nor should an application hoard more resources than it currently needs. Therefore, applications are supposed to inform the system when their resource requirements change. The decisions which free resources to award to whom are made by a system authority implemented in the form of an agent system [Kob+11], where each application is represented by an agent. A request for a resource, expressed as a combination of constraints, leads to negotiations between the supplicant and other agents in a bid to maximise the system’s overall efficiency. For scalability reasons, agent bargaining is implemented in a distributed, decentralised manner. If the applications involved are malleable [FSS13; BMZ15], the agent system may also initiate a proactive redistribution.

The agent system offers a high-level interface for the invade and retreat operations. It builds upon the low-level functionality provided by OctoPOS and maps abstract constraints to concrete resource requests and elementary claims. In combination, OctoPOS and the agent system form the so-called invasive runtime support system (iRTSS): the agent system makes strategic global resource-assignment decisions, which are subsequently enforced locally by the kernel. Since resource bargaining is not in the focus of this work and the agent system is an optional part of the overall system, it will not be explicitly covered further in the remainder of this thesis.

### 2.3.1 Execution Model of OctoPOS

As mentioned above, the stated goal of OctoPOS is to make the handling of fine-grained concurrency, expressed in the form of i-lets, as efficient as possible. Hence, its execution model differs significantly from that of traditional operating systems in that the scheduler directly deals with lightweight i-lets as opposed to threads. Four different i-lets entities are distinguished: candidate, instance, incarnation and execution [Tei+11]. An i-let candidate is the combination of a piece of concurrent code and its actual arguments. For each candidate there may be one or more i-let instances tailored
for the concrete computing resources available. One of these instances is picked and passed to the infect function, at which point it becomes an i-let incarnation: an entity bound to physical resources and ready to run. Finally, when the dispatcher picks the incarnation from the run queue and puts it onto the processor, it turns into an i-let execution. The OctoPOS kernel concerns itself with the handling of i-let incarnations and executions.

An i-let is not unlike a fibre as implemented by some operating-system APIs (such as Windows) and user-level libraries, but there are certainly important differences. In particular, fibres usually conflate a control flow with its execution context whereas OctoPOS makes a clear distinction between the two. When referring to a control flow, we speak about a schedulable entity: an item that is put into the scheduler; when dispatched, it runs sequentially from an entry point (or resumption point, for that matter) until it voluntarily returns control to the scheduler or is forcibly preempted. An execution context, on the other hand, holds all the resources necessary to sustain the control flow during its lifetime. In the minimal case, it consists of a stack and a memory area dedicated for saving and restoring register contents in case of a context switch, plus a control block holding management information. Fibres normally have a 1:1 association between control flows and execution contexts, which has two important implications: on the one hand, every creation of a new control flow leads to the allocation of a context; on the other hand, switching between two control flows necessarily involves a context switch.

By contrast, OctoPOS makes a number of assumptions about the properties of i-lets that can be exploited in favour of greater efficiency:

- **Shortness.** i-lets are assumed to be short self-contained snippets of code rather than possibly long-running and complex.

- **Cooperativeness.** Since every given CPU core belongs exclusively to one application at a time, no protection against monopolisation of that core is necessary. All the application’s i-lets coexist side by side and do not preempt one another. An i-let in execution runs either until it terminates or until it cooperatively yields control of the processor.

- **Run-to-completion semantics.** Due to their brevity, it is supposed that i-lets usually run to completion and rarely block midway through their execution. Blocking is only possible at predetermined points in the i-let code.

With these assumptions in mind, OctoPOS breaks up the 1:1 association between control flows and execution contexts. When most i-lets run to completion, not every single one of them needs its own context. After an i-let has finished its execution and leaves no state on its stack, that stack can simply be reused by the subsequent i-let on the same core. In the best case, an entire sequence of i-lets can be executed on the same stack without ever switching contexts. Only if an i-let does block must its state be saved and the subsequent execution be performed in another context. When the blocked i-let is resumed, its context gets restored.

Contexts in OctoPOS have a fixed size $S$ and all lie in a consecutive piece of memory starting at address $a$. This imposes certain restrictions on the system’s flexibility; for instance, contexts have to be preallocated and it is not possible to dynamically create more contexts or to change a context’s size at run time [Oec18]. On the other hand, computing the current-context pointer $c$ from the stack pointer $p$, an operation that is needed frequently, is as simple as this:

$$c = a + \left\lfloor \frac{p - a}{S} \right\rfloor \cdot S$$

\[This is similar to the interrupt model implemented by some operating-system kernels, where each core has a single kernel stack associated with it. For a discussion of that model, see Section 3.5.\]
With $S$ a power of two, this boils down in assembly code to a subtraction, a bitwise and and an addition.

The life cycles of $i$-lets and execution contexts on a CPU core are depicted in Figure 2.4. When a core becomes ready to run user code, one context is taken from the free-pool and is attached to the core; it becomes active. The dispatcher fetches an $i$-let from the scheduler's ready-queue and executes it in the active context. If the $i$-let runs to completion and terminates, the context remains active and the next $i$-let is dequeued from the ready-list. If the $i$-let blocks, on the other hand, then its state is saved in the active context, the context is made inactive, and a new context is taken from the pool and attached to the CPU core for the execution of further $i$-lets. Resuming a blocked $i$-let is seamlessly integrated into the scheduling scheme – it is implemented with the help of a special wakeup $i$-let that receives the inactive context as its argument. The wakeup $i$-let is submitted to scheduling just like a regular $i$-let. Once its code gets executed, it performs a context switch back to the saved context, discards the previously active context and releases it to the free-pool. The unblocked $i$-let then picks up right where it left off.

An $i$-let can relinquish the CPU – similar to the `sched_yield()` system call on POSIX – by creating a wakeup $i$-let for its own context, putting it into the ready-list and then blocking immediately afterwards. This can be used, for instance, to implement a back-off mechanism for tight polling loops. To ensure the correctness of this operation, an obvious concurrency issue needs to be avoided: if the wakeup $i$-let gets scheduled to a different core, that core may start executing the wakeup $i$-let before the block operation has finished. In that case, the context restoration performed by the wakeup $i$-let must not happen before the save is complete. This problem is solved by marking the context with a Boolean in-transition flag, which indicates that the context is currently in an inconsistent state. Before restoring the context, the wakeup $i$-let must loop until its in-transition flag is unset [Oec18]. Similar issues arise when an $i$-let gets unblocked while it is still in the process of blocking, or when a context is inserted back into the free-pool and immediately picked up by another CPU core before the former core has completed the stack switch. In both cases, simultaneous use of the same stack by multiple cores is prevented through polling of the in-transition flag.

In short, a freshly created $i$-let carries no implicit state; it is a self-contained entity that can be freely passed around until it gets submitted to scheduling. It is not until the $i$-let is brought to execution

---

**Figure 2.4: State chart of the $i$-let execution model.**

Context states (green boxes) change depending on transitions between $i$-let states (yellow ovals). The $i$-let operations block, wakeup and exit are marked in the same colour as the respective context-state changes they entail. Chart based on Oechslein [Oec18, Figure 3.7].
that a context is bound to it, which results in a temporary 1:1 relationship between the control flow and the context. The i-let does not possess a control block of its own; all of its management data and run-time state are held within the context. When the i-let’s control flow terminates, the context is unbound and can be recycled. Provided no blocking is involved, binding a context to an i-let is a cheap operation that merely copies the i-let data structure into the context that is attached to the local CPU core. Unbinding is an implicit operation that entails zero cost. This is a major contributing factor to the light weight of the i-let model.

To express dependencies between the control flows of different i-lets, application code can resort to data structures called signals. Signals are used when a dependency exists between a group of i-lets that produce events and another i-let that wants to be activated once a certain number of events have been triggered. For example, in a fork–join scenario, a parent i-let may spawn a number of worker i-lets and then wish to wait until all workers have terminated. A signal resembles a counting semaphore [Dij65; Dij68] in that its core is a non-negative atomic counter – however, that counter is used in a slightly different way. OctoPOS offers two variants of signals: a blocking and a spawning one [Oec18].

- **Blocking signal.** A blocking signal offers three operations: `wait`, `signal` and `addSignallers`. The `wait` operation checks the value $v$ of the counter. If $v = 0$, then `wait` returns immediately; otherwise it blocks the i-let and stores a handle referring to its context in the signal data structure. As with the semantics of a private semaphore, only one i-let can call a signal’s `wait` operation at a time. The `signal` operation, on the other hand, decrements $v$ by 1. If $v$ thus reaches 0 and another control flow is already blocked in `wait`, that control flow gets unblocked. Hence, $v$ indicates how many times `signal` must be called before `wait` can continue. In the fork–join case, the parent would initialise $v$ with the number of control flows created. If that number is not known in advance, the parent can initialise $v$ with 0 and for every `infect` call `addSignallers`, which atomically increments $v$, with an argument of 1.

- **Spawning signal.** The semantics of a spawning signal are mostly identical to those of a blocking signal, but with one major difference: there is no `wait` operation. Instead of unblocking an existing blocked i-let, `signal` infects the claim with a new i-let once $v$ reaches 0. The signal data structure is initialised with the i-let in question. In a fork–join scenario, the parent code would be split up into two parts: a pre-i-let which initialises the spawning signal and forks off all the work, and a post-i-let which contains the code that is to be executed after the join. This scheme is appropriate if none or little of the pre-i-let’s state needs to be transferred to the post-i-let.

In summary, OctoPOS offers a unique execution model that exploits the exclusive reservation of computing resources to efficiently support fine-grained concurrency. The system scheduler treats lightweight i-lets as first-class citizens; the i-let model decouples control flows from their execution contexts. Signal data structures can be used by applications to model dependencies between i-lets. This thesis builds upon the execution model of OctoPOS and introduces a number of additional enhancements and optimisations.

### 2.3.2 Platform and Hardware Support

OctoPOS is designed to be portable across different hardware platforms. This is achieved through a layered architecture as shown in Figure 2.5. A hardware abstraction layer (HAL) abstracts from the underlying hardware and platform-specific driver code, allowing the bulk of the kernel’s
A hardware abstraction layer decouples the operating-system kernel from the underlying platform-specific code, allowing the core functionality to be written in a platform-independent manner. Kernel services are made accessible to user programs through an application programming interface. Note that the concrete extent of the kernel’s functionality is subject to configuration and tailoring.

Most regular operating systems such as Linux are portable to various processor architectures but essentially make a number of somewhat strict assumptions about the hardware design. For example, Linux was never designed to run on a non-cache-coherent tiled architecture; also, it is not particularly good at handling heterogeneous processor cores within the same system [Pit12]. OctoPOS, in contrast, is more flexible in this regard: it was built with the heterogeneous invasive hardware architecture in mind while supporting commercial off-the-shelf (COTS) systems at the same time. Currently ports exist for three diverse target platforms, with an additional port to AArch64 (64-bit ARMv8-A) being planned.

**Invasive SPARC LEON3**

SPARC LEON3 is the primary architecture for which OctoPOS was initially designed. It will play a subordinate role in the main part of this thesis, but is important nevertheless for understanding some of the essential concepts of OctoPOS and their origins.

32-bit SPARC LEON cores [Gai02] are employed as general-purpose processing elements in the invasive prototype design. Admittedly, SPARC can be considered a somewhat unusual choice of instruction-set architecture (ISA) for use in a future-oriented research platform. The LEON3 was selected because a VHDL implementation by Cobham Gaisler is available under an open-source licence [Cob18], which makes it ideal for developing hardware extensions such as the i-Core’s reconfigurable fabric. Each CPU core has a local first-level instruction and data cache in write-through mode; all first-level caches within the same tile are kept coherent by means of bus snooping.

OctoPOS and the invasive hardware were developed in parallel. Interfaces between the two were specified in a hardware-software co-design, resulting in the formation of special hardware components for offloading operating-system services. Two of these components have significantly influenced the programming interface of OctoPOS. Other hardware elements specific to invasive
computing such as the i-Core or the TCPA will not be discussed in more detail because they are of limited relevance to the overall design of OctoPOS.

**C/C** The core i-let controller is a hardware unit for the efficient scheduling of i-lets within a tile [Puj+11]. It consists of a per-CPU-core input port and first-in, first-out (FIFO) output buffer, plus a mapping logic that is programmed with a rule set given by the operating system. The rule set tells the C/C which of the local cores belong to which elementary claim. When the operating system schedules an i-let for execution on a given elementary claim, it writes the i-let and a numerical tag identifying the destination claim into the input port associated with the local core. The C/C’s mapping logic – essentially a lookup table – evaluates the rule set and the claim tag to determine the set of candidate destination cores. Based on a number of optional heuristics such as queue length or core temperature, it then selects one destination core from these candidates and moves the i-let into the FIFO associated with that core [PWH15]. In the simplest (and default) case, the elementary claim’s cores are supplied with work in a round-robin manner. On each CPU core, the operating system’s dispatch routine does nothing more than poll the local C/C output FIFO and execute all i-lets retrieved, going to sleep when no further work is available and receiving a wakeup interrupt from the C/C when new work arrives.

Since the C/C has a separate set of input and output registers for each individual core, no software synchronisation measures such as locks are necessary to protect them from concurrent accesses. Any possible contention is dealt with inside the C/C’s state machines, on the hardware level.

One potential issue with the C/C is that its design favours work-sharing scheduling over state-of-the-art work-stealing strategies [BL94]. The C/C makes an unalterable scheduling decision each time it receives an i-let. Given that different i-lets can have varying running times, the utilisation of an elementary claim’s cores may become unbalanced as a result: in the worst case, one core is still busy executing a long-running i-let and other i-lets pile up behind it while the output FIFOs of all other cores belonging to the same elementary claim run empty. Rebalancing of the output queues is not envisaged. It would be possible for a core to steal i-lets from another core’s FIFO, but then appropriate synchronisation measures would have to be added in software and the previously mentioned benefits would be void.

**Network adapter** The invasive network adapter is the interface connecting a tile with the NoC. It offers four classes of communication operations, ordered from primitive to complex:

1. **Load/store memory accesses.** When a CPU core dereferences a pointer to a remote address – that is, to local memory on another tile or to global DDR memory – the load/store request is forwarded over the NoC to the local bus of the corresponding tile. The result is sent back to the caller’s tile. Since the contents of all caches are kept in sync within a tile, all of that tile’s CPU cores have a consistent view on remote addresses. However, unless explicit cache invalidations and flushes are performed, there is no guarantee that one’s own view is up to date nor that one’s modifications are globally visible. Hence, using loads and stores to access memory other than a partition of global DDR memory is discouraged.

2. **Sys-i-lets** are a low-level mechanism that allows OctoPOS instances to communicate asynchronously with one another via RPC. A sys-i-let is a fixed-size data packet that is sent over the NoC, put into a predefined memory buffer by the receiving network adapter, and triggers an interrupt on one of the destination tile’s CPU cores. The operating system’s interrupt handler then processes the packet’s contents, which consist of a pointer to a system function and its
marshalled arguments. The RPC mechanism and its implementation based on sys-i-lets are explained in detail in [Oec18].

3. **Remote infect** is a hardware-accelerated means to spawn an application i-let on a remote elementary claim. The destination tile’s network adapter hands the incoming i-let directly to the CiC, which schedules it to a CPU core as described above. Directly coupling the network adapter with the CiC brings the advantage that accepting the i-let and submitting it to scheduling is performed with zero operating-system intervention. In particular, no interrupt is ever triggered unless the CPU core chosen by the CiC is asleep and must be woken up.

4. **Push data transfers** allow user applications to transfer a block of data from the local memory of one tile to that of another tile. The transfers work in a direct-memory-access (DMA)-like manner: given a descriptor consisting of a local start address, a remote destination address and a number of words, the network adapter on the sending side asynchronously copies the source buffer to the destination address. A notification mechanism allows the application to react to the completion of the asynchronous transfer. This is done with the help of a pair of i-lets that the sender passes to the network adapter along with the DMA descriptor: one i-let for the sending and one for the receiving side. The first i-let is enqueued into the local CiC as soon as the source buffer has been read completely; it can be used to free the source buffer. The second i-let is submitted to the remote CiC as soon as the DMA unit has finished writing into the destination buffer, at which point the computation can be started on the newly available data.

Invasive applications commonly make use of all four operations. Non-local loads and stores are generally performed only on one’s own partition of the off-chip DDR memory. A situation where sys-i-lets come into play occurs when an application wishes to spread to another tile and calls the operating system (either through the agent system or directly) to perform a low-level invade operation. OctoPOS will in turn issue an invade RPC to the destination tile, causing a sys-i-let to be sent forth and another sys-i-let containing a descriptor of the newly staked elementary claim to be transmitted back. The application will subsequently wish to utilise the remote resources by infecting them with i-lets and feeding the i-lets with data. This is usually performed in a sequence of steps, depicted in Figure 2.6, that involve a forth-and-back roundtrip of remote infects and a push data transfer. First, the claim on the destination tile is infected with an i-let that dynamically allocates a buffer and sends the resulting pointer back as argument to another i-let. That second i-let, executed on the sender-side claim, kicks off the asynchronous data transfer from the source buffer to the destination buffer, passing a pair of i-lets: one that frees the local source buffer upon completion and one that gets sent to the destination-side claim and starts the actual computation on the transferred data set, potentially spawning more i-lets there to exploit the available parallelism.

In summary, the co-design between OctoPOS and the invasive hardware components brings efficient offloading mechanisms for the scheduling of control flows and for cross-tile data transfers. The local activation of an i-let requires no synchronisation in software, and triggering a remote application activity can be achieved without having to interrupt the target CPU. This helps to minimise the latencies of such operations and to enhance the predictability of the computing system.

To facilitate the comparison with other operating systems, OctoPOS was ported to COTS 64-bit AMD/Intel platforms. The focus is on multi-socket NUMA systems with dozens of cores. While not all specifics of the invasive hardware can be reflected in such systems, the multikernel design of OctoPOS maps well onto the hardware architecture. The properties of a NUMA domain are roughly
2.3 The OctoPOS Operating System

![Sequence diagram of a cross-tile data-transfer operation.](image)

**Figure 2.6: Sequence diagram of a cross-tile data-transfer operation.**
The operation requires two roundtrips over the NoC: the first to allocate a buffer on the destination tile, the second to copy the payload and notify the sending and receiving sides. Asynchronous actions performed by the hardware are marked grey.

equivalent to those of a tile, with a group of processor cores sharing a cache hierarchy and a slice of closely connected memory. Hence, it suggests itself to deploy one operating-system instance per NUMA node.

A memory mapping similar to that of the invasive prototype is achieved with the help of the memory-management unit (MMU). Since every memory bank is tied to one NUMA node, the concept of global memory is not applicable on x86_64; only local memory exists from the operating system’s point of view. In terms of caching behaviour, x86_64 with its sophisticated cache-coherence protocols gives considerably stricter guarantees than the invasive hardware. With the notable exception of memory barriers, no additional software measures are needed to keep the view consistent that CPUs have on memory, even across the boundaries of NUMA domains. Distributed invasive programs touch only their respective tile-local memory and resort to message-based communication between different application parts, hence following a stricter separation regime than would be technically necessary. This has the upside that disadvantageous memory-locality decisions are avoided by design: where Linux resorts to heuristics such as a first-touch policy [Mar+95], OctoPOS always chooses the memory region nearest to the CPU accessing it, which will remain the nearest over the object’s entire lifetime.

Special accelerator units such as the i-Core and TCPA have no equivalent on x86_64. The CiC and the NoC adapter are emulated in software, with the HAL exposing the same interface as for the native hardware drivers. The CiC driver is replaced with a software scheduler that distributes i-lets in a round-robin fashion to the cores belonging to the corresponding destination elementary claim. The CiC’s per-core output FIFOs are substituted with lock-free multi-producer, single-consumer (MPSC) queues. That is, software synchronisation is necessary to ensure correctness in the face of multiple concurrent enqueue operations.

Cross-node communication is emulated with the help of shared memory. Non-local loads and stores need no special handling and no explicit cache-management instructions. Communication operations rely on per-node multi-producer, multi-consumer (MPMC) queues residing in the receiver’s local memory. Sending a sys-i-let is done by writing it into the destination node’s queue and triggering an inter-processor interrupt (IPI). For a remote infect, the i-let is wrapped into a sys-i-let; the interrupt handler on the receiving tile unwraps the i-let and submits it to the scheduler.
2 Fundamentals and Background

<table>
<thead>
<tr>
<th>Hardware component</th>
<th>Guest-layer equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Process</td>
</tr>
<tr>
<td>Tile</td>
<td>Process address space</td>
</tr>
<tr>
<td>Core</td>
<td>Thread</td>
</tr>
<tr>
<td>Interrupt</td>
<td>Signal</td>
</tr>
<tr>
<td>Memory</td>
<td>Shared memory</td>
</tr>
<tr>
<td>Tile-local memory</td>
<td></td>
</tr>
<tr>
<td>Global off-chip memory</td>
<td></td>
</tr>
<tr>
<td>I/O devices</td>
<td></td>
</tr>
<tr>
<td>Serial output</td>
<td>stdout</td>
</tr>
<tr>
<td>Ethernet interface</td>
<td>Network socket</td>
</tr>
<tr>
<td>Hardware accelerators</td>
<td></td>
</tr>
<tr>
<td>C/C</td>
<td>Software emulation</td>
</tr>
<tr>
<td>NoC adapter</td>
<td>Software emulation</td>
</tr>
<tr>
<td>i-Core</td>
<td>—</td>
</tr>
<tr>
<td>TCPA</td>
<td>—</td>
</tr>
</tbody>
</table>

Table 2.1: Mapping between invasive hardware units and their guest-layer equivalents.

For most of the components a simple 1:1 translation exists on Linux, with the exception of the special invasive hardware accelerators.

Push data transfers are carried out synchronously through the combination of a `memcpy()` call, a local `infect` and a remote `infect`.

It is possible to reserve one or more cores per node for the purpose of offloading scheduling, communication and other operating-system or hardware-emulation tasks onto them. This was examined and evaluated in a separate thesis [Sch18].

Linux i686 Guest Layer

As an alternative to the native ports, a 32-bit Linux guest-layer implementation of OctoPOS was built to decouple the development and testing of the invasive software stack’s upper layers from the progress of the underlying hardware. It aims to serve as an operating-system prototype that exposes an identical programming interface but offers two major benefits:

1. Large parts of the software stack can be run independently of the invasive hardware, particularly at a stage when hardware prototypes are not yet available or still unreliable.\(^5\)

2. It is possible to examine and debug applications running on OctoPOS with the help of the standard utilities available under Linux such as GDB.

Linux acts as an emulated hardware platform on top of which the guest-layer variant of OctoPOS sits. As shown in Table 2.1, hardware resources are represented in software by similar UNIX concepts. A tile and its local memory are modelled as a UNIX process and (part of) its associated protected address space. Each CPU core is represented by a POSIX thread; interrupts are implemented as UNIX signals. Global memory is mapped as shared memory into the address spaces of all tile-emulation processes; the local memories of other tiles are mapped as well to imitate the physical address-space layout of the invasive platform. Instead of a serial output port, messages are printed to `stdout`. An emulated Ethernet interface is built on top of UNIX sockets. CIC and NoC functionality are implemented in the same way as on x86_64.

\(^5\)This point cannot be overstated; an eloquent elaboration is given by Mickens [Mic13].
2.3 The OctoPOS Operating System

2.3.3 Dynamic Memory Protection in OctoPOS

To ensure confidentiality and integrity in the computing system, two types of isolation between software components are required: horizontal and vertical isolation. Horizontal isolation prevents an application from accidentally or deliberately accessing the state of other applications, while vertical isolation fends off attempts by user code to read out or modify operating-system data. Evidently, both dimensions of isolation are strictly necessary to be able to give any security guarantees for the system. Support for horizontal isolation in OctoPOS is being explored in a separate thesis [Dre20] and taken as a given in this thesis; providing efficient vertical isolation is a contribution of this dissertation.

While it is possible to implement isolation purely in software, operating systems commonly provide horizontal isolation by means of hardware-supported memory protection. In the simple cases, this is done with the aid of a memory-protection unit (MPU) that enforces access restrictions on memory regions defined by the operating system. With a more advanced mechanism such as an MMU, the operating system can place every application process into its own (logical or virtual) address space that contains only those parts of the physical memory that the process is allowed to access. OctoPOS opts for a different memory layout that is similar to that of a single-address-space system [Mur+93]: all applications and the operating system see the same memory mapping. However, OctoPOS is not a true single-address-space system in the literal sense of the word, such as Opal [Cha+94] or Mungi [Hei+98], as it makes use of the MMU to enforce different access permissions for different applications. The latter systems, aiming to keep context-switching overheads as low as possible, do not make use of the MMU's access-protection features but offer protection based on software capabilities instead. OctoPOS, on the other hand, does not need to worry about context switches. Once a CPU core has been invaded as part of an application's claim, it remains bound to that application until retreated from. During that time, it will only ever execute code that belongs either to that application or to the operating system, without having to switch page directories.

The memory-mapping scheme of OctoPOS is depicted in Figure 2.7. The graphic shows the situation on SPARC LEON3, where the physical pages are identity-mapped (that is, each page's logical address equals its physical address); on x86_64 a simple mapping is set up as described above. The i686 guest-layer variant of OctoPOS does not support memory protection. Application memory is managed in the form of a set of disjoint contiguous regions. Regions can be fixed-size or resizeable and have associated access rights; for instance, the .text region has a static size and r-x rights (in symbolic UNIX notation) whereas a heap region can grow and shrink and has rw- permissions. It is possible to create and destroy regions at run time. An initial page mapping for the operating system (not explicitly displayed in the figure) comprises the entire available memory and marks all of its pages as accessible; it is activated on the CPU cores that are not currently part of an application. For each application, a dedicated mapping is derived from that initial mapping by marking only the application's own regions as accessible. The application mapping gets activated on the cores invaded by the application. For lack of privilege isolation, the operating-system kernel's region remains readable and writeable to the user.

Although contemporary processors are optimised to handle paged virtual memory [Den70] as efficiently as possible, memory protection does not come for free. In particular, costs arise when the memory mapping of a parallel application gets changed: besides the modifications to the page table itself, outdated entries in the translation lookaside buffer (TLB) must be invalidated, causing TLB misses for subsequent memory accesses. Even worse, when a memory page is removed from an application's address space, it is necessary to update not only the page table and the local TLB, but also the TLBs of all other CPU cores belonging to that application – otherwise, later accesses by those cores to the now-invalid page might not get caught by the MMU, causing potentially...
Every CPU core has an identity-mapped view of the physical memory; in the example, cores 0 and 1 are reserved for application A while cores 2 and 3 are reserved for application B. For each application a dedicated page mapping exists that allows accesses to the regions belonging to the application itself and to the operating system.

security-relevant use-after-free issues. This so-called TLB shootdown [Bla+89] entails sending to all other application cores an IPI whose handler routine invalidates the corresponding local TLB entries. The originating core must wait until completion of the shootdown has been confirmed by all cores involved. This makes address-space changes a potentially costly operation that introduces jitter which may be unacceptable for applications with high demands for predictability.

For that reason, OctoPOS allows memory protection to be enabled and disabled at run time, determined by the applications' predictability requirements on the one hand and by the security demands posed and safety guarantees given by them on the other hand [Dre+16]. For example, if an application needs real-time guarantees, the system can place it exclusively on a tile and abstain from forcing it into a hardware-based memory-protection sandbox, sparing it the cost and unpredictability caused by TLB shootdowns. Programs that are written in a type-safe language such as Java or X10 and use a trusted runtime system do not necessarily need to be isolated either. The decision whether to enable or disable memory protection for an application is made by the operating system whenever the demands for isolation change – for example, when a tile gets invaded by a new application. It must be noted that a security requirement by one participant has the potential to affect all other applications: if an application demands to be protected, it must be ensured that all others are restricted from reaching outside of their own memory regions. When the system decides to let an application run unprotected, it activates the initial operating-system page mapping instead of the restricted mapping for that application. This way, no page-table updates, TLB invalidations or TLB shootdowns are necessary. For more details, the reader is referred to Gabor Drescher’s upcoming dissertation [Dre20].

 Critics may point to the defence-in-depth principle [Aik+06], but we argue that the significant trade-off between the costs and benefits of in-depth defence justifies granting the option to forgo it [Dre+16].
2.3.4 Runtime Systems

Application programs that run on OctoPOS can either directly use the C interface or they can resort to a higher-level runtime system built on top of OctoPOS. Support exists for message-based programming with MPI and for type-safe concurrent PGAS programming in the X10 language.

MPI

Message-Passing Interface (MPI) is a widely adopted standard that defines the interface and semantics of a set of library functions for communication between parallel processes [MPI93]. It abstracts from the concrete communication mechanism. For example, communication between MPI processes running on the same machine can be achieved via shared memory or UNIX sockets, whereas processes spread across different cluster nodes can use InfiniBand or TCP/IP sockets. By and large, MPI’s notion of parallelism is limited to the interaction between multiple individual processes, with an MPI process usually corresponding to a single-threaded UNIX process (commonly referred to as a ‘heavyweight process’). Hence, it is customary to launch as many processes as there are CPU cores available in the system or cluster. Although MPI offers no dedicated support for multithreading, modern MPI libraries are implemented in a thread-safe manner and do not hinder programmers from implementing their own shared-memory parallelism in their applications. For example, a hybrid program may launch only one process per node that exploits intra-node parallelism with the help of OpenMP and uses MPI to communicate with its peers on other nodes.

An MPI library was developed for OctoPOS to allow running unmodified MPI benchmark applications for comparison purposes. Instead of porting an existing implementation such as MPICH or Open MPI, whose design is rather UNIX-centric, it was written from scratch in a bid to exploit the API of OctoPOS as efficiently as possible. Communication is done with the remote infects and push data transfers described in Section 2.3.2. Each MPI process is mapped to a static single-core claim; dynamic invasion and retreat is not supported. As a consequence, the MPI runtime system for OctoPOS should be regarded more as a proof of concept than as an officially supported subsystem ready for production use.

invaX10

X10 [Cha+05] was chosen as the preferred high-level language for developing invasive applications since it is a type-safe language for distributed programming that gives the programmer a PGAS view of the system and its concepts map well to the invasive programming model. At its core, X10 is largely similar to Java, but with a number of enhancements. Besides syntactic sugar and type-system augmentations such as operator overloading, value types, type inference and constrained types, two unique features stand out [Sar+19]:

- **Fine-grained concurrency.** X10 provides the concept of activities for expressing concurrency. An activity is an X10 statement that is executed independently of the rest of the program. Like fibres, activities have their own state and their local variables. Every program starts with exactly one root activity that executes the main() method. Further activities can be spawned on demand using the async keyword. The runtime system executes activities concurrently, possibly in parallel if multiple cores are available. X10 has a hierarchical notion of termination: a statement is said to terminate locally when it has finished its computation, and to terminate globally when it has terminated locally and all activities it has spawned recursively have terminated globally. The program itself exits as soon as the root activity terminates globally.
Putting `finish` in front of a statement converts the statement's global termination into local termination – that is, `finish` waits until the statement and all of its transitive child activities have completed their computation.

If access to shared data needs to be coordinated, the corresponding code can be put into an `atomic` block. Atomic blocks are executed in a serialisable order. Another synchronisation mechanism anchored in the X10 language are clocks, a kind of barriers.

- **Places.** Places are the concept through which X10 represents locality. A place is an abstract location where data resides and activities are executed. Objects are tied to the place on which they are created. Accessing a remote place may involve communication and thus take significantly longer than locally. The `at` operation is used to execute code on another place. By default, `at` itself operates synchronously: the current activity is moved to the given place, runs there and, after terminating locally, returns to the original place. Data that is used by the remote control flow but belongs to the original place is packed together into a closure and copied to the remote place. The remote operation can be made asynchronous by combining `at` with `async`.

The combination of asynchronous activities with the place concept make X10 a programming language that supports a programming model that its authors call asynchronous PGAS [Sar+10]. Details on the official X10 runtime system are given in Section 3.3.1.

To support the invasive programming model, X10 was enhanced in three respects: the standard library was extended with constructs for handling claims and constraints, the X10 compiler received a new backend, and the runtime system was ported to OctoPOS. The X10 standard-library extensions make the claim concept accessible to programmers by modelling a claim as a set of one or more places [Bra+14], with each place corresponding to a low-level elementary claim. Within a claim, an application behaves just like a regular resource-unaware X10 program. The `infect()` method can be used to cross claim boundaries and launch an activity on another claim. The invasion of resources is achieved through calls to `invade()` or `reinvade()`, which take as argument an abstract description of the resource request in the form of a `Constraint` object. Constraints are represented in a hierarchical, tree-like manner and can express, among others, the desire for a given number of processing elements (PEQuantity), additional restrictions such as FPUAvailable, performance hints such as a ScalabilityCurve, and a logical combination thereof. The constraints are passed to the aforementioned agent system for resolution through bargaining among the agents of the applications involved.

The new compilation system, called `x10firm`, reuses the frontend of the existing X10 compiler and plugs into it a transformation pass that translates the abstract syntax tree (AST) into the Firm intermediate representation (IR), an entirely graph-based representation [BBZ11] whose level of abstraction is comparable to that of the well-known LLVM IR [LA04]. The Firm IR is then further processed by the `libFirm` compiler infrastructure, which supports a wide range of optimisations – among them whole-program optimisation – and offers backends for x86_64, i686 and SPARC, the three architectures also supported by OctoPOS. Some adaptations to `libFirm` were necessary in order to process X10-specific language features such as generic classes and native methods – that is, methods written in an unmanaged language such as C++ [Bra+12].

The OctoPOS port of the X10 runtime system is a relatively thin layer. Much of its original functionality, including scheduling, is left to the operating system [Moh+15]. Asynchronous activities are mapped to `i-lets` in a 1:1 fashion. A `finish` block is implemented with the help of a dedicated blocking signal that counts the number of still-alive child activities spawned recursively. Spawning an activity on a remote claim using the `at` expression may involve a preparatory step that serialises the referenced outer objects into a closure and copies the closure to the receiver. Likewise,
it may entail a follow-up step that transfers the result and signals the activity's termination back to its origin. Copying the serialised closure and launching the remote activity is done in the same way as previously outlined in Section 2.3.2 and depicted in Figure 2.6. The return value is sent back according to the same scheme, but in the opposite direction. Optimisations are performed, among others, for remote activities that do not incur data transfers because they do not necessitate a closure. Moreover, specialised data-transfer mechanisms were designed for the transmission of pointered data structures on non-cache-coherent hardware [MT17; Moh18].

In short, X10 is a language that is well suited for invasive programming due to its native support for PGAS architectures and for fine-grained concurrent activities. Its language constructs can be mapped natively to the mechanisms offered by the OctoPOS programming interface. X10 applications are type-safe and thus suitable candidates for running without memory protection as presented in Section 2.3.3.

### 2.4 Interim Conclusions

This section discusses design decisions made during the creation of OctoPOS, and draws a number of conclusions from them. With the problem statement from Section 1.1 in mind, it identifies the concepts that are worth keeping and points out the shortcomings and opportunities for improvement that set the agenda for this dissertation.

One thing that stands out about the resource-aware programming model of invasive computing is its conscious simplification of the way resources are handled by the system. Instead of virtualisation and temporal multiplexing, which require a complex abstraction and virtualisation layer to be hardwired within the system, invaded resources are strictly separated spatially. The application or omission of virtualisation is directly reflected in the abstractions of control flows: in commonplace operating systems this has led to the creation of threads as the abstraction of a virtual processor, with the shortcomings outlined in Section 1.1. OctoPOS gives applications exclusive control over their physical processor cores, thus eliminating the possibility that the kernel makes adverse scheduling decisions. At the same time, the assignment of CPU cores to application claims provides a natural view of the available parallelism: no pinning of threads to physical cores is needed, there is no possible oversubscription of cores, and multiple applications can coexist side by side. In addition, the multikernel approach of OctoPOS makes handling memory locality simpler and more predictable on NUMA machines.

The decision to make i-lets first-class citizens of the operating system facilitates exploiting the functionality offered by the GIC and network adapter [Oec+14]. The benefits of an operating system offering a native lightweight execution model for applications to use have been demonstrated [Moh+15]. Also, operating-system primitives themselves can be implemented as i-lets – a technique employed by the Ethernet stack, for instance. These design aspects make the invasive programming model well-suited for developing concurrent applications for highly parallel systems. Although OctoPOS was originally built with special hardware support for offloading scheduling and communication in mind, it also runs well on COTS systems – the class of hardware on which this dissertation puts a focus.

The combination of type-safe PGAS programming in X10 with optional memory protection on demand is a very useful concept – but currently only the horizontal dimension of isolation is implemented. In order to enforce the guarantees given by the security model of OctoPOS, an option for vertical isolation is needed as well. The crux is how to implement privilege isolation between user and kernel space without affecting the efficiency of the execution model. On the one hand, it is
evident that simply putting all of the kernel code and data into a privileged protection level would turn every single system operation from a regular function call into a system call, which would make supposedly lightweight i-let operations such as scheduling and dispatching prohibitively costly. System calls, traps and context switches have been shown to be far from inexpensive [And+91; Lie95] despite recent improvements through dedicated optimised processor instructions such as syscall/sysret. On the other hand, we now need to distinguish between kernel-internal and kernel-external control flows that are logically decoupled from one another through the system-call interface. This opens up an opportunity to design a specialised control-flow model for in-kernel activities and to allow applications to provide their own control-flow abstraction on top of the system-call interface – either in the form of i-lets or through the use of an alternative runtime system. Thereby it is possible to support arbitrary concurrency platforms even if their constructs do not map well onto the i-let model.

Following these considerations, the logical conclusion is a far-reaching redesign of the operating-system structure into two parts: a minimal kernel that manages hardware resources and exposes CPU cores, and an interchangeable runtime system in user space that provides support for lightweight concurrent user activities to applications. The resulting two entities are the AtroPOS kernel and the InvRT runtime library.

\[\text{Footnote: Note that kernel page-table isolation, the software mitigation proposed against the Meltdown attack [Gru+17], further adds to the system-call overhead because it involves changing the page mapping on every single kernel entry and exit.}\]
State of the Art

This chapter discusses the state of the art in support for fine-grained concurrency and provides an overview of related work. First, the fundamental aspects of threading and modern parallel programming are introduced, among them the graph-based model by Blumofe and Leiserson and the principles of work stealing. We subsequently examine a selection of contemporary concurrency platforms along with relevant implementation details. We then turn to operating-system support for such concurrency platforms and consider scheduler activations and asynchronous system calls, among others. Finally, a survey of existing operating systems in the field and in related fields is conducted.

3.1 Threading Models

One great challenge in the design of a system-software stack lies in the question of how to efficiently represent and manage control flows. In the best case, an operating system is flexible enough to provide support for multi-model parallel programming [SLM90]: it allows several parallel-programming models to coexist, possibly even within the same address space. To this end, in the Psyche operating system [SLM90] the kernel interface is tailored not for direct consumption by application programmers, but for use by thread packages or concurrency platforms that wrap the kernel’s low-level primitives into specialised higher-level abstractions. The design of Psyche attempts to minimise the assumptions the kernel has to make about the nature of the threads implemented in user space.

Today’s operating systems generally implement one of the following two models.

1:1 threading Commodity operating systems usually offer a 1:1 threading model, where every thread seen by the user-space application is represented by a corresponding entity in the kernel. Since the operating-system kernel is aware of all threads, the thread-blocking anomaly [SPH98] mentioned at the outset of this thesis is avoided by design. However, kernel threads generally carry a large run-time context and a lot of management state, and all operations concerning thread creation, destruction and switching necessitate a system call. The resulting overhead makes the 1:1 model
State of the Art

far too heavyweight for managing vast numbers of short-living threads. Also, the kernel's scheduler – usually a complex construct with multi-level priority queues – is not tailored for the cooperative scheduling of small, mostly run-to-completion work items. Some experimental microkernel-based approaches move all scheduling-related decision-making from the kernel into user space [FS96; Sto07]: the current thread gets executed until it invokes a blocking operation, at which point the calling thread tells the microkernel through a directed yield primitive to which destination thread to grant the processor. While it gives applications great flexibility in the choice of an appropriate scheduling policy, user-controlled scheduling still requires kernel intervention for switching between threads. The resulting overhead is too expensive for managing very fine-grained control flows.

M:N threading M:N threading is an alternative model that employs scheduling on two different levels of abstraction – leading to a so-called dual-concurrency model [Bla90]. A classic representative of this two-level scheme is SunOS [SS92], where a library (often referred to as a thread package) multiplexes M user-level threads onto a pool of N lightweight processes exposed by the kernel. User-level threads are handled entirely in user space. The pool of lightweight processes is managed by the threading library; it is dynamically grown and shrunk on demand. SunOS threads implement a generic POSIX interface that includes concepts such as signals and thread-local data.

Concurrent applications often do not require user-level control flows to be fully featured threads. Such applications typically use user-space abstractions that are even more lightweight than user threads, often called fibres. Fibres are scheduled cooperatively on top of a regular kernel-level thread, which makes them very cheap to create, destroy and switch [TB15]. On the other hand, fibres are often not optimally integrated into the overall system structure. For example, Windows fibres cannot safely call Windows-API functions because not all parts of the API were specifically designed for that use case: certain library functions place data in thread-local storage, where it may get overwritten when another fibre running on the same kernel thread calls into the library [SGG18].

Naively implemented M:N-threading systems where the two levels do not exchange state information regarding their respective scheduling entities are susceptible to the thread-blocking anomaly: when a user-level thread (or fibre, for that matter) synchronously performs a blocking system call such as an I/O operation, the kernel puts the underlying kernel-level thread to sleep. From the application's point of view, this results in an unnoticed temporary reduction in parallelism – something that should be avoided. We will discuss techniques to tackle this issue later in this chapter.

A concurrency platform is not the same thing as a thread package, but there are certain commonalities. The primary difference between the two is the fact that a thread package has a relatively low-level interface that directly exposes threads and coordination primitives, whereas a concurrency platform provides a set of higher-level and easier-to-use abstractions that shield the programmer from having to deal with the low-level details [Lei08]. Still, it can be said that the runtime systems of most concurrency platforms rely on some form of M:N threading: they schedule fibres, lightweight user threads or some other kind of work packages on top of a number of kernel threads.

3.2 Fine-Grained Concurrency

With regard to computations on a parallel computing system, one has to distinguish between two categories: data parallelism and task parallelism [RL75]. Data parallelism puts a focus on distributing a data structure across different processing elements or execution units, which then operate on the data in parallel. It is applied in parallelised loops and single-instruction, multiple-data (SIMD) vector operations [Fly72], for instance. By contrast, task parallelism focuses on distributing
3.2 Fine-Grained Concurrency

Fine-grained work packages called tasks\(^1\) across processors. Task parallelism is distinguished from data parallelism in that it signifies the execution of different tasks on the same or different data. Tasks are generally interdependent and require scheduling and coordination support. In practice, since most of today’s modern processors fall into the multiple-instruction, multiple-data (MIMD) category, real-world parallel applications are effectively situated somewhere on a continuum between data and task parallelism.

A lot of fundamental groundwork with respect to the scheduling of parallel tasks was laid in the 1990s at MIT by the creators of Cilk, on both the theoretical and the practical level. The most important contributions from that era are a graph-based model of fork–join parallelism and the formalisation of the work-stealing concept. The relevance of these contributions is reflected in the fact that in 2008, ten years after its initial publication, the seminal paper ‘The Implementation of the Cilk-5 Multithreaded Language’ by Frigo et al. [FLR98] received the ACM’s Most Influential PLDI Paper Award [ACM]. By today, many of the concepts originally developed for Cilk have found their way into other concurrency platforms.

### 3.2.1 The Graph Model of Multithreading

Blumofe and Leiserson [BL93] formally model a multithreaded computation as a directed acyclic graph (DAG)\(^2\), which allows them to reason about time and space efficiency. Although the authors refer to the scheduling of threads in their work, their findings are directly applicable to tasks or other lightweight entities such as fibres or i-lets; we will consistently use the term task in this section. As visualised in Figure 3.1, each node in the DAG corresponds to one instruction; edges indicate dependencies between instructions. There are three different kinds of edges:

- **Continue** edges (horizontal in the figure) represent the sequential ordering within a task.

- **Spawn** edges (vertical) denote the creation of a new task.

- **Data-dependency** edges (dashed) indicate that a data value produced by an instruction of one task is consumed by an instruction of another task, which necessitates coordination.

---

\(^1\)Note that the term task in this context has nothing to do with Mach’s task concept, which was introduced in Section 1.1: a Mach task is a heavyweight process, whereas task parallelism refers to small self-contained work packages.

\(^2\)DAG is occasionally also spelled dag.
In this model, each task equals a set of vertices connected through continue edges. The tasks and the spawn edges between them form an activation tree, depicted in Figure 3.2. The graph can be simplified by merging paths of serially executed instructions that contain no forks or joins into strands [Lei09], highlighted with darker colours in Figure 3.1. A computation is called strict if all data-dependency edges from a task go to its ancestors in the activation tree, and it is called fully strict if all data-dependency edges from a task go to its direct parent.

As a spawn is roughly equivalent to an asynchronous call, each task needs a different view of the stack, as is shown in Figure 3.2. The result is a so-called cactus stack (sometimes referred to as a spaghetti stack) whose layout follows that of the activation tree, but with all edges inverted. A contiguous view of a task’s slice of the cactus stack can be established through memory mapping, but contiguity is not strictly necessary: usually, a spawn merely entails taking a new frame from a pool or allocating it on the heap, then linking it together with its predecessor frame [FLR98].

With the graph model in place, Blumofe and Leiserson conduct mathematical reasoning about certain properties of the multithreaded computation. They define two important measures: work and span. Work is the total amount of time spent in all the instructions. It equals the time it takes to execute all parts of the program on a single processor and is thus denoted by $T_1$. Span corresponds to the longest path in the graph – that is, the graph’s critical path. Since the span is the minimum time required to execute the computation on an infinite number of processors (assuming no overheads), it is denoted by $T_\infty$. With work and span, we can give two lower bounds for the execution time $T_P$ on a machine with $P$ processors:

$$T_P \geq \frac{T_1}{P}$$
$$T_P \geq T_\infty$$

In other words, the execution can neither be faster than the work divided by the number of processors, nor can it be faster than the span. These two inequations are called the work law and the span law. When we rearrange the work law, we see that the speedup $\frac{T_1}{T_P}$ on a $P$-processor machine cannot exceed $P$:

$$\frac{T_1}{T_P} \leq P$$

We define the average parallelism $\overline{P}$ of a multithreaded computation as the ratio of work to span:

$$\overline{P} = \frac{T_1}{T_\infty}$$
It can be described as the average amount of work along each step of the critical path, and is a structural property of the computation. Obviously, perfect linear speedup (that is, $\frac{T_1}{T_P} = P$) is only achievable if the number of processors $P$ does not exceed the computation's average available parallelism $\bar{P}$: the introduction of more than $P$ processors would merely lead to underutilisation.

Blumofe and Leiserson apply previous work on DAG scheduling [Gra66; Gra69; Bre74] to prove that an upper bound on $T_P$ can be obtained by greedy schedules: schedules in which at each step, if at least $P$ items are ready, then $P$ items are executed, and if fewer than $P$ items are ready, then all of them are executed. With this, they formulate the greedy-scheduling theorem [BL94]:

For any multithreaded computation with work $T_1$ and dag depth $T_\infty$, for any number $P$ of processors, any greedy execution schedule achieves $T_P \leq \frac{T_1}{P} + T_\infty$.

Consequently, any greedy schedule yields an execution time that lies within a factor of two of an optimal schedule. Blumofe and Leiserson further demonstrate that it is possible to schedule strict multithreaded computations with linear speedup using a central run queue if the computation's available parallelism is $\Omega(P)$. However, as they acknowledge, this approach per se is of limited practical applicability since contention for the central queue is a major scalability bottleneck [ALL89].

### 3.2.2 Work-Stealing Scheduling

To circumvent the scalability issues inherent in a central run queue, it suggests itself to provide a dedicated scheduling queue for each processor (or worker) involved. Work may need to be migrated among processors in order to prevent underutilisation of the system in case of unbalanced queues. Two fundamental approaches exist for redistribution: work sharing and work stealing. With work sharing, some new tasks created by a worker are moved to another worker's run queue by their producer. This technique is used by the CICC, for example. In a work-stealing environment, on the other hand, the initiative for migration comes from the underutilised workers themselves: when a worker runs out of work, it helps itself to an item from another worker's queue.

The principal idea behind work stealing is thought to stem originally from research on the parallel execution of functional programs [BS81]. Intuitively, work stealing causes fewer migrations and better balancing than work sharing, as migration only ever takes place when necessary as opposed to preemptively. Formally, Blumofe and Leiserson present a provably good randomised work-stealing scheduling strategy for fully strict computations, with ‘good’ referring to efficiency in terms of time, space and communication [BL94]. If the parallelism of the application is far larger than the number of processors, the algorithm guarantees near-perfect linear speedup [Lei09]. Each worker holds its ready tasks in a double-ended queue (deque) with a top end and a bottom end. The queues reside in shared memory and are accessible from all workers. Each worker uses its local deque like a stack with last-in, first-out (LIFO) semantics, pushing tasks to the bottom and removing them from the same end. If its own queue is empty, the worker becomes a thief: it chooses a random victim and tries to steal a task from the top of the victim’s deque. If the victim has no work either, then the thief randomly picks another victim. This usage of a deque has the crucial advantage that accesses to the bottom end of the queue, which constitute the overwhelming bulk of the load, are performed by a single worker and thus do not need to be synchronised. Synchronisation is only necessary between a thief and its victim, and between different thieves. Consequently, work stealing provides dynamic, decentralised and reactive load balancing with generally low contention and a low task-spawning overhead, which makes it suitable for handling a large number of fine-grained concurrent tasks.

---

3Deque is commonly pronounced /dεk/; just like the word deck.
Work-Stealing Queues

There are several different variants of work-stealing deques. The first non-blocking implementation was published by Arora et al. [ABP98] and is based on a fixed-size array, a bottom index and a top index. Concurrent accesses to the top index are synchronised with the help of the compare-and-swap (CAS) instruction. To defend against the inherent ABA problem [Sto90], the top index is mangled with a tag. Since the bottom and top indices both move into the same direction and the array is not circular, every successful theft effectively reduces the deque's capacity by one. Arora et al. mitigate this potential overflow problem by setting both indices back to zero when the popBottom() operation finds the queue empty. Chase and Lev [CL05] present an improved deque variant that prevents overflows by design by building upon a circular array and using 64-bit counters (and the modulo operation) instead of indices. They also provide a means to dynamically grow and shrink the circular array on demand. Another extension [HS02] of the original deque by Arora et al. allows thieves to steal not a single item at a time, but up to half of the queue's elements.

In all the aforementioned deque implementations, which assume cache-coherent shared memory, the popBottom() procedure performs the same two steps at the beginning: first, it decrements the bottom index (or counter) to register its intent to take an element; then it compares it with the top variable to see if an element is actually available – if not, it resets the bottom variable. The order of these steps is critical: the load in step 2 must not take effect before the store from step 1 is globally visible, otherwise a concurrent thief may steal the same element the local worker is about to take. Hence, on processor architectures with relaxed memory consistency – that is, on all modern processors, including x86_64, i686 and SPARC – a memory fence is required between the two steps. This fence is not inexpensive and can impose a considerable performance penalty because it lies on the code path that is executed the most frequently. Acar et al. [ACR13] suggest replacing the shared deques with fence-less private ones and using explicit message-based communication for load balancing. Conversely, Morrison and Afek [MA14] propose a clever scheme with slightly relaxed semantics that elides the fence by exploiting detailed knowledge about the processor's microarchitecture: if the CPU implements the total-store-order memory model and its store-buffer size is known, a thief can determine whether it is safe to steal from a would-be victim by looking at the fill level of its queue, that is, at the distance between its top and bottom indices. If that distance is below a certain microarchitecture-dependent threshold (a fraction of the store-buffer size), the victim's store buffer may still be holding relevant values that are not yet globally visible. In that case, stealing would be potentially unsafe, so the thief aborts its attempt and picks a different victim.

Theft Strategies

When it comes to spawning a new work item, a choice has to be made: push the child to the deque and keep executing the parent, or execute the child and push a continuation for the parent to the deque? Both strategies have their advantages and disadvantages. The latter strategy – also called continuation stealing – aims to minimise the scheduling overhead borne by the work of the computation, and to move the overhead to the computation's span instead. While burdening the computation's critical path for the benefit of its work may appear counter-intuitive, Frigo et al. [FLR98] demonstrate that this trade-off pays off provided the computation's parallel slackness – the ratio of the average parallelism to the workers available – is sufficiently large. The extra cost is paid by thieves, not by the 'honest' worker. The underlying concept is referred to as the work-first principle [FLR98]. Continuation stealing typically relies on dedicated support from the compiler for automatically extracting the continuations at spawn sites.
By contrast, the former strategy – called child stealing – is more straightforward to implement and does not require special compiler support. It is better suited for scenarios where the expected number of thefts is very high or where there are a lot of workers [Guo+09]. Since thieves relieve the producer of parts of its load, this scheme is described as a help-first policy.

When deciding between the two strategies, a trade-off between stack size and queue pressure must be taken into account: in a scenario with deep parallel recursion, continuation stealing is at risk of overflowing the call stack because workers can descend very deeply; with child-stealing a worker executing a parallelisable loop with many iterations may push countless items to the deque before doing any actual work itself.

### 3.2.3 Summary

The state of the art in writing parallel programs is the employment of either language constructs or library infrastructure to split the application code into a number of concurrent tasks. These work items and the dependencies between them form a DAG. While not all possible classes of graphs are equally well-suited for parallel execution, it was demonstrated for fully strict fork–join task graphs that work-stealing scheduling is both effective and efficient. One prerequisite is an adequately high degree of available parallelism, which can be achieved by creating sufficiently many small (but not too-small) tasks. Programmers should be wary to use generic synchronisation primitives such as spinlocks or mutexes in their parallel code, and try to resort to the higher-level mechanisms offered by the concurrency platform instead.

### 3.3 Concurrency Platforms

The following section gives an overview of the most important and relevant concurrency platforms, which can be divided into language-based and library-based approaches.

#### 3.3.1 Language-Based Concurrency Platforms

There are countless programming languages that have built-in support for concurrency and parallelism – too many to name them all. We will pick four particular representatives that occupy different roles: Cilk as the trailblazing pioneer, Go as a modern concurrent language with a slightly different orientation, X10 as a type-safe language for asynchronous PGAS programming, and OpenMP as a popular language extension. We will also briefly touch upon the particularities of a select number of other language-based concurrency platforms.

**Cilk**

Cilk [Blu+95; FLR98] is the reference platform in whose context many of the concepts and techniques presented above were originally invented. It is considered to be one of the prime influences on all concurrency platforms that are in use today. The name Cilk is a pun on the word *silk* (‘nice threads’). Cilk has undergone various releases under different names, among them Cilk-5, Cilk++ and Cilk Plus. In the following, when we speak of Cilk without any further designation, we refer not to a specific release but to the overarching concepts of the Cilk language family.
State of the Art

The Cilk project originated from the group of Charles E. Leiserson at MIT. At its core, it is an extension of the C programming language that offers support for shared-memory fork–join parallelism through a set of special keywords with easy-to-use and clear semantics. In 2006, Leiserson founded Cilk Arts, a start-up company that took over the development and eventually launched Cilk++ [Lei09], which added support for C++, loop parallelisation and so-called reducer hyperobjects: an elegant way to avoid data races on non-local variables. A reducer hyperobject is a mutable shared object on whose state each concurrent strand has its own view, with modifications getting merged back (reduced) into the global object when the modifying strand terminates [Fri+09]. In 2009, Cilk Arts was acquired by Intel. Its technology was merged with Intel's array-notation and auto-vectorisation features, resulting in a simplified but functionally enhanced product named Cilk Plus. With Cilk and Cilk++ having been proprietary source-to-source compilers, Cilk Plus was integrated as a new frontend into the Intel C++ Compiler and into GCC, and its runtime library was published under a 3-clause BSD licence. In 2017, Intel announced that they would deprecate Cilk Plus and focus on OpenMP and TBB instead. Since then, the coordination of further development has moved back to MIT – now on the basis of LLVM [SML17].

Concurrency is expressed in Cilk Plus through the addition of three keywords to C and C++:

- **cilk_spawn** is used to introduce concurrency. Annotating a call to a function (or lambda expression) with **cilk_spawn** results in the creation of a new logical child activity that may be executed concurrently with its parent.

- **cilk_sync** acts as a kind of local barrier that blocks the parent until all its previously spawned children have finished. An implicit **cilk_sync** statement is executed when the function returns.

- **cilk_for** is a higher-level construct that can be used to easily parallelise a **for** loop. Its implementation details are deliberately left unspecified, but the compiler commonly translates it into divide-and-conquer recursion using **cilk_spawn** and **cilk_sync**.

Additional features of Cilk Plus include a notation **array[0:n]** to concisely express loop operations on entire arrays, and loop vectorisation, both automatic and programmer-guided. The restrictive semantics of **cilk_spawn** and **cilk_sync** allow the programmer to express only fully strict computation DAGs. An example of a simple parallelised recursive function written in Cilk Plus that computes the $n$th Fibonacci number is given in Listing 3.1. Obviously, the naive recursion scheme is not the most efficient way to implement **fib()** – but it is frequently used as a sort of parallel equivalent of the ‘hello world’ program.

With the work-first principle applied consistently in its design and implementation [FLR98], Cilk makes use of the continuation-stealing approach described in Section 3.2.2. For every spawning function, the compiler generates two clones: a **fast clone** and a **slow clone**. The fast clone is the one that gets called when the function is invoked normally. Being designed for minimum overhead, it contains no direct support for actual concurrent execution – for instance, **cilk_sync** is translated to an empty statement. This keeps the work costs down in the frequent case that no theft takes place. When a continuation does get stolen, however, the thief resumes its execution in the corresponding slow clone, which contains all the operations necessary for restoring the execution context and for synchronising with child activities.

In our Fibonacci example in Listing 3.1, the spawning statement in line 5 would be transformed roughly into the following sequence of steps for the fast clone of fib():

1. Create a continuation for the code that follows the spawning statement (lines 6ff).
2. Push the continuation onto the work deque.
3.3 Concurrency Platforms

Listing 3.1: Cilk Plus implementation of the Fibonacci function.

```c
unsigned long fib(unsigned n) {
    if (n <= 1) {
        return n;
    }
    unsigned long x = cilk_spawn fib(n - 1);
    unsigned long y = fib(n - 2);
    cilk_sync;
    return x + y;
}
```

Each step of the recursion spawns a child activity to concurrently execute the first recursive descent (line 5); the parent itself performs the other descent in line 6. The parent then waits in line 7 for its child to finish before it can compute the overall result from the two partial results.

3. Carry out the function call fib(n - 1) and store its return value in x.

4. Try to pop the own continuation from the deque again. If successful, the continuation was not stolen and is no longer needed; discard it and continue normal execution in line 6. Otherwise, the continuation was stolen and its slow clone is now being executed by another worker. Hence, terminate the current activity and return to the runtime system's scheduler.

One interesting property of Cilk is that it is a faithful extension of C/C++ [FLR98]. That is, it is possible to construct the serial elision of a Cilk program—a program that has identical semantics but operates strictly serially—by leaving out all occurrences of `cilk_spawn` and `cilk_sync` and by redefining `cilk_for` as a regular `for`. Moreover, when running a Cilk program on a single processor, its functions are executed in the exact same order as by its serial elision—with reasonably low overhead because only the fast clones are invoked.

The Cilk Plus runtime system is implemented as a library on top of Windows or a POSIX-compatible operating system. The number of workers is fixed and can be specified at application start-up, with the default being as many workers as logical CPU cores are available. The runtime system creates one kernel thread per worker. When a kernel thread performs a blocking operation, the result is a temporary reduction in parallelism. Cilk’s fine-grained activities are managed by fibres—either in the form of Windows fibres or implemented as custom plain C code using `setjmp()`/`longjmp()`.

Overall, Cilk is not the best choice for I/O-intensive data-centre and server applications or for programs with frequent and irregular access patterns to a lot of shared state; it is best suited for compute-intensive scientific applications whose structure favours regular fork–join parallelism.

Compared to Cilk, the i-let model of invasive computing is more generic in several respects—but also more complex. Cilk is specialised for programming fully strict multithreaded applications, which facilitates optimisations but also limits the language’s expressiveness. Conversely, with i-lets and the explicit use of signals it is possible to express arbitrary patterns of concurrency at a lower level of abstraction. Cilk assumes a shared address space and has no support for message-based programming or resource awareness, while the invasive programming model was designed expressly for distributed cross-tile applications and resource-aware programming with dynamically shifting resource allocations. An effort to map the Cilk constructs onto i-lets [Sch15] demonstrated impressively how extensively optimised the default runtime system of Cilk Plus is—something that the OctoPOS-based variant could not keep up with at the time. Said experiment resulted in the insight that it is better to let concurrency platforms implement their own specialised, highly
func fib(n uint32) uint64 {
    if (n <= 1) {
        return uint64(n)
    }
    c := make(chan uint64)
go func() {
        c <- fib(n - 1)
    }()
    y := fib(n - 2)
    x := <-c
    return x + y
}

Listing 3.2: Go implementation of the recursive Fibonacci function.

For synchronising the parent function with the goroutine that makes the first descent, a channel is created (line 5). The goroutine executes an anonymous function that performs the computation and writes the result into the channel (line 7). The parent reads the result from the channel (line 10), which implicitly blocks until the value is available.

Go

Go is a systems-programming language developed at Google. It is statically typed, memory-safe and garbage-collected, and offers features for concurrent programming based on Hoare's concept of communicating sequential processes (CSP) [Hoa78]. Unlike Cilk or X10, whose focus is on scientific computing and high-performance computing (HPC), Go's preferred use cases are in the field of scalable server applications running on manycore machines in data centres. Moreover, Go allows for more generic concurrency patterns compared to Cilk's fully strict fork–join model.

The primary concurrency construct in Go is the goroutine, a kind of coroutine or fibre scheduled on the user level. Effectively, a goroutine is a function running concurrently with other goroutines in a shared address space. A goroutine may be preempted whenever it performs a function call, in which case the runtime system executes an implicit yield. Goroutines communicate with one another through channels – shared FIFO queues used for passing messages of a certain type. Writing to and reading from a channel is a blocking operation, but asynchrony can be established through use of the select statement. Coordination between goroutines can be achieved through the WaitGroup data structure, whose semantics are very similar to those of the blocking signal provided by OctoPOS as introduced in Section 2.3.1. Just like a blocking signal, a WaitGroup wraps a counter that can be incremented and decremented, and offers a primitive to wait for the counter to become zero. However, unlike with the OctoPOS signal, more than one goroutine can wait for the same WaitGroup at a time.

Listing 3.2 gives a demonstration of how the recursive Fibonacci function from Listing 3.1 would be expressed in Go with goroutines and channels. Again, note that this is merely a simplified showcase whose implementation is not optimised for efficiency by any means.

Goroutines are designed to be lightweight, consisting of little more than a stack and an instruction pointer. This constitutes a difference to coroutines, which do not necessarily have an associated stack [Dig72]. A goroutine starts with a small preallocated stack (typically 2 KiB) that grows and
3.3 Concurrency Platforms

shrinks on demand. Originally implemented as segmented stacks [HDB90], this was later changed to an approach where the runtime system reacts to stack exhaustion by allocating an entirely new region at twice the size, copying the original contents and adjusting pointers as needed [Mor14]. This technique makes use of the already-present garbage-collection information and takes place exclusively within the runtime system, without requiring any direct operating-system interaction through operations and constructs such as memory remapping, guard pages or signal handling.

The Go runtime library implements a work-stealing scheduler for goroutines that follows a steal-half approach. Continuation stealing is performed thanks to support from the compiler. The scheduler builds upon kernel threads provided by the operating system, which makes it susceptible to the thread-blocking anomaly. As a workaround, Go employs a technique called lazy promotion: it spawns an additional worker thread when a blocking operation is called. Lazy promotion will be described in detail in Section 3.4.1. By default, the runtime system uses as many workers as CPU cores are available in the system.

From a Go program’s point of view, all I/O operations are blocking. ‘The Go ecosystem is built around the idea that you write against a blocking interface and then handle concurrency through goroutines and channels rather than callbacks and futures’ [Mor13b] – a design approach that had already been followed by the PEACE operating system [Sch88b] in the context of inter-process communication (IPC). While lazy promotion is presumed to work well enough for most common use cases, it fails to scale for network I/O, where thousands of requests may easily be pending at a time, which would lead to the lazy creation of thousands of threads. To address this bottleneck, the runtime system makes all network sockets non-blocking and uses the corresponding I/O mechanisms offered by the underlying operating system. A brief overview of these non-blocking mechanisms is given in Section 3.4.3. A dedicated kernel thread called the netpoller is responsible for handling incoming events in an infinite loop, identifying the corresponding waiting goroutines and setting them ready.

Even though Go is intended primarily for writing server applications, its goroutine concept can be applied for shared-memory parallel programming just as well. Go’s runtime system goes to great lengths to avoid the thread-blocking anomaly on the user level, pulling out all the stops because the mechanisms and interface offered by the underlying operating systems are not well-suited for the use case of Go. Moreover, the way the Go runtime system dynamically juggles with kernel threads is geared to utilising all CPU cores available in the system; it is less well-suited for resource-partitioned systems where an application gets assigned only a share of the resources and has to coexist with other applications.

X10

As the fundamentals of the X10 programming language have already been introduced in Section 2.3.4, we will not discuss them here for a second time, but focus on the technical aspects of its original runtime system (as opposed to the aforementioned invadeX10 runtime library). As a concrete example of what an X10 program looks like, a pertaining version of our less-than-optimal but demonstrative recursive Fibonacci function is shown in Listing 3.3. This code demonstrates how shared-memory parallel programming works in X10, language version 2.6 [Sar+19]. X10 differs from Cilk and Go in that it supports not only concurrent shared-memory programming, but also has constructs that facilitate programming distributed parallel applications.

The X10 runtime system creates one process per place and launches a number of worker threads in them, usually equivalent to the number of cores available. Runnable activities are managed by a non-preemptive work-stealing scheduler with per-worker queues. The runtime system follows
the work-first principle and performs work-stealing scheduling on a local basis: load balancing happens within a place, but not between places [TWL12]. A heap-allocated cactus stack that holds a frame for each concurrent scope is managed in addition to the threads' regular call stacks. When an activity blocks on a \texttt{finish} statement, its child activities are processed on the same stack – no context switch is necessary. Like Go, X10 tries to work around the thread-blocking anomaly through lazy promotion: the runtime system dynamically starts additional worker threads before executing blocking operations.

Several mechanisms exist that implement communication between places: shared memory, sockets, MPI and others. The concrete implementation can be chosen at compile time or run time. The official X10 compiler has two backends: a Java-bytecode generator and one that emits C++ code. The Java backend generates bytecode for a Java virtual machine (JVM) such as Hotspot, which uses just-in-time compilation to achieve efficiency. The C++ backend performs ahead-of-time compilation into unmanaged C++ code that is then processed further with a regular C++ compiler and linker, for example with the GNU toolchain, to yield a native binary. Both backends have their advantages and downsides – for instance, the JVM backend cannot optimise aggregate value types whereas the C++ backend can only perform minimal object-oriented optimisations.

**OpenMP**

OpenMP [DM98] is a popular extension to C, C++ and Fortran that has found widespread use in the fields of HPC and scientific computing. It is made up of compiler directives and a set of library functions that provide shared-memory multiprocessing. OpenMP originally supported only loop-level parallelism; fork–join task parallelism was added in version 2.0 [Ayg+07]. As of version 4.0, OpenMP can also handle vectorisation. OpenMP is often used in combination with MPI to write distributed applications for large clustered computing systems.

An OpenMP variant of our exemplary \texttt{fib()} C function is given in Listing 3.4. Note that just like in a Cilk program, it is possible to create a serial elision by ignoring the \#pragma directives. Each annotation of a block with one of OpenMP's work-sharing directives is compiled into code that produces one or more work packages at run time. The runtime system is responsible for executing such work packages on a configurable number of kernel threads. In contrast to Cilk, Go and X10, OpenMP follows the simpler help-first principle by default: a work package (or task) corresponds...
3.3 Concurrency Platforms

```c
unsigned long fib(unsigned n) {
    if (n <= 1) {
        return n;
    }
    unsigned long x;
    #pragma omp task shared(x)
    { x = fib(n - 1); }
    unsigned long y = fib(n - 2);
    #pragma omp taskwait
    return x + y;
}
```

Listing 3.4: OpenMP implementation of the recursive Fibonacci function in C.

The code is very similar to the Cilk Plus variant from Listing 3.1, but uses special compiler directives instead of inline keywords to fork and join tasks.

to an annotated block itself, not to its parent’s continuation. For the parallel execution of a loop, the programmer can specify an execution schedule. With a static schedule, the loop iterations are partitioned and distributed to the worker threads beforehand. A dynamic schedule leads to a dynamic division of the work through the application of work stealing.

The OpenMP directives are supported by virtually every respectable C/C++ compiler, including GCC, Clang, the Intel C++ Compiler and Microsoft Visual C++. Every compiler ships its own implementation of the runtime library. The invocation of a blocking system call from an OpenMP task is not handled in a special manner, which makes OpenMP applications susceptible to the thread-blocking anomaly.

Other Language-Based Concurrency Platforms

Chapel [CCZ07] and Habanero-Java [Cav+11] are often mentioned in the same breath as X10. All three fall into the same category insofar as they are designed for programming PGAS systems. While Chapel is a dedicated language in its own right, Habanero-Java is an extension of Java with features very similar to the constructs offered by X10.

Functional programming languages are considered to be especially suitable for concurrent programming because their principal structure makes it easier to write independent tasks that do not contend for mutable shared state. Erlang, a representative of a functional language, was created at Ericsson for the implementation of large-scale distributed telecommunication systems. Erlang borrows heavily from the CSP model [Hoa78], to the extent that ‘everything is a process’ [Arm03]. Processes in Erlang are very lightweight concurrent entities that have no shared state and get scheduled by a virtual machine. Process scheduling is preemptive and based on four priority levels. Preemption is only possible at specific preemption points: the runtime system counts how many reductions (roughly equivalent to function calls) a process performs and forces it to yield when the counter reaches a certain threshold. Communication between processes is achieved through asynchronous messages, allowing for actor-style programming [Agh85]. Unsurprisingly, the Erlang scheduler employs work stealing. Other examples of popular functional programming languages include Haskell [Mar10], Scala [Ode+06] and Clojure [KK09].
3 State of the Art

As a side note, Rust – Mozilla’s memory-safe language for systems programming, which has lately been generating significant interest – offers no built-in support for concurrency constructs such as fibres as first-class citizens. However, its type system, memory management and ownership model were carefully designed with concurrency in mind, making Rust very suitable for concurrent programming under the motto *fearless concurrency* [KN19]. Essentially, Rust does not predetermine a certain concurrency paradigm by default, but rather advocates building support for one’s own concurrency paradigm as a library.

3.3.2 Library-Based Concurrency Platforms

The primary advantage of a programming language with built-in support for concurrency control is that it gives the compiler the opportunity to perform arbitrary, potentially complex, transformations on the source code – a feature that is used extensively by the Cilk, X10 and Go toolchains, for example. Hence, it is foreseeable that the trend will shift further towards native concurrency support in the form of languages offering adequate constructs for multi- and manycore programming. By contrast, writing a concurrent application in a conventional language and building it upon a library-based concurrency platform requires no dedicated toolchain support: the programmer can use whichever standard compiler they prefer. Naturally, such platforms cannot offer all possible features, and programs making use of them may be somewhat more verbose to write – but they have been both powerful and simple enough for many use cases so far. Virtually all libraries presented in this section manage a pool of kernel threads and schedule lightweight tasks on top of them via a help-first work-stealing scheduler.

One example of such a platform is the Java fork–join framework proposed by Lea [Lea00], which was included into Java’s standard class library with the release of Java 7. Inspired in part by Cilk, the framework consists of a set of classes and interfaces that support a programming style where problems are recursively split into concurrent subtasks. In typically cumbersome Java style, the programmer defines a concurrent task by writing a class that derives from the base class `ForkJoinTask` and implements a `run()` method. The framework includes a straightforward help-first work-stealing scheduler. Every worker thread attempts to take `ForkJoinTask`s from its own queue before it steals from another worker. If thefts were unsuccessful, it takes work from a global input queue. Users of the Java fork–join framework must be careful to use only synchronisation classes that are advertised to cooperate with fork–join scheduling [TWL12].

Quasar\(^4\) is an open-source library that brings fibres and channels to Java in order to provide a framework for actor-based and CSP programming. It allows writing Java programs in a style that closely resembles idiomatic Go. With Project Loom [Pre18], there is an ongoing effort to bring fibres and continuations to the JVM, building upon the existing work-stealing `ForkJoinPool` from the fork–join framework.

Microsoft’s .NET Framework includes the Task Parallel Library (TPL) [LSB09] that supports both map–reduce and non-structured parallelism – the latter with the help of futures\(^5\) [BH77]. TPL is heavily based on generics and delegate expressions.

Meanwhile, a wide variety of concurrency libraries exist in C/C++ land. Intel Threading Building Blocks\(^6\) (TBB) is an open-source C++ template library that makes it easy to implement task patterns for parallel algorithms. TBB provides the programmer with basic algorithms such as parallel_for,
3.3 Concurrency Platforms

parallel_reduce or parallel_sort, concurrent versions of Standard Template Library (STL) containers, an optimised memory allocator, several mutex variants, atomic operations and task-management primitives. The library splits parallel operations into tasks and arranges them in a task graph. A cache-aware work-stealing scheduler maps these tasks onto kernel threads [SGG18]. TBB is fairly popular and is considered to be an easy-to-use, yet efficient means for developing parallel applications. Microsoft Visual Studio comes with a similar library, the Parallel Patterns Library (PPL), which aims in the same direction and shares many constructs with TBB – consequently, code written for PPL is partly compatible with code building upon TBB and vice versa.

HPX [Kai+14] is another open-source C++ template library that exposes a standards-oriented API to facilitate the programming of parallel PGAS applications. It encourages application programmers to make extensive use of futures. The scheduler is configurable and offers a number of scheduling policies, the default being NUMA-aware work stealing; it also supports work sharing and priority queues. HPX aims to be as standards-compliant as possible: it is implemented in modern C++14 and orients itself both at the STL and at the Boost project7.

In macOS and iOS, Apple introduced Grand Central Dispatch (GCD), a thread-pool library with a C API. Programmers push tasks into dispatch queues and GCD takes care of dispatching them to kernel threads [SGG18]. A task is expressed either as a function or as a block – a lambda-like object with a closure. Blocks are a non-standard extension to C and require special compiler support.

The Filaments thread package [EAL94], which was created to support the development of scientific applications, can be considered an early precursor of concurrency platforms. Filaments supports three different types of threads: run-to-completion, barrier-synchronised and fork–join. Like an i-let, a Filaments thread is a very lightweight non-preemptible entity that has no stack of its own: threads that are executed consecutively on the same virtual processor (called server) share the same stack. No context switching ever takes place even if a fork–join thread needs to wait for its children to terminate: the join operation is implemented as a polling loop whose body calls into the dispatcher and tries to execute another thread on the same stack. Hence, the joining thread’s context is implicitly saved on the stack; obviously, the available stack space must be large enough to potentially accommodate multiple levels of this kind of recursion. The Filaments library comes with a rather primitive (that is, pre-Cilk-era) work-stealing scheduler where both the worker and thieves remove items from the same end of the queue. An enhanced version called Distributed Filaments [FLA94] supports multiple computing nodes through distributed shared memory.

Concurrency libraries are a useful approach to supplementing a non-concurrent language with constructs for concurrent and parallel programming. Still, it is difficult for a library-based solution to be as powerful and flexible as a language-based concurrency platform. For instance, performing a blocking system call from concurrent code usually triggers the thread-blocking anomaly. Moreover, interaction with arbitrary (potentially legacy) library code can be problematic if that library uses blocking system calls or coordination constructs not supported well by the concurrency platform.

3.3.3 Concurrency Substrates

One issue with concurrency platforms built on top of traditional kernel threads is a lack of coordinated interplay when multiple concurrent applications are executed simultaneously on the same system. When the respective runtime systems greedily create as many workers as there are CPU cores, the system suffers from oversubscription, which can lead to frequent process switches and the induction of unnecessary interference by the kernel scheduler. To prevent such situations, it is

7https://www.boost.org/
useful to coordinate the concurrent application processes with the help of a concurrency substrate: a common layer between the runtime system and the operating system.

Lithe [PHA10] is such a low-level substrate that provides basic primitives and a standard interface for the efficient composition of parallel codes. Lithe allows parallel applications to coexist on top of a POSIX-compatible operating system without interfering with one another. Its two principal abstractions are harts (hardware threads) and contexts. A hart represents a physical (or logical in the case of simultaneous multithreading [TEL95]) processor core, with a fixed 1:1 mapping between the two; a context is a vessel for a computation being executed on a hart and allows for blocking operations on the runtime-system level. With ports of the runtime libraries of OpenMP and TBB to the Lithe interface, multiple applications based on the same or on different concurrency platforms can be made interoperable without requiring any changes to the application code.

Another example of a concurrency substrate is Microsoft's Concurrency Runtime (ConcRT) under Windows. It consists of a resource manager and a task scheduler. Microsoft's Asynchronous Agents Library (a library supporting actor-based programming), PPL and the Windows port of TBB all build upon ConcRT to coordinate different parallel workloads.

Operating systems such as OctoPOS that follow a spatial resource-partitioning scheme do not necessitate a dedicated concurrency substrate – that functionality is an integral part of the operating system's interface.

3.4 Avoiding the Thread-Blocking Anomaly

In a threaded operating system, the thread-blocking anomaly occurs when a concurrent task performs a blocking system call such as an I/O operation. As the kernel blocks the underlying kernel thread, this leads to a temporary reduction in the application's parallelism. The default implementations of Cilk Plus and OpenMP as well as virtually all library-based concurrency platforms simply ignore that issue and advise programmers not to do blocking calls from parallel sections of their programs. Such a restriction may be sufficient for shared-memory number-crunching applications that very rarely communicate or do disk I/O, but distributed applications, for instance, may communicate fairly frequently. Essentially, there is a need for a latency-hiding technique on the level of the operating-system interface. Some concurrency platforms attempt to work around the issue on the user level. However, a real solution necessitates cooperation with the kernel, either through a mechanism such as scheduler activations or through asynchronous system calls.

3.4.1 Lazy Promotion from Tasks to Threads

In the absence of an appropriate kernel mechanism, it is possible for a concurrency platform's runtime system to avoid the blocking anomaly on the user level through carefully constructed workarounds. This is an option if one has no influence over the design of the underlying operating system's blocking system-call interface but the runtime library comprehensively wraps that interface.

Concurrent Cilk [Zak+15], a third-party extension of Cilk Plus, employs a technique its authors call lazy promotion from tasks to threads. When a Cilk fibre becomes blocked, it gets promoted to a thread: the worker is detached from its kernel thread and a replacement worker is created instead that runs on a new thread and resumes the execution by stealing work from a random victim. After the blocking operation has finished, the previous worker is restored. Lazy promotion allows Concurrent Cilk to support concurrency patterns other than fully strict fork–join concurrency.
Concurrent Cilk also comes with an I/O library that provides variants of I/O operations such as `read()` or `write()` that block only the current Cilk fibre instead of the entire kernel thread.

Similarly, the Go runtime system introduces an extra layer of abstraction between goroutines and the workers that execute them: goroutines are multiplexed on top of virtual processors, which are in turn mapped to kernel threads [Mor13a]. The number of kernel threads typically exceeds the number of virtual processors, whose default value is the number of CPU cores available. When a goroutine is about to perform a blocking system call, the thread that is executing it hands off its virtual-processor context to another thread, which is either created on demand or taken from a pool. The other thread then starts executing other goroutines in the context of the virtual processor. After returning from the system call, the original thread tries to steal a context from one of the other threads to continue executing the goroutine that issued the call. If unsuccessful, it places the goroutine into a global run queue and becomes idle. Virtual processors periodically check that global queue in addition to their own local work-stealing queues.

X10 also follows a comparable approach that dynamically spawns an additional worker thread before performing a blocking operation.

A prerequisite for lazy promotion is the ability of the runtime system to intercept blocking calls and handle them accordingly. Hence, lazy promotion is more widespread in language-based concurrency platforms than in library-based ones – for the simple reason that the former class has far more control over the machine code that gets executed eventually. For instance, the compiler is able to inject management code at arbitrary program locations such as call sites, function prologues and epilogues – a technique that the Cilk, Go and X10 compilers use extensively. With a concurrency library that merely supplements existing program code by being linked against it, it is hard to impossible for the runtime system to intercept all calls to potentially blocking operations – especially calls coming from third-party libraries.

Either way, the lazy-promotion technique has to be considered more of a hack than a clean solution. Essentially, the runtime system uses kernel threads in unintended ways. A real solution requires a proper kernel mechanism such as scheduler activations.

### 3.4.2 Scheduler Activations and Related Concepts

On the kernel level, scheduler activations [And+92] are a well-known technique to construct a two-level scheduling system that avoids the thread-blocking anomaly. A scheduler activation corresponds to a virtual processor executing user code; to that effect, no more than one activation can be running on a CPU core at a time. A user-space thread package is responsible for requesting processors from the kernel and for mapping the application’s threads to the allocated processors. For every system event that affects the user address space, an activation is sent to user space. This allows the thread package to examine the situation and react to the event before jumping into the dispatch loop, taking threads from the ready-list and executing them as usual. The notification is done in the form of an upcall to one of four handler functions defined by the thread package. The following events cause the kernel to trigger a scheduler activation:

1. **Dispatching.** An additional core has been made available to the user address space. A new activation is launched on that core and directly jumps to the thread package’s dispatcher.

2. **Preemption.** A core has been taken away from the user address space. One of the address space’s remaining cores gets interrupted by a new activation that puts both the preempted and the interrupted user-level thread into the ready-list. The preempted and the interrupted scheduler activation are discarded.
3. **Blocking.** An activation has blocked in the kernel, either because it invoked a blocking system call or due to a page fault. A new activation is sent up with the old activation as argument for the handler. The handler saves the old activation’s user-level-thread state.

4. **Unblocking.** A previously blocked activation has become ready again. One of the user address space’s cores gets interrupted by a new activation that puts both the unblocked and the interrupted user-level thread into the ready-list. The unblocked and the interrupted scheduler activation are discarded.

Technically, every scheduler activation consists of one kernel stack and one user stack. The kernel stack is only used for setting up an upcall sequence and can thus be very small; the user stack is supplied by the thread package. Arguments to the handler function are passed via the user stack. For efficiency reasons, the kernel holds a number of preallocated activations in a pool.

Scheduler activations were originally implemented in the Topaz system [TSS88]. An M:N-threading mechanism that is conceptually very similar, but somewhat less generic, was simultaneously introduced in the Psyche operating system [Mar+91]. Instead of upcalls, it uses software interrupts to signal kernel events to user space. Software interrupts are executed on a dedicated per-core user-level stack and are assumed to be very short-running; nesting is prevented through software-interrupt masking. Information about run-time state, including an interrupt-disable flag and the location of the interrupt stack, is communicated between user and kernel space via a shared memory region mapped at a fixed address. In contrast to scheduler activations, the virtual processors managed by the Psyche kernel are long-living: they do not constantly get replaced whenever a new event is signalled. Memory consumption is lower because no pairs of kernel and user stacks have to be preallocated in potentially large numbers. Moreover, there are significant differences in how preemption is handled: Psyche gives user space a two-minute warning prior to a core’s preemption, allowing the application to clean up its state and avoid losing a core at inopportune times, for example while holding a spinlock in a critical section.

Scheduler activations quickly gained popularity after their publication and were regarded by the operating-systems community as ‘the right kernel mechanism for supporting the user-level management of parallelism’ [Bar+92]. They were incorporated in the Mach kernel as an optional mechanism to support M:N threading through the C-Threads library: Mach kernel threads can be marked or unmarked as scheduler activations [Bar+92]. Scheduler activations were also implemented in Solaris, BSD/386 [SS95a], FreeBSD (as a variation called kernel-scheduled entities [Eva00]), NetBSD [Wil02], and later in several research operating systems (see Section 3.6.1). An implementation for Linux 2.4 exists [DNR00] but was never merged into the mainline kernel.

Still, scheduler activations are far from uncontroversial. A frequent point of criticism concerns their memory footprint, as the operating system must hold a large number of activations ready. Scheduler activations were also criticised for being unnecessarily expensive [SPH98]. On the one hand, an activation’s kernel stack never gets actually used if the activation was created solely for the purpose of event notification. On the other hand, every single event notification entails a new activation – hence, if a new event occurs while an event is already being processed, the kernel must create another activation and interrupt the current one. Such a situation can arise, for example, when a thread gets unblocked that has blocked only very recently. In the worst case, a quick succession of events can lead to a cascade of activations interrupting one another. Sun removed support for M:N threading and scheduler activations with Solaris 9 in favour of 1:1 threading, citing issues with implementation complexity and inefficiency [Sun02]. Developers of thread packages built on top of scheduler activations also struggled with achieving full POSIX conformity – in particular with regard to the behaviour of signals – and encountered scalability issues, which eventually led to the removal of scheduler activations from both FreeBSD and NetBSD [Ras09].
3.4 Avoiding the Thread-Blocking Anomaly

In the context of real-time systems, Seo et al. propose an alternative mechanism in the Arx operating system [SPH98; Seo+99] that aims to tackle the issues of memory waste and interruption cascades. It builds upon the concepts of dynamic virtual-thread binding and scheduling-event upcalls. A virtual thread is a passive entity consisting of a kernel stack and a reference to a user-level thread. A new virtual thread is taken from a pool and dynamically bound to a user-level thread when the latter makes a system call, when a system call blocks or when a system call gets interrupted by an event; no new binding needs to take place when an event interrupts a thread that is running in user space.

Like scheduler activations, events are signalled to user space via upcall – but there is only a single scheduler entry point for all events, and its arguments are passed via a fixed-size lock-free queue residing in shared memory. The scheduler function consists of two parts: an ‘upper’ part that takes kernel events from the queue and processes them, and a ‘lower’ part that selects the next user-level thread to run. If a new event arrives while the upper part is being executed, it is appended to the event queue, but no upcall is performed – execution resumes in the event-handling loop. The lower part, on the other hand, is written in such a way that it can be safely aborted at any time. Hence, if the execution of that part gets interrupted, the kernel simply enqueues the event and then issues an upcall on the old user stack; with this, the state of the old scheduling operation gets discarded.

Seo et al. give an upper bound for the size of the event queue in statically configured real-time systems as a function of the overall number of threads in the system and the number of dedicated interrupt-handler threads.

Microsoft introduced a related concept called user-mode scheduling with Windows 7 [SGG18]. User-mode scheduling essentially follows a traditional 1:1 threading model, but tries to reduce costs by moving scheduler functionality into user space. It builds upon the fact that a Windows thread actually consists of two parts: a user thread and a kernel thread, each with its own stack and descriptor. With user-mode scheduling, user threads can be scheduled and switched without kernel intervention. When the user-mode scheduler hands control to another user thread, the corresponding switch of the underlying kernel threads is delayed until the next kernel entry [RSI12]. At that point in time, the kernel only has to take the most recent thread switch into account – all previous switches that had been performed by user space are irrelevant for the consistency of the kernel state. When a system call blocks inside the kernel, the operating system switches to a dedicated primary thread that returns into the user-space scheduler, which takes another user thread from its ready-list. When the system call finishes, the corresponding user-thread descriptor is put into a completion list residing in shared memory, from which it can be picked up by the user scheduler. User-mode scheduling is not intended as an API for application programmers, but rather as a low-level foundation for runtime systems such as ConcRT. Ultimately, user-mode scheduling could not fulfil the initial expectations in terms of performance gains and was dropped from ConcRT in 2012 in favour of regular 1:1 threading [Mir13]. One reason for this is that the creation and destruction of threads still involves kernel intervention and is hence not entirely lightweight.

### 3.4.3 Asynchronous System Calls

In abstract terms, scheduler activations and its relatives can be seen as a technique to implicitly turn blocking synchronous system calls into asynchronous ones, with the user-level thread package hiding the asynchrony from the application programmer. An alternative way to avoid the thread-blocking anomaly is the explicit use of asynchronous system calls. Instead of blocking in the kernel, system calls return immediately in the context of the calling thread and later send a notification to user space upon completion. This allows the application’s thread to continue running until it needs

---

8 It should be noted that the authors put a focus on single-processor systems; they do not mention how well their approach is applicable to systems with more than one core.
the result of a system call. Waiting for the completion of a system operation is handled entirely in user space. Most common operating systems offer some asynchronous mechanism for certain I/O operations. Obviously, asynchronous system calls cannot help in case of a page fault caused by a swapped-out memory page.

A precursor of truly asynchronous operation is the non-blocking I/O mode supported by TCP sockets in BSD and POSIX. In non-blocking mode, a read or write system call may return prematurely before it has completed all of its work, possibly even before it has done any work. The application itself is responsible for keeping track of how much progress has already been made and for repeating the operation on the remaining bytes that have yet to be read or written. A system call such as `select()`, `poll()` or `kevent()` can be used to temporarily put the thread to sleep while no sockets are ready to perform I/O [Elm+04]. Non-blocking I/O is both limited in its applicability and cumbersome to use.

With POSIX asynchronous I/O, the operating system offers a dedicated interface with clearly defined semantics that works for arbitrary file descriptors, not only for TCP sockets. I/O requests are expressed in the form of `struct aiocb` records. The asynchronous operation (`aio_read()`, `aio_write()`, `aio_fsync()`) either completes the corresponding request or enqueues it and returns immediately. The application can choose to be notified of the request’s completion via signal or function callback, or it can poll and suspend using `aio_suspend()`. Asynchronous I/O still covers only select use cases that involve reading and writing of file descriptors, but not general operations that modify the file-system state such as creating a directory or removing a file.

Capriccio [Beh+03], a scalable user-level thread package for use with high-concurrency servers, is built as a wrapper around the Linux implementation of asynchronous I/O operations. Capriccio intercepts blocking I/O system calls on the libc level and issues asynchronous calls instead. A similar approach is followed by the MassiveThreads library [NT14], but only on the basis of non-blocking socket I/O.

Some operating systems provide a generic interface that allows arbitrary system calls to be executed either synchronously or asynchronously at the user’s choice. A well-known representative of such a system is Symunix [ELS88]. Its stated goal is to provide asynchrony for any system call for which it makes sense, including I/O and file-system operations. The API of Symunix exposes three meta system calls: `syscall()` issues a system call and returns a value indicating whether the operation has already finished; `syswait()` blocks in the kernel until a system call’s completion; `syscancel()` cancels a running system call. A system-call incarnation is described by a control block that is allocated by the caller and contains the system-call number, arguments and flags, a pointer to a handler function and a status word. When an asynchronous system call finishes, the kernel disposes a SIGSCALL signal; the application’s signal handler then executes the user-supplied handler function. Applications that wish to avoid the thread-blocking anomaly should abstain from using `syswait()` and instead build upon a user-level context switch to wait for completion.

Elmeleegy et al. [Elm+04] implement an asynchronous system-call mechanism as a wrapper in user space. Their library, `liblai` (lazy asynchronous I/O), builds upon FreeBSD’s kernel-scheduled entities [Eva00]. It intercepts the blocked-event upcall to lazily create a continuation for system calls that are still active, and subsequently directs control back to the calling user thread. The descriptors of completed system calls can be retrieved by calling `lai_poll()`, whose semantics are analogous to those of the `poll()` system call. The primary use case for which `liblai` is designed are event-driven server applications built around an infinite event loop that dispatches event handlers. A separate library called `laiogen` [Cra05] was developed that implements the same interface but does not depend on scheduler activations. Instead, it manages a pool of worker threads and passes every asynchronous system call to one of these threads for asynchronous processing. Interestingly,
3.5 Kernel Execution Models

A comparison between `liblaiogen` on FreeBSD and `liblaiogen` on both FreeBSD and Linux showed that `liblaiogen` performs significantly worse on FreeBSD but equally well or better on Linux\(^9\), bearing witness that a lot of optimisation work has flowed into the thread implementation of Linux.

The introduction of generic asynchronous system calls was discussed in the Linux community for a while, with `fibrils` being contemplated as the abstraction of lightweight in-kernel control flows [Cor07a; Cor07b]. Another proposition was to misuse `fork()` inside the kernel to create a child process that immediately returns to user space instead of blocking, while the parent process continues in the kernel until the system call finishes [Cor07c]. A related idea are `syslets` and `threadlets` as a mechanism to create asynchrony on demand for the execution of blocking system calls, with the completion of such activities being signalled by putting events into a ring buffer shared between kernel and user space [Cor07d]. Despite intense debate, the above ideas never found their way into the mainline Linux kernel.

FlexSC [SS10; SS11], a modified variant of the Linux kernel, proposes a radically different model for the execution of system calls in event-driven servers: the user application commissions a system call by depositing a request descriptor in a shared memory page; the kernel manages a number of privileged `syscall threads` which concurrently consume these requests and process them – possibly on a different CPU core than the caller's. This `exception-less` system-call mode allows for batching requests and improves the temporal locality of the overall system, thus cutting down on both the direct and indirect costs of system calls. This results in an inherently asynchronous mode of operation with respect to system calls.

3.4.4 Summary

Calling a blocking system operation from a concurrent program is a potential issue that may needlessly lead to a temporary loss of parallelism for the application. Some existing concurrency platforms work around this anomaly on the user level. While there are kernel mechanisms such as scheduler activations or asynchronous system calls that help solve the issue by construction, they are not widely available in the operating systems commonly targeted by the established concurrency platforms. A co-design approach between runtime library and operating system would be helpful.

3.5 Kernel Execution Models

Two fundamental models exist for the way operating-system code is executed in the kernel: the process model and the interrupt model [Dra+91]. With the process model, the kernel manages a dedicated stack for every single kernel thread in the system. When a thread issues a system call or traps into the kernel, the corresponding handler code is executed on the kernel stack associated with the thread. Since the thread's entire in-kernel run-time state resides on the stack, blocking is a simple matter of pushing the current register contents and switching stacks. The thread can be preempted at any time while inside the kernel; this allows even kernel pages to be swapped out. Linux, Windows and most other commodity operating systems follow the process model.

Conversely, with the interrupt model, the kernel (also called an `event-based` kernel) keeps only one stack per CPU core at hand. System calls and traps are treated just like interrupts: all of them

\(^9\)The measurements in [Elm+04] and [Gra05] were both conducted on a single-core processor (possibly with hyper-threading enabled; this is not mentioned by the authors) and thus have little informative value with respect to the scalability of the approach.
are executed on the same stack. A kernel function that wants to block needs to be split into a pre- and a post-block function. The pre-function explicitly saves the thread's relevant execution state and creates a continuation for the post-function, which restores the state and resumes work. A continuation is an object that can be built, invoked and passed around; it can only call other functions or continuations, but can never return. With the interrupt model, threads are decoupled from kernel stacks; initially only one stack is allocated per core. Preemption is only possible at specific preemption points that must be introduced explicitly [For+99]. The interrupt model has the advantage that it is more frugal with respect to its memory footprint and has better cache/TLB-locality properties because it produces a smaller working set. It is best suited for microkernels with relatively uncomplicated system calls, where the kernel's primary job is to dispatch messages to user-level servers.

The Fluke kernel can be statically configured to use either the process model or the interrupt model [For+99]. The Mach microkernel implements a hybrid execution model that aims to combine the best of both worlds [Dra+91]. It allows kernel programmers to decide for each individual blocking operation whether to create a continuation or to block using the process model. If a thread needs to block inside a deeply nested function call or has a lot of state to carry over, or if it triggers a page fault while accessing a swapped-out kernel page, it gets suspended in the regular way by having its stack moved aside; a new stack is allocated and attached to the next runnable thread. Otherwise, if only little of the previous kernel context is needed after the wakeup, the kernel programmer can instead choose to create a continuation and store the context directly in the thread control block's scratch region (or in a bigger buffer allocated manually). In that case, if the next runnable thread itself has previously blocked with a continuation, the existing stack is handed off to the new thread for reuse; otherwise it is discarded. In addition, Mach facilitates an optimisation called continuation detection; that is, it takes specialised, faster code paths for certain well-known continuation functions. This can speed up cross-address-space RPCs through the elision of redundant checks that are statically known to have been performed by the caller just before the thread switch. Mach also uses continuations for returning to user space from a system call or trap. Moreover, infinite loops in kernel-internal threads that process messages can be converted into tail recursion with continuations.

As a side note, continuations were also introduced in Mach’s C-Threads library to implement control transfers on the user level [Dea93]. Continuations are used to handle user threads that are inactive – for example, waiting on a condition variable. When blocking, the user thread pushes the callee-saved registers onto its stack, then saves the current stack pointer in the continuation. To switch to the next thread, that thread's continuation is called, which restores the corresponding stack pointer. In this way, continuations serve a similar purpose and implement a mechanism similar to the wakeup i-lets in OctoPOS.

3.6 Survey of Related Operating Systems

The following section gives a brief structured overview of the vast zoo of related operating systems in the field.

3.6.1 Research Operating Systems for Multi- and Manycores

While 1:1 threading has largely established itself in commodity operating systems and their derivatives, various kinds of two-level scheduling are still an active topic in the operating-systems
3.6 Survey of Related Operating Systems

research community and have recently been experiencing a resurge – often in a mostly classic fashion, sometimes in combination with fresh ideas.

K42 [Kri+06; Sil+06], an IBM research project, explores an extensible and scalable object-oriented operating-system design for shared-memory NUMA systems. K42 has a microkernel architecture and is well known for its capability of being customised and updated dynamically at run time [Bau+05; Bau+07]. It follows a two-level scheduling approach; the pertaining kernel entities – called dispatchers [App+02] – are closely related to scheduler activations. Dispatchers are assigned to one of five priority classes and are scheduled accordingly in a preemptive manner; they may be migrated between CPU cores. K42 has a lightweight user-level thread abstraction. Arbitrary application-specific scheduling models can be implemented on top of that abstraction, including a full-blown POSIX-thread library.

Barrelfish is an operating system that aims to be scalable and to support heterogeneous CPU cores. To that end, it is structured as a multikernel [Bau+09]: every core runs its own instance of the operating-system kernel; all kernel state is replicated instead. This distributed design is more extreme than the hybrid structure of OctoPOS, where one operating-system instance exists per tile or NUMA domain. The operating system is factored into a privileged CPU driver and a user-level monitor process. Inter-core communication is carried out via explicit messages. On target platforms that offer cache-coherent shared memory, message channels are implemented in the form of shared-memory ring buffers inspired by user-level RPCs [Ber+91]: message slots are cache-line-sized; the receiver polls on the last word. Measurements have shown that this greatly enhances the system’s scalability in comparison to Windows or Linux [Bau+09]. Just like K42, Barrelfish follows a two-level scheduling approach and uses scheduler-activation-like dispatchers, but it does not allow dispatchers to be migrated to other cores. The default user-level thread package built on top of the Barrelfish kernel’s dispatchers offers a traditional POSIX-like programming interface. A customised version of Barrelfish with a library operating system on top provides dedicated support for database engines [Gic+16]: it schedules run-to-completion activities that are grouped together and co-scheduled on the same CPU socket. The corresponding CPU cores are exempt from preemption and all interrupts in order to provide for a low-noise execution environment. The multikernel design principle was also applied in Popcorn, a replicated-kernel operating system based on Linux that targets platforms with heterogeneous ISAs [Bar+15].

In the context of cloud computing, the factored operating system (fos [WA09]) targets manycore computers, with scalability as its primary design goal. Like OctoPOS, fos applies spatial multiplexing instead of temporal multiplexing. It consists of a microkernel and a set of operating-system services, which are separated into subsystems and implemented by server processes. Servers of the same subsystem are grouped into a fleet and are spatially distributed across the chip; each server runs exclusively on a dedicated core. A system call is effectively a message that is sent to a server and processed there. Cores that are not occupied by operating-system servers are made available to applications. The fos kernel provides a threaded execution model based on a cooperative scheduler [Wen+10].

The issue of application-directed resource management is addressed by the Tessellation operating system through its concept of space–time partitioning [Liu+09]. Tessellation wraps resources into performance-isolated containers called cells [Col+13], which provide an execution environment for application components and serve a similar function as claims do in OctoPOS. The Tessellation kernel is a thin layer, similar to a hypervisor, that serves as a basis for the implementation of higher-level abstractions in user space. Unsurprisingly, the resulting two-level scheduling system relies on an upcall mechanism related to scheduler activations. Tessellation ships two different user-thread packages: a preemptive one called PULSE and a cooperative one based on Lithe [PHA10].
ROS, which is closely related to Tessellation, focuses on a process model that facilitates scalability to large manycore machines [Klu+10]. Like Tessellation, ROS builds upon the concept of space–time partitioning. Resources such as CPU cores and memory are explicitly granted and revoked by the kernel. The concepts behind ROS eventually flowed into the development of the Akaros operating system [Rho+11]. The primary contribution made by ROS and Akaros is a new control-flow abstraction, the manycore process (MCP), which represents a virtual multiprocessor. An MCP’s virtual cores are gang-scheduled [FR92] by the kernel – that is, all of them are dispatched and preempted at the same time, allowing applications to always be aware of which cores are running. Cores are not preempted without prior warning and do not receive any interrupts that are not strictly necessary. Faults are redirected to user space, which is responsible for handling them. In contrast to kernel threads, which fundamentally represent concurrency, Akaros through its MCP model exposes parallelism to applications; management of concurrency is left to user space [Rho14]. Akaros was originally intended to be a purely event-based kernel, but later adopted the process model for pragmatic reasons.

The combination of Akaros with parlib, a framework for building user-level thread packages, results in a two-level model to which its authors refer as an M:C model [Klu15]: user threads are multiplexed not on top of kernel threads but on top of the cores of an MCP. All system calls in Akaros are asynchronous – even those that do not usually block in the kernel [Rho+11]. This allows for a unified system-call interface. If a thread-based application needs to wait for the completion of a system call, the corresponding thread blocks in user space. Event-driven applications can make use of the asynchronous interface without having to block. System calls are passed to the kernel in the form of an in-memory data structure – either through a trap on the local core or by being enqueued into a shared-memory ring buffer and picked up by another core. Events are communicated back to user space through unbounded concurrent queues (UCQs) designed for MPMC operation, with the consumers trusting the producers but not the other way round. A UCQ is a linked list of dynamically mapped memory pages containing event messages, synchronised with spinlocks. Separate locks are used by producers and consumers, so mischievous user code cannot provoke deadlocks in kernel space.

Both the Lithe substrate and the Go runtime system were ported to Akaros and parlib. Ideally, docking the Go runtime library directly on top of the parlib interface would have made its lazy-promotion workaround (see Section 3.4.1) superfluous. However, the authors made the decision to leave the runtime system largely unmodified and to build on top of a POSIX compatibility layer instead [Klu15].

### 3.6.2 Lightweight Kernels for HPC and Exascale Computing

In the HPC field, manufacturers and operators have long identified the need for specialised operating-system solutions. For them, mainstream operating systems offer an overabundance of features that are neither necessary nor useful but add unwanted noise and jitter to the execution environment. For this reason, lightweight kernels [Rie+15] have been built that put a strong focus on the feature set actually required by high-performance applications running on a clustered system, with the non-functional goal to minimise jitter and to leave as many clock cycles and as much memory for the application as possible. Lightweight kernels commonly strive to reduce the overheads imposed by both processor and memory virtualisation. They implement a simplified process management, often with only one task per CPU core and no multi-user operation.

Developed for IBM’s Blue Gene series of supercomputers, the compute-node kernel (CNK [Mor+05]) is one of the best-known lightweight kernels. CNK operates in single-user mode and performs
simple non-preemptive scheduling with a fixed number of threads per core [Gia+10]. Instead of a virtual-memory system, there is a single big address space with privilege isolation between user and kernel space, but no demand paging. There is no file-system support. CNK provides a POSIX-like programming interface.

Catamount [KB05], developed for the Cray XT3, is another lightweight kernel that was developed from scratch. It includes a severely stripped-down port of the GNU libc with no support for threading, dynamic process creation and code loading, networking or memory-mapped files. Complex requests such as file I/O are served by dedicated nodes that run Linux. Catamount’s successor Kitten [Lan+10] has a partially Linux-compatible interface with support for threads. It implements a simple round-robin scheduling scheme with no task migration. Kitten integrates a virtual-machine manager called Palacios that allows running other operating systems as guests next to native applications.

Most of today’s lightweight kernels aim to provide a certain degree of API compatibility with Linux. Some HPC operating systems such as Cray’s Compute Node Linux [Wal07] are based on a full-weight Linux kernel that was systematically stripped down to remove unneeded features. Other approaches such as McKernel or mOS couple a custom lightweight kernel with a Linux kernel that runs on a dedicated CPU core on the same node [Ger+15]. Simple system calls are processed directly by the lightweight kernel; complex requests such as network I/O are forwarded to Linux – either by delegating the call to a Linux proxy process that has an identical page mapping or by migrating the calling process directly to the Linux CPU core via Linux’ soft-real-time FIFO run queue. In the field of distributed-memory MIMD systems, delegating system calls from a high-performance kernel to a dedicated node was practised in the SUPRENUM project by connecting the PEACE kernel to a UNIX system [Sch88a].

The trend towards manycore processors and away from COTS multicore server CPUs, as mentioned at the outset, can also be observed in the HPC systems currently in operation or in development. As of June 2019, several of the cluster computers in the worldwide top-twenty list are composed solely of manycore chips such as the 68-core, 272-thread Intel Xeon Phi 7250 or the 260-core Sunway SW26010 [TOP19]. The next expected stage in HPC is called exascale computing – machines capable of executing at least $10^{18}$ floating-point operations per second, or one exaFLOPS. In that context, Intel’s Runnemeđe project [Car+13] explores a custom hardware and software architecture that bears many similarities with invasive computing, including a tiled chip design with a NoC interconnect, a multi-level memory hierarchy and relaxed cache coherence. Every tile-like entity, called a block, holds nine low-power processing elements: eight execution engines (XEs) for running application code and one slimmed-down control engine (CE) dedicated for runtime-system code. For efficiency, a simplified runtime environment is used that provides significantly less abstraction than a regular operating system [KCT12]. The issue of both horizontal and vertical isolation is solved through spatial separation at the network level: each block incorporates a programmable gate unit that provides range-based protection; privileged code is executed only on CEs. Data items are managed not on top of coarse-grained (that is, paged) virtual memory, but as explicitly placed and movable data blocks, which are treated as first-class citizens by the system. Runnemeđe implements an event-driven execution model based on codelets [Zuc+11]: fine-grained stateless snippets of code with run-to-completion semantics. In contrast to i-lets, codelets are organised in a graph structure (derived from dataflow graphs), with explicit inputs and outputs as their only run-time context. Codelets along with their data and control-flow dependencies are specified either directly by the programmer or derived from higher-level code by a compiler. The explicit visibility of these dependencies enables the runtime environment to perform clever optimisations such as task or data migration.
MyThOS, the many-threads operating system [Rot+16], is a modular operating-system architecture for HPC. It aims to reduce operating-system overheads that affect the scalability of parallel applications – especially the cost of thread creation and management. MyThOS is a modular, configurable distributed microkernel that shares similarities with a multikernel. The kernel itself performs only simple FIFO scheduling of its internal activities, which are called tasklets; the scheduling of software threads (called execution contexts) is up to the application. Tasklets have run-to-completion semantics and are organised into three priority levels: high-priority kernel tasks that process events such as interrupts or inter-core RPCs, regular user tasks, and low-priority idle tasks for carrying out clean-up work.

3.6.3 Pico- and Exokernel Designs

Operating systems are usually designed to provide a level of abstraction from the underlying machine that hides all the gory hardware details behind neat concepts such as threads, address spaces, files and sockets. Pico- and exokernels, by contrast, head in the exact opposite direction.

The pico-kernel design was propagated by PANDA [Ass+93a], a runtime package that aims to support distributed applications written in C+++. Panda was built to provide features such as multithreading, distributed shared memory and garbage collection in the form of accessible C++ abstractions. Its stated goal is to be flexibly adaptable to specific application requirements by exposing not a fixed programming interface but a user-definable interface family, allowing multi-model parallel programming. To this end, PANDA moves as much functionality as possible into the runtime package, leaving a very small operating-system kernel – a pico-kernel – in charge of two functional areas: protection and monopolisation, both of which require the privileged instructions of the processor. The corresponding fundamental kernel abstractions are page-based protection domains and virtual processors. On top of every virtual processor sits a runtime package that provides tailored functionality; several alternative runtime packages can coexist in user space [Ass+94]. There is a 1:1 mapping between runtime packages and virtual processors; the application itself is responsible for load balancing by migrating threads between its runtime packages. Device drivers and network-communication agents are implemented as service applications in user space; they can optionally be pinned to a dedicated processor.

Since most of the operating-system functionality resides in user space, the pico-kernel design greatly decreases the frequency of calls into the kernel. Interaction between a runtime package and the kernel occurs in three ways: system calls (implemented as traps), upcalls and shared memory regions. All kernel calls are non-blocking. No thread will ever block within the kernel, thus stopping the whole runtime package' [Ass+93b]. Exceptions and interrupts are forwarded to user space: immediate interrupts (including exceptions) are delivered via upcall; delayed ones are exposed through a dedicated read-only data structure.

While PANDA’s pico-kernel is related to the microkernel concept, the exokernel design [EKO95] goes to even greater extremes, motivated by the well-known end-to-end argument, which ‘suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level' [SRC84]. In other words, an exokernel implements only the absolute bare minimum of functionality and leaves as much to the application level as possible. It provides protection of hardware resources but leaves the actual management of these resources to application software, thus separating the two concerns of protection and management [Kaa+97]. Application programs can either make direct use of the extremely low-level interface provided by the exokernel, or they can link against a library operating system that offers a higher (and more familiar) level of abstraction.
An exokernel exposes resources and hardware mechanisms mostly as is, but it takes care of multiplexing and permission checking and implements an access-revocation protocol. CPU cores are time-sliced in a simple round-robin fashion. Interrupts and exceptions are dispatched to predefined handler functions supplied by the application. Virtual memory is managed by the application, but the kernel keeps control over the TLB and requires a system call for any page-table modifications. Applications can share a physical memory page by passing around a capability associated with the page. Network packages are demultiplexed to their intended destination with the help of a programmable packet filter. Disk blocks are managed by a kernel-internal buffer-cache registry, which binds them to physical memory pages. Exokernel implementations commonly follow the interrupt model: their abstractions are at such a low level that kernel system calls never have a reason to block [For+99].

Notable examples of exokernels include Aegis [EKO95], Xok [Kaa+97] and Corey [Boy+08], with the design of the Nemesis operating system [Han99] being closely related. The Xen hypervisor [Bar+03] and unikernels such as MirageOS [MS13] are also occasionally regarded to be variants of an exokernel [Raa05].

3.7 Summary

Most existing concurrency platforms are built upon kernel threads – but threads are the wrong abstraction for representing virtual processors that act as workers, executing concurrent tasks in parallel. On the one hand, threads offer features that are unnecessary or even counter-productive for our scenario, among them arbitrarily complex scheduling strategies and migration between cores (which often gets disabled through pinning anyway). On the other hand, they fall short in terms of functionality that would be helpful, such as scheduler activations or asynchronous system calls. This dissertation argues in favour of a co-design between the operating-system kernel and runtime libraries, resulting in a low-level kernel interface that is tailored for use by concurrency platforms and avoids the shortcomings of a threaded API. AtroPOS, the operating system that was developed in the process on the basis of OctoPOS, draws inspiration from many of the systems presented above.
Revisiting the Invasive Execution Model and Its Implementation

The invasive execution model is designed to efficiently create, distribute, schedule and execute large numbers of i-lets. In this chapter, the execution model and its implementation in OctoPOS are revisited, shortcomings are pointed out and selective refinements on both the conceptual and the implementation level are proposed. A particular focus is put on the aspects that are involved in synchronisation: as synchronisation times effectively come within the serial portion of an application, it follows from Amdahl’s law [Amd67] that the elimination of performance bottlenecks in the synchronisation code can benefit the application’s overall scalability. Another point of focus is the improvement of the system’s temporal predictability through an effort to make frequently used synchronisation operations wait-free. All improvements presented in the following sections were originally developed for upstream OctoPOS but have also been merged into the InvRT runtime library developed specifically for this thesis.

For a better understanding of the technical aspects described in this chapter, it is recommended to first get acquainted with Section 4.2 of Benjamin Oechslein’s (German-language) dissertation [Oec18], which describes in detail the implementation of the execution model in OctoPOS.

4.1 Context-Management Improvements

The efficiency of the execution model of OctoPOS springs from the light weight of i-lets and their execution contexts. Due to the cooperative nature of i-lets, a context switch is only carried out whenever an i-let execution actively blocks, yields or resumes; until one of these operations is executed, the same context remains active and is reused repeatedly. With the spawning-signal data structure in conjunction with a pre- and post-i-let as described in Section 2.3.1, it is possible in principle to write applications in a way that they never block and thus never need to switch contexts. In practice, however, it is advisable to consider the following trade-off: if the portion of the pre-synchronisation state that is still needed after the synchronisation point is large, an explicit transfer of that state to the post-i-let can be rather costly; in such a situation, it is often cheaper to resort to blocking instead. Hence, although blocking is ideally considered the exception rather
than the rule, it rightly still occurs frequently in invasive applications. This makes it worthwhile to re-examine the way OctoPOS handles execution contexts. In this section, a number of inefficiencies in the implementation of context management in OctoPOS are identified and resolved.

4.1.1 Context-Memory Management

In its original version, OctoPOS holds a predefined, statically configured number of fixed-size execution contexts in a contiguous memory area [Oec18]. Context allocation is managed with the help of a centralised free-list implemented as a shared bitmap, synchronised in a non-blocking manner via the atomic CAS instruction. This is a simple and straightforward scheme, but there is space for improvement:

- **Flexibility.** The number of contexts required at run time can vary greatly with the type of application that runs in the system – from only a handful up to several hundred for large programs prone to blocking. This makes it undesirable and wasteful to sweepingly prepare for the worst case by statically reserving the maximum number possibly needed.

- **Contention.** The central bitmap, which is accessed by all cores within the same tile, is a potential point of contention. Moreover, even in the absence of contentious accesses, every context allocation and deallocation involves a costly atomic operation on the bitmap.

The context-memory management in OctoPOS was reworked to address these shortcomings.

A minor efficiency improvement concerns the computation of the current-context pointer, an operation used frequently – for instance, every time an i-let is dispatched and bound to the context. This operation is already very cheap: provided the context size $S$ is a power of two, it takes three arithmetic instructions (see Section 2.3.1). However, with the additional constraint that a context must be aligned at a multiple of $S$, the current-context pointer $c$ can be computed from the stack pointer $p$ like this:

$$c = p - p \mod S$$

With $S$ a power of two, this is expressed in assembly code by a single bitwise and with a constant bit mask – see Listing 4.2 later in this chapter. A similar technique is used by Mach’s C-Threads package to find the current thread’s control block [Bar+92].

Execution contexts are now allocated dynamically using the system’s default memory-allocation mechanism. Once allocated, a context is held in a core-local queue with a configurable capacity that requires no synchronisation. When an i-let blocks, the runtime system first tries to efficiently snatch a context from the local queue; if that fails, it dynamically allocates a new one. When a context is discarded, it is pushed back into the queue of the local CPU core; if the queue’s capacity is exceeded, the context is released back to the memory allocator. A configurable number of contexts are preallocated at system start-up and distributed evenly among the queues.

With this two-level scheme, increased efficiency and predictability in the default case are traded for a potentially higher overhead and jitter in the exceptional case that a new context needs to be allocated or freed. The two knobs to twiddle are the number of preallocated contexts and the capacity of the queues. For applications with high predictability demands, both values should be set high enough (based on profiling data) that no dynamic allocation or deallocation takes place: if that condition is fulfilled, both fetching and releasing a context are constant-time operations.
4.1 Context-Management Improvements

4.1.2 Stack-Switch Optimisations

As explained in Section 2.3.1, one reason why i-lets are generally more lightweight than threads or fibres is the fact that they have no firmly associated runtime context: it is only when an i-let’s execution is immanent that a context is temporarily bound to it until the execution finishes. This is reflected in the structure and implementation of the dispatcher. When revisiting the state chart from Figure 2.4 on page 20, one can observe that stack switches are performed only in two particular cases:

1. An i-let execution blocks or yields – the current context is saved and a new context is activated.
2. A wakeup i-let resumes a previously blocked (or yielded) i-let execution – the current context is discarded and the blocked context is reactivated.

The Context class in OctoPOS provides two separate operations for switching between stacks: saveAndSwitch(), which saves the old execution’s register contents, switches to the new context and restores its register contents; and discardAndSwitch(), a stripped-down variant that forgoes the context save because the old context will be discarded anyway. The saveAndSwitch() primitive is used to implement case 1; case 2 make use of the optimised discardAndSwitch(). Both functions are written in assembly code that adheres to the C/C++ application binary interface (ABI); they save and/or restore only the callee-saved registers – the caller-saved registers are implicitly taken care of by the compiler. However, in both cases described above, the usage of these functions causes superfluous register saves and restores that are not strictly necessary.

A fresh context is prepared for execution by setting up an initial call-stack frame that returns control into a kick-off function which then calls the dispatch loop – but the only two register values actually relevant at that point are the stack pointer and the instruction pointer (the latter in the form of a return address on the initial stack). We define a specialised function saveAndStart() that performs a full context save just like saveAndSwitch() does, but after switching stacks restores only the instruction pointer. The resulting optimised context-switch scheme is depicted in Figure 4.1. Note that, as seen in steps 1 and 2 in the figure, this scheme allows register contents from the blocked i-let execution to survive the context switch and leak into the next i-let. This is not security-relevant because both i-lets are part of the same protection domain\(^1\) – it is only between different applications or from kernel to user space that information leakage must be prevented.

Another small efficiency improvement concerns the discardAndSwitch() function. As we have seen, that function does not save the contents of any non-volatile registers; it merely overwrites them with the CPU state stored in the target context. However, the compiler still emits spill code for the volatile registers around the call sites of discardAndSwitch(). By declaring discardAndSwitch() as \[\text{inline and } \text{[[noretunn]]},\] we advise the compiler to omit the unnecessary spill code and to embed the context-switch instruction sequence directly into the code that invokes it.

While these two optimisations are only minor in scale, they affect hot code paths that have the potential to be executed fairly frequently, which justifies the application of such micro-optimisations.

4.1.3 Post-Switch Hooks

As we have seen in Section 2.3.1, every context switch entails polling a Boolean in-transition flag located in the destination context. The use of this flag prevents a race condition where two

\(^1\)In fact, with OctoPOS’ implementation of the i-let execution model, both register and stack contents routinely leak between successive run-to-completion i-let executions belonging to the same application.
Figure 4.1: Optimised context-switch scheme: blocking and resuming an i-let execution.

This figure should be read clockwise from the top left. When a running i-let is blocked, the CPU state is saved on its stack (step 1). The new context is set up with a minimal call stack consisting only of a return address, and is activated (step 2). Later, in order to resume the blocked execution, the wakeup i-let discards the current register contents (step 3) and overwrites them with the saved CPU state from the blocked context (step 4).

Separate cores might temporarily run on the same stack because one core has not yet completed its pre-switch actions while the other is already switching to that very context. The in-transition flag is also interwoven with the wait operation of the signal data structure [Oec18]. The polling loop constitutes a section of serialising code and is an extra source of jitter, which we strive to avoid.

Inspiration: Enqueue Handlers in MAOS

To get rid of the loop, a technique is used that was originally proposed and implemented in the MAOS kernel [Mai13]. Its crux is that critical actions that would normally be executed before the context switch, and thus require appropriate synchronisation, are delayed until after the switch has happened. MAOS employs this technique during a thread switch to enqueue the parting thread into either the ready-queue or a wait queue. With this, a race condition is avoided where another core could take that thread from its queue and reactivate it before it had fully gone to sleep. This is done with the help of a hook method referred to in MAOS as an enqueue handler [Mai13].

MAOS uses the enqueue-handler technique as a generic building block for arbitrary synchronisation primitives, where it serves as an elegant non-blocking solution to the inherent lost-wakeup problem. The wait operation of such synchronisation data structures follows a fundamental pattern:

1. Check the wait condition; return if it is not fulfilled.
2. Do a context switch to another thread that is ready to run.
3. Enqueue the previous thread into the synchronisation data structure.
4. Recheck the wait condition; if it is now false, dequeue the previous thread and set it ready.
4.1 Context-Management Improvements

Listing 4.1: Implementation of ContextManager::resume() with a post-switch hook.

This function disposes of the current context and restores a previously saved context (named `to`). Through the use of a hook, it ensures that the former context is not inserted into the free-pool until after the context switch. The hook function is written as a C++11 lambda expression.

This sequence avoids two potential race conditions. Firstly, enqueuing the thread after the context switch ensures the aforementioned requirement that the thread must not be reactivated before its state is fully saved. Secondly, the renewed check in step 4 catches situations where the wait condition is concurrently revoked between steps 1 and 2; in that case, the thread is immediately woken again. The enqueue handler itself, which is implemented as a virtual method of the respective synchronisation class [Mai13], is responsible for executing steps 3 and 4.

Post-Switch Hooks in OctoPOS

The improved variant of OctoPOS’ context-switch code draws heavily on the enqueue-handler concept but implements it in a somewhat different fashion. Unlike the virtual enqueue() method in MAOS, which is tied to a class and is invoked through polymorphism, the mechanism in OctoPOS operates at a lower level of abstraction, with an explicit function pointer termed a post-switch hook. The hook function receives one or two arguments: a pointer to the previous context and optionally a pointer to arbitrary data. A pointer to such a hook function (plus optionally its extra parameter) is passed as argument to the respective context-switch primitives, which perform the switch before calling the hook. Post-switch hooks are employed as a generic mechanism to execute arbitrary code after a context switch. An example of how such a hook is used in OctoPOS is given in Listing 4.1.

To see how this translates into concrete assembly code, one can examine the implementation of the wakeup-i-let function, which takes as argument a pointer to the blocked context it is supposed to resume. In order to do so, it calls the ContextManager::resume() method shown in Listing 4.1. As can be seen in Listing 4.2, the compiler performs aggressive function inlining and even manages to statically resolve the hook-function pointer passed to Context::discardAndSwitch() to its concrete symbol. What remains is the absolute bare minimum instruction sequence necessary to resume a blocked i-let execution. It is not possible to embed the hook function itself because the hand-crafted assembly code from Context::discardAndSwitch() is sensitive and could easily break if arbitrary C++ code were inserted between the stack switch and the return. The stack pointer is temporarily adjusted for the hook-function call because x86_64 expects stack frames to be aligned at a multiple of 16 bytes [Mat+14].

In summary, the post-switch hook is a technique to eliminate the in-transition flag and the associated polling loop in context switches. Together with the context-management modifications and the stack-switch optimisations presented above, it selectively improves the non-functional properties of the handling of execution contexts in OctoPOS. We will pick up on the concept of post-switch hooks in Section 4.2.1, where it is applied to implement wait-free synchronisation primitives.

2Sebastian Maier, the author of MAOS, was also the one who suggested using that technique in OctoPOS; hence, the contribution described in this chapter is not the idea itself but the way it is implemented and used in OctoPOS.
Listing 4.2: C++ source and the resulting x86_64 assembly code of the wakeup-i-let function.

Except for the hook itself, all code has been inlined – including ContextManager::resume() (see Listing 4.1) and Context::discardAndSwitch(). The attentive reader will deduce from the constant bit mask that the context size is 256 KiB.

Wait-Free Synchronisation and the CAS Instruction

Herlihy defines wait-freedom as follows [Her91]:

A wait-free implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, regardless of the execution speeds on the other processes.

In a non-blocking algorithm that is implemented with a retry loop controlled by a CAS instruction, it is generally not possible to give an upper bound for the number of loop iterations unless comprehensive knowledge of the software system as a whole – especially of the maximum number of simultaneous processes involved – is available. The same is true for loops based on load-linked/store-conditional (LL/SC) instead of CAS. Consequently, such an algorithm is not wait-free: it may take an arbitrary number of iterations to succeed, or might even starve in case of extreme contention.

Wait-freedom is particularly relevant in systems where predictability matters, for example in a real-time context or when operating under a service-level agreement [KP11].
4.2 Wait-Free Synchronisation

The signal data structure that was introduced in Section 2.3.1 is a generic coordination primitive with easy-to-understand semantics. It is well suited for use with fork–join task graphs as shown in Figure 4.2: the spawning i-let creates a signal and initialises it with the number of child i-lets it generates; it then invokes the wait operation to block until its children have finished their work. Each child i-let calls the signal primitive after doing its job and before returning; the parent is woken up as soon as the last child has finished. Signals are not limited to fork–join scenarios but can also be used to coordinate more complex and less specific concurrency patterns.

At its core, a blocking signal in OctoPOS consists of a non-negative integral counter and a reference to a blocked context (via pointer or index). The two are mangled together into a 32- or 64-bit memory word. Concurrent modifications of the signal’s internal state are serialised with the help of the CAS instruction [Oec18] and an unbounded retry loop. As a result, the signal data structure and its operations are not wait-free, which degrades the predictability of concurrent applications. An additional source of jitter that prevents wait-freedom is the polling loop on the blocked context’s in-transition flag. These predictability considerations led to the design of two new types of signal in the context of this thesis: a fully wait-free binary signal and a unified counting signal most of whose operations are wait-free.

4.2.1 Wait-Free Binary Signal

Applications frequently expose coordination patterns where one activity needs to wait for only a single other activity. One example of such a pattern is the implementation of a future as used by OctoPOS’ RPC mechanism, where the calling i-let execution is blocked until the RPC has finished [Oec18]. A specialised binary signal that supports only the counter values 1 and 0 – that is, a kind of private binary semaphore – is sufficient for such cases where only two parties are involved.
In this section, a binary-signal implementation is proposed. All operations provided by the data structure are wait-free.

As depicted in Figure 4.3, a binary signal does not need to manage an explicit counter but can be represented as a finite-state machine with three distinct states: *initial*, *waiting* and *signalled* (or \( i \), \( w \) and \( s \) in short). In the *initial* state, the signal is ready for use. The *waiting* state denotes that an \( i \)-let is currently blocked on the signal. Once the signal has reached the accepting state *signalled*, it must be explicitly reinitialised before it can be used again. For the sake of simplicity, we assume for now without loss of generality that the *wait* and *signal* operations are mutually exclusive and are executed sequentially; ensuring correctness in the face of overlapping execution patterns will be dealt with later in this section. Since either of the two operations may be executed first, both possible sequences are considered in the state machine. If *wait* is invoked before *signal*, the calling \( i \)-let must go to sleep and *signal* must subsequently take care of waking it. Otherwise, if *signal* comes first, the automaton is immediately brought into the *signalled* state; the subsequent *wait* is a no-op because the resumption condition is already fulfilled and the calling \( i \)-let can continue without blocking.

In the C++ implementation that was built for OctoPOS in the context of this work, the binary signal’s internal state is represented by a pointer-sized memory word. The state values *INITIAL* and *SIGNALLED* are distinct integer constants that have no valid pointer equivalents. By contrast, the *waiting* state has no distinct numerical representation but is expressed as a valid pointer to a wakeup \( i \)-let. An excerpt of the source code, slightly simplified for easier readability, is shown in Listing 4.3. The code utilises the atomic swap instruction, which unconditionally exchanges the contents of a register and a memory operand. Since a binary signal has exactly two participating control flows – one calling *wait()* and one calling *signal()* – overlapping can only occur crosswise between these two operations.

Both *wait()* and *signal()* contain a slow path and a fast path; the latter contains no atomic operation and is taken whenever it can be ruled out that an overlapping execution might interfere with the own operation. This condition is met if the signal’s state indicates that the critical part of the respective opposite operation has already finished. Provided there are no overlapping executions, the first method to be called will take the slow path and its counterpart will subsequently take the fast path. Only if the executions of the two methods do overlap do both have to take the slow path to ensure a consistent outcome.

The *wait()* method uses a post-switch hook and implements the previously described four-step pattern of MAOS. It first checks whether the signal is already in the *SIGNALLED* state (line 7); if this is the case, it has no further business and returns immediately without blocking. Otherwise, it takes
4.2 Wait-Free Synchronisation

```cpp
void BinarySignal::init() {
    this->state = INITIAL;
}

void BinarySignal::wait() {
    // Fast path: if the state is already SIGNALLED, return immediately
    if (this->state == SIGNALLED) {
        return;
    }
    // Slow path: initialise the wakeup i-let, switch to a new context and
    // execute the post-switch hook
    ContextManager::getCurrentContext()->prepareToBlock();
    ContextManager::Inst().next([](Context *ctx, BinarySignal *signal) {
        // Atomically store the wakeup i-let in the signal; if signal() has
        // been called in the meantime, set the state back to SIGNALLED and
        // immediately restore the blocked context
        if (Atomic::swap(&signal->state, ctx->getWakeupILet()) == SIGNALLED) {
            signal->state = SIGNALLED;
            ContextManager::Inst().resume(ctx);
        }
    }, this);
}

void BinarySignal::signal() {
    auto oldState = this->state;
    if (oldState != INITIAL) {
        // Fast path: someone is already waiting; set the state to SIGNALLED
        this->state = SIGNALLED;
    } else {
        // Slow path: atomically set the state to SIGNALLED; return unless wait()
        // has been called in the meantime
        oldState = Atomic::swap(&this->state, SIGNALLED);
        if (oldState == INITIAL) {
            return;
        }
    }
    // oldState is a valid pointer to a wakeup i-let; submit it to scheduling
    InvasiveScheduler::Inst().ready(oldState);
}
```

Listing 4.3: C++ implementation of the binary-signal data structure.

The source code is slightly simplified for better readability by manually inlining methods and by leaving out assertions, casts and namespace prefixes. The post-switch hook is a C++11 lambda expression.

the slow path by setting up a wakeup i-let and switching to a new context with a post-switch hook. In the example, the hook is a non-capturing C++11 lambda expression [JF10] – an anonymous function whose address is passed to next() as a function-pointer parameter. Immediately after the context switch, the hook function atomically marks the signal as waiting by storing a pointer to the wakeup i-let in it, and examines the signal’s previous state (line 17). In the default case, the signal

\[3\] Since the wakeup i-let is stored preinitialised inside the context, said set-up procedure consists of nothing more than copying the current i-let’s control word.
did not get modified by an overlapping call to `signal()` in the meantime – so the old context is now in a blocked state, the hook function returns and the new context starts in the dispatch loop. Otherwise, the signal has concurrently been brought into the `SIGNALLED` state and the calling i-let must not be blocked after all: doing so would trigger the classic lost-wakeup problem. We could either set the wakeup i-let ready or we could immediately switch back to the previous context and resume its execution. We use the latter method because this way the processor can take advantage of the still-warm caches. To bring the signal into a consistent state, its value is explicitly set back to `SIGNALLED` – an action that is not strictly necessary but makes inspection and debugging simpler.

The `signal()` method takes the fast path if the signal is already in the `waiting` state (line 28): it changes the state to `SIGNALLED` and sets the wakeup i-let ready. Again, explicitly setting the state word is not essential, but useful nonetheless to maintain consistency. In the slow path, an atomic swap instruction is executed to set the state to `SIGNALLED` and to examine the previous state: if it is still `INITIAL`, nobody is waiting yet and there is nothing else to do. Otherwise, an overlapping `wait()` has completed blocking and needs to be woken up again, so the wakeup i-let is submitted to scheduling.

In case of overlapping executions, the two crosswise atomic swaps mark the watershed that determines which of the two control flows is responsible for unblocking the context: namely the control flow that came second. The delayed placement of the swap inside the post-switch hook function guarantees that the context save is complete before the watershed is reached, obviating the need for an in-transition flag and a corresponding polling loop. Consequently, the binary signal is a wait-free data structure with predictable timing behaviour.

### 4.2.2 Counting Signal

We can build upon the wait-free binary signal and combine it with an atomic counter to yield an alternative implementation of a generic counting signal. The interface and functional semantics of the resulting data structure are identical to those of the original `SimpleSignal` class. The main difference is that the new implementation is mostly wait-free, with two notable caveats:

1. The target ISA must provide an atomic fetch-and-add/fetch-and-subtract instruction, which is the case on x86_64, i686 and ARMv8.1+, for example. If such an instruction is not available – as is the case on SPARC LEON3, PowerPC or ARMv8 and below – it needs to be emulated with the help of a CAS or LL/SC loop, which defeats wait-freedom.

2. The number of child i-lets to wait for must be known at the time the signal is initialised. This is because, as will be shown, the `addSignallers()` method, which increments the internal counter, is not wait-free.

Listing 4.4 displays the source code of the new counting signal. The methods `init()`, `wait()` and `signal()` are fairly straightforward. The `init()` method sets the counter to the given value and initialises the binary signal, with its inceptive state depending on the counter: if there are i-lets to wait for, then it is set to `INITIAL` as usual; otherwise the state is set to `SIGNALLED` to indicate that no actual waiting needs to be performed. The `wait()` operation is a trivial wrapper around the corresponding `BinarySignal` method. In `signal()`, the counter is decremented atomically; if it reaches zero, the underlying binary signal is signalled. The atomic decrement ensures in the face of concurrent calls to `signal()` that the counter value remains consistent and that only one control flow ultimately gets to notify the binary signal.

The crux lies in `addSignallers()`, which can be invoked to tell the signal about the dynamic creation and activation of more child i-lets to wait for. Sans the loop marked with the cyan box,
4.2 Wait-Free Synchronisation

Listing 4.4: C++ implementation of the counting-signal data structure.

Assuming hardware support for atomic read-modify-write instructions, all methods except addSignallers() are wait-free.

which is explained later, the method increments the counter by the given value, then makes sure that the binary signal – which may already be SIGNALLED if the counter was previously zero – is back in the INITIAL state.

The data structure has to maintain the following invariant: while no operation on the signal is currently running, the counter is either greater than zero and the binary signal is not SIGNALLED, or the counter is zero and the binary signal is SIGNALLED. Two particular uses of addSignallers() are considered programming errors because they lead to violations of that invariant:

1. A call to addSignallers() that marks the creation of a new i-let must not happen after, or concurrently to, the corresponding signal() invocation from that very i-let – otherwise, the signal counter may temporarily reach a negative value, which is an illegal state. This is easily ensured by incrementing the counter before spawning the activity. It is still perfectly legal to invoke addSignallers() concurrently to signal() calls from other, previously spawned activities.

2. A call to addSignallers() that happens after, or concurrently to, wait() is also a programming error. In other words, all spawn operations that create additional concurrent activities to
4 Revisiting the Invasive Execution Model and Its Implementation

Table 4.1: Lost-update problem caused by the overlapping execution of `signal()` and `addSignallers()`.

<table>
<thead>
<tr>
<th>Line</th>
<th>Signal state</th>
<th>Line</th>
<th>Signal state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>counter binarySignal</td>
<td></td>
<td>counter binarySignal</td>
</tr>
<tr>
<td>10</td>
<td>1 INITIAL</td>
<td>10</td>
<td>1 INITIAL</td>
</tr>
<tr>
<td>11</td>
<td>0 INITIAL</td>
<td>11</td>
<td>0 INITIAL</td>
</tr>
<tr>
<td>17</td>
<td>0 INITIAL</td>
<td>17</td>
<td>0 INITIAL</td>
</tr>
<tr>
<td>21</td>
<td>1 INITIAL</td>
<td>21</td>
<td>1 INITIAL</td>
</tr>
<tr>
<td>27</td>
<td>1 INITIAL</td>
<td>27</td>
<td>1 INITIAL</td>
</tr>
<tr>
<td>12</td>
<td>1 SIGNALLED</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a) Problematic execution sequence.

(b) Corrected execution sequence.

Each row indicates the counting signal’s internal state after the execution of the respective source-code line. The two concurrent control flows are highlighted in different background shades: `signal()` in light grey, `addSignallers()` in dark grey. Correct intermediate states are marked in green, incorrect ones in red.

wait for must happen strictly before the call to `wait()`. This is because at the time the `wait()` method gets invoked, the signal’s current counter value dictates how many signalisations are to be waited for.

Both points can lead to obvious invalid behaviour if they are not respected – not only for our concrete counting signal but for any use of a counting semaphore in a fork-join scenario. Furthermore, it is assumed for every counting-signal instance that only a single control flow spawns additional i-lets and calls `addSignallers()` at a time. Otherwise, extra synchronisation measures are required to ensure the sequentiality of `addSignallers()` invocations.

With these restrictions in mind, the only remaining race condition that can occur under legal conditions is between `addSignallers()` and `signal(); wait()` can be left out of the equation. When we still ignore the framed loop in lines 23–25 for now, the invariant ends up being violated by the execution sequence given in Table 4.1a if the counter equals 1 at the outset and the binary signal is in the INITIAL state. Since i-lets are scheduled non-preemptively, the overlapping pattern can only be caused by parallel execution on different CPU cores, not through preemption. Without loss of generality, it is sufficient to consider only one execution of `addSignallers()` and one of `signal();` even with more concurrent participants, the serialising semantics of the atomic read–modify–write instructions would ensure that the conditions in line 11 and line 21 evaluate to true for exactly one participant each. The issue itself is a typical lost-update problem: if `signal()` gets overtaken halfway by `addSignallers()`, it subsequently overwrites the state change made to the data structure, leaving it in a SIGNALLED state even though the counter indicates that there is still another i-let to wait for. This race condition is eliminated with the help of the loop in lines 23–25, which delays the further execution of `addSignallers()` until the concurrent `signal()` operation has finished. The resulting execution sequence in Table 4.1b ultimately brings the signal back into a consistent state.

Thanks to the non-preemptive nature of i-let scheduling, `signal()` is expected to run to completion. Provided detailed knowledge of the underlying hardware is available, it is thus possible in theory to give an upper bound on the number of loop iterations needed until the store done in line 12 becomes visible and `addSignallers()` can finish.

A race condition in the opposite direction – `addSignallers()` getting overtaken by a concurrent `signal()`, leaving the binary signal INITIAL despite a counter value of zero – would only be possible if the counter were initially zero, which would constitute a programming error according to condition 1.
4.2 Wait-Free Synchronisation

In summary, given appropriate hardware support, the new counting signal permits wait-free synchronisation for fork-join scenarios where the number of children is known at the outset. Unlike the original SimpleSignal, which mangles the counter and a context identifier together and updates them via a (single- or double-word) CAS, it manages its counter and the wakeup-i-let pointer separately. With this, the number of atomic instructions executed is higher, but the instructions themselves are less complex and more predictable in principle than CAS.

4.2.3 Unified Blocking and Spawning Signal

In OctoPOS, the blocking signal and the spawning signal are distinct data structures [Oec18]. The new binary and counting signals presented above open up the possibility to build unified signals that can be used in either blocking or spawning mode, depending on what suits the application’s needs the best under a concrete circumstance. In this way, the semantics of the unified binary signal are similar to those of the future exposed by the asynchronous system-call interface of OctoPOS-RI [Sch18]. The decision which mode to use does not necessarily have to be made in advance by the programmer but can be made by the application at run time. The program calls either wait() in order to block or onCompletion() to have the local claim infected with a given i-let once the counter reaches zero.

The implementation of the onCompletion() method is shown in Listing 4.5. Its general principle is analogous to that of wait() – but it manages a completion i-let, which it takes as argument, as opposed to generating a wakeup i-let and blocking by switching contexts. The method is designed for being possibly called concurrently to signal(). In the counting signal, onCompletion() wraps the corresponding method of the binary signal used internally, just like wait() does. In the binary signal, the method contains a fast path and a slow path; the former, which is taken if the state is already SIGNALLED, sets the completion i-let ready at once. On the slow path, the completion i-let is atomically stored into the state field and the previous state is examined: if it has changed due to a concurrent signal(), the state is set back to SIGNALLED for consistency reasons and the completion i-let is set ready. Otherwise, it is signal() that will ultimately take care of submitting the completion i-let to scheduling.

```cpp
void BinarySignal::onCompletion(iLet *iLet) {
    if (this->state != SIGNALLED) {
        // Slow path: atomically store the i-let in the signal; return unless
        // signal() has been called in the meantime
        if (Atomic::swap(&this->state, iLet) != SIGNALLED) {
            return;
        }
        this->state = SIGNALLED;
    }
    InvasiveScheduler::Inst().ready(iLet);
}

void CountingSignal::onCompletion(iLet *iLet) {
    this->binarySignal.onCompletion(iLet);
}
```

Listing 4.5: C++ implementation of the onCompletion() method of binary and counting signals.

The method sets a given i-let ready as soon as the signal data structure is signalled.
It is a programming error to call both `wait()` and `onCompletion()` on the same signal – but the unified data structure makes it possible to dynamically decide at run time which of the two primitives to use.

### 4.3 Handoff Scheduling

In many situations where the signal data structure is used, such as in a fork–join scenario, the `signal` primitive is routinely the very last operation to be carried out by its calling i-let. Provided the parent is already waiting, the child control flow that performs the last call to `signal`, thus decrementing the counter to zero, bears the responsibility for setting the wakeup i-let ready by submitting it to the scheduler. As an optional optimisation, the scheduler can be circumvented entirely and an immediate switch to the blocked context can be executed instead. This technique of directly passing control from one cooperating control flow to another is known under the name *handoff scheduling* [Bla90]. It is also related to Hoare's *monitor* concept [Hoa74], where the `signal` operation causes exactly one waiting participant to be resumed immediately. The direct context switch eliminates the scheduling and dispatching overheads, and obviates the need to execute an explicit wakeup i-let. As an indirect effect, data items produced by the child i-let are still warmer in the cache, which has the potential to speed up their consumption by the parent.

The primary use case for handoff scheduling as implemented in Mach or in the Taos operating system [Ber+90] is the acceleration of RPCs from a user-application process to a server process; the requesting thread is suspended until the call is completed and an address-space switch to the server's address space is performed. Likewise, the `signal` operation of a Hoare-style monitor temporarily relinquishes control; its direct handoff to a waiting control flow ensures that the latter can enter the just-released monitor without the risk of a third party being faster and entering the monitor instead. By contrast, handoff scheduling in OctoPOS is intended to optimise a one-way control transfer from one i-let to another that belongs to the same address space: passing control implicitly terminates the calling i-let.

Handoff scheduling is established in OctoPOS through the introduction of a new signal operation called `signal and exit`. This operation modifies the state of the signal data structure in the same way the regular `signal` operation does, but it ultimately does not return to its caller\(^5\). Instead, it attempts to perform handoff scheduling to the wakeup or completion i-let. If the signal is holding a completion i-let that was set by a call to `onCompletion()`, the stack is unwound and control is passed directly to that i-let. Otherwise, if the signal is referring to a resumable context through a wakeup i-let, `signal and exit` extracts that context from the i-let and switches to it. To differentiate between these two cases, it suffices to compare the i-let's function pointer against the address of the aforementioned `unBlockProxy()` method – a technique related to continuation detection [Dra+91]. As an additional constraint, since an invasive computing system is space-partitioned into elementary claims, handoff scheduling is only legal if the i-let's intended destination claim equals the caller’s claim – otherwise, the i-let would be executed on the wrong claim. If the claims do not match, one must fall back to the unoptimised code path that submits the i-let to regular scheduling. If the call to `signal and exit` does not trigger a handoff (because the signal counter is non-zero, no i-let is set yet or the claims do not match), it exits the current i-let execution by unwinding the stack and jumping back into the dispatch loop.

\(^5\)As a consequence, the programmer must keep a close eye on the lifetimes of their resources, in particular when using the *resource acquisition is initialisation (RAII)* idiom [Str13].
Figure 4.4: Fork–join scenario using the *signal and exit* primitive.

The principle is identical to Figure 4.2, but signalisation and returning from the i-let have been coalesced into a single, optimised operation. Being the last control flow to call *signal and exit*, the i-let on the right is the one that does a handoff back to the parent i-let.

Figure 4.4 shows what the fork–join scenario from Figure 4.2 looks like when the *signal and exit* primitive is used. By circumventing the scheduler, handoff scheduling violates the system’s scheduling policy and flatly awards maximum priority to the resumed i-let. The application programmer is free to ponder on a case-by-case basis whether this is acceptable, and to use the *signal and exit* primitive only in code paths where overriding the scheduling policy is deemed reasonable. For real-world scenarios, it is expected that developers are primarily interested in throughput rather than in the enforcement of a particular scheduling scheme, and that they consequently make frequent use of *signal and exit*.

### 4.4 Summary

The improvements presented in this chapter aim at making synchronisation in invasive applications both more efficient and more predictable. The management of execution contexts was overhauled for better locality and faster accesses, and stack switches were made more efficient. Building upon MAOS’ enqueue-handler concept, the technique of using post-switch hook functions led to the development of a specialised binary-signal data structure, all of whose operations are wait-free. Moreover, a mostly wait-free counting signal was proposed and the blocking and spawning signal semantics were unified. Finally, handoff scheduling was implemented to bring down wakeup latencies.

All these enhancements and optimisations were originally developed and implemented for OctoPOS and have been in use there for several years. They are also included in InvRT, the invasive runtime system built on top of AtroPOS. Their efficacy and their performance impact are analysed in Section 7.3.2.
I conclude that there are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies.

Charles Antony Richard Hoare, ‘The Emperor’s Old Clothes’ [Hoa81]

The Design of AtroPOS and InvRT

The selective improvements presented in the previous chapter serve to enhance the efficiency and predictability of OctoPOS. In this chapter, we will go deeper and describe a far-reaching redesign of the OctoPOS kernel that culminates in a new operating-system design called AtroPOS and a complementary runtime system called InvRT.

The primary factor motivating this redesign is the need to introduce vertical isolation: the state of the operating-system kernel must be protected against being compromised, either accidentally or maliciously, by user applications. Moreover, confidential data held by the kernel, such as cryptographic keys or I/O-buffer contents, must be prevented from leaking to user applications not authorised to see them. The missing building block that is required for enforcing the OctoPOS security model (see Section 2.3.3) is vertical isolation. Without it, the already existing horizontal isolation is effectively worthless as a security mechanism because there is no way yet to bar an attacker from simply manipulating the operating system’s data structures that configure memory protection. Consequently, only through the consistent and gap-less application of both horizontal and vertical isolation is it possible to give any security guarantees for the overall system.

The challenge is to introduce vertical isolation in a way that does not severely hamper the system’s efficiency. As OctoPOS deals with vast quantities of fine-grained application control flows in the form of i-lets, drawing the fence around its existing programming interface would render the handling of those control flows far more expensive: every operation related to scheduling, dispatching or synchronisation would become a costly system call. Thus, the demarcation line must be drawn at a lower level, straight through the existing operating-system architecture. The new system-software stack resulting from that redesign consists of two entities: a minimal (atrophied) operating-system kernel, called AtroPOS, that exposes a low-level interface, and on top of it a user-space runtime system that implements the invasive execution model, named InvRT.

As a second aspect, the redesign opens up an opportunity to shape the new kernel interface in such a way that AtroPOS is able to serve as a generic foundation for arbitrary concurrency platforms. The goal is to offer elementary low-level functionality upon which runtime systems other than InvRT can be built, so that application programmers are free to pick the technology suited best for writing their concurrent programs, be they invasive or non-invasive.
5 The Design of ArroPOS and InvRT

5.1 Design Principles

Ensuing from these considerations, the ArroPOS kernel is designed with the following underlying guiding principles in mind.

Maintain support for the invasive operating-system concepts. The well-proven fundamental concepts of invasive computing should be preserved and the operating system should be structured around them. These concepts include the operating system’s overall configurability and tailorability, the spatial partitioning of system resources in the form of claims, and the multikernel approach that sees the operating system as a collection of individual kernel instances that are coupled together. At the same time, the operating system should remain application-oriented: design decisions that directly affect the user application should be left to the latter (or to its runtime system, for that matter) as far as feasible.

Be generic and let user space have its way. The application itself knows best how to structure its control flows in order to efficiently execute concurrent code – be it as i-lets, Cilk tasks, X10 activities, fibres, or as something else entirely. The operating system should not prescribe a particular higher-level abstraction, such as threads, that offers unneeded features, often behaves in unexpected or undesired ways, and forces the application’s runtime system to work around its idiosyncrasies. Instead, the operating system should expose the system’s CPU cores to the application as is, and leave the rest up to the runtime system. In particular, scheduling decisions should be made in user space. Moreover, multi-model parallel programming [SLM90] should be made easy: not only should different user programs with different runtime systems be able to coexist and run side by side, but that type of heterogeneity should also be possible within applications.

Do not get in the way. Like OctoPOS, the operating system should strive to maintain a mostly interrupt-free mode of operation to shield applications from jitter. Performance-critical functionality should reside in user space so as not to necessitate excessively frequent context switches. While it should be possible to fully protect the kernel from user space, costly protection techniques such as vertical and horizontal isolation should be configurable both statically and dynamically at the request of the applications involved.

5.2 Overall Architecture

Based on the principles given above, a radical redesign of the original OctoPOS kernel is carried out by cutting it in two, yielding the ArroPOS kernel and the InvRT runtime system. The newly introduced fault line between these two entities is shown in Figure 5.1.

The operating-system kernel has a mostly monolithic architecture, but it draws inspiration from micro- and exokernel designs [Dra+91; EKO95] in certain respects such as the kernel execution model. In similar fashion as PANDA’s pico-kernel approach [Ass+93b], ArroPOS strives to keep the bulk of the code out of the kernel and in an application-specific runtime system, in a bid to effect a low frequency of system calls. Still, all code related to the control of hardware devices – including drivers, the hardware abstraction layer and the management and communication facilities built on top of it – remains in the kernel. Resource handling also mostly stays part of the kernel, whereas the implementation of the execution model is moved into the InvRT runtime library.
Starting from the layered structure shown in Figure 2.5 on page 22, the programming interface and most of the functionality in terms of scheduling, dispatching and synchronisation are moved into the runtime system. The lower-level resource management and the other subsystems remain part of the kernel.

In other words, the low-level invade, rein invade and retreat operations are exposed by the kernel interface; by contrast, everything related to i-lets, tile-local infects, synchronisation constructs and the management of execution contexts are handled entirely by InvRT. Infecting a remote tile with an i-let or initiating a push data transfer involves interaction between the runtime system and the kernel. Generally speaking, the abstractions provided by AtroPOS are either at the same level as the invasive primitives or – the majority of them – at a lower level. Functionality that is not security-relevant is moved out of the kernel and into the invasive runtime system if one of the two following conditions applies:

- It constitutes a hot code path that is executed frequently and is thus performance-critical. The best example of such functionality is the scheduling and dispatching of i-lets.
- Its interface is too specific to serve as a foundation for ports of other, non-invasive runtime systems. One instance is the inter-tile communication mechanism, which is deeply interwoven with the i-let subsystem in OctoPOS.

In both cases, the pertaining abstraction is relocated into InvRT and implemented there on top of a lower-level, more generic interface offered by the AtroPOS kernel. An important design goal herein is that, as in Psyche [Mar+91], the number and extent of assumptions the kernel needs to make about the structure, layout and behaviour of user-space entities be kept minimal – optimally even smaller than the assumptions made by Psyche. The contract between the two tiers of the system that defines their interface and interplay is to remain limited in terms of scope and complexity.

**AtroPOS** in its fundamental structure remains hardware-centric: physical CPU cores are exposed to the application as is, not encapsulated into some kind of higher-level abstraction of concurrency such as threads. As in OctoPOS, NUMA domains are logically represented as separate tiles with disjoint local memory regions, and every tile runs its private instance of the AtroPOS kernel. The operating-system instances communicate with one another via RPC. Likewise, applications that span multiple tiles have distributed semantics; their parts communicate with one another with the help of messages. At the same time, AtroPOS does not require the availability of any of the special invasive hardware components introduced in Section 2.3.2, including the CiC for the offloading of i-let scheduling and the special i-let primitives supported by the NoC adapter – but it can make efficient use of these components if they are present. The primary target platform of AtroPOS
is the class of large x86_64-based COTS machines with a NUMA architecture. The SPARC-based invasive hardware platform is equally supported in principle, but the prototypical implementation of AtroPOS was carried out for x86_64 for a number of reasons, among them greater hardware stability and better comparability with existing systems, especially with Linux.

The kernel’s basic abstraction of a set of computing resources belonging together is the elementary claim as introduced in Section 2.3: a bundle of homogeneous CPU cores (with associated memory) made available to a particular application, all of the same type and residing on the same tile. Higher-level entities in the claim hierarchy, such as heterogeneous claims or claims that span multiple tiles, are handled in user space by the invasive runtime system. By default, elementary claims provide performance isolation through strict resource exclusivity because AtroPOS enforces spatial multiplexing: every CPU core is assigned to at most one claim at a time. Optionally, through claim virtualisation, computing resources may be shared among multiple claims and may be multiplexed temporally. With claim virtualisation enabled, the properties of an elementary claim resemble those of an MCP in ROS [Klu+10] and Akaros [Rho14]: a claim represents a kind of virtual multiprocessor; its cores are gang-scheduled by the kernel in long time slices and are insulated from unnecessary interrupts. Note that claim virtualisation is not a central contribution of this dissertation and will only be covered briefly insofar as it is relevant for this work, not in full exhaustive detail.

Like OctoPOS, AtroPOS is a configurable family of operating systems that can be tailored at compile time. The existing configuration and build infrastructure is reused, and certain new optional features such as vertical isolation are added, which is described in detail in the next section.

The clear separation between kernel and user space that is introduced with AtroPOS has far-reaching consequences on both sides for the way control flows are represented and handled. While OctoPOS makes no distinction between application control flows and kernel activities, and the respective execution contexts associated with them, AtroPOS draws a strict line between the user level and the kernel level. User control flows are clearly delineated from kernel control flows, and the two are decoupled from each other. The operating-system kernel no longer supports user i-lets as first-class citizens, and its low-level interface has no notion of i-lets. Instead, it exposes virtual CPU cores to the application; all i-let functionality is located in the InvRT runtime system. Kernel-internal activities are expressed in the form of kernel jobs. The control-flow model and its implications are discussed in Section 5.4.

Communication between elementary claims that reside on the same tile is achieved through shared memory and does not warrant any kernel intervention; it is carried out entirely in user space by InvRT. The kernel provides low-level facilities for inter-tile communication which ensure that only claims belonging to the same application can exchange messages with one another. These mechanisms are presented in Section 5.5.

Finally, the design of the new InvRT runtime system and its interplay with AtroPOS are presented in Section 5.6, and efforts to port other existing concurrency platforms to the AtroPOS interface are reviewed in Section 5.7.

5.3 Configurable Privilege Isolation

Vertical isolation – the logical separation of a privileged kernel space from an unprivileged user space – always comes at a cost to user programs. The overhead entailed by the separation is twofold: direct costs are caused by the extra work needed to switch privilege levels and to transfer control from the user program to the kernel and back [And+91; Lie93]; among the indirect costs
5.3 Configurable Privilege Isolation

is the pollution of caches, TLBs and branch predictors by that extra code [Lie95]. Although CPU architects have expended effort into reducing both cost categories through the introduction of special system-call and -return instructions, the overheads are still far from negligible and constitute a source of jitter.

Existing operating systems are commonly either constructed with isolation as an integral part, or they offer no isolation at all. OctoPOS features dynamically configurable isolation in the horizontal dimension but lacks a concrete implementation of the vertical dimension. AtroPOS, presented here, adopts the principle of dynamic memory protection from OctoPOS and additionally implements a mechanism for privilege isolation that is optional and configurable both statically at compile time and dynamically at run time. Applications are given the choice and opportunity to balance their need for security against their other requirements such as efficiency or predictability, and the operating system is tailorble to those demands. The needs of applications may shift during their execution – for example, maximal performance may be paramount during certain phases and subordinate in other phases of the same program run. The system is able to dynamically react to such changes and to adapt itself in situ at run time [Dre+16].

Although vertical and horizontal isolation are often rightfully seen as two sides of the same coin, we regard them as two distinct concepts that are applied orthogonally. While memory protection alone cannot thwart a malicious attacker, it is still able to confine the repercussions of programming errors or transient bit flips. Privilege isolation, on the other hand, cannot on its own prevent a mischievous user program from wreaking havoc in arbitrary unisolated memory regions – but it protects the kernel itself from being taken over. AtroPOS accommodates this circumstance by allowing the two dimensions of isolation to be enabled and disabled independently of each other. A number of possible example scenarios are depicted in Figure 5.2. They are similar in principle to the scenarios discussed by the authors of Singularity [Aik+06; HL07], but the major feature added by OctoPOS and AtroPOS is the system’s ability to reconfigure the protection at run time.

This dissertation does not answer the question of how an application’s trustworthiness can best be asserted and attested. While code signing can help to ascertain the origin of a piece of software, a cryptographic signature cannot provide any meaningful guarantees as to the actual behaviour of that software. Even if the application’s source code is type-safe and memory-safe, we additionally have to trust its runtime system and the language virtual machine or ahead-of-time compiler. Furthermore, even an application that is trustworthy per se in the sense that it never violates memory-safety constraints may still behave maliciously by carrying out side-channel attacks. Note, however,
that memory protection and privilege isolation – regardless of whether they are implemented in
hardware or in software – are insufficient for defending against side channels anyway.

As is the case for memory protection, when deciding whether to enable or disable privilege isolation
for a particular application, one must take into account the demands of all participants involved.
For instance, if an application has strict security requirements because it handles confidential data,
the operating system must enforce these requirements by disallowing all others from running in
privileged mode [Dre+16]. The focus of this thesis, however, lies not primarily on the strategies
but on the corresponding enforcement mechanisms included in the operating system.

5.3.1 Building Blocks and Hardware Requirements

Configurable vertical isolation in AtroPOS is composed of three fundamental building blocks:

1. Two separate modes of execution – a privileged and an unprivileged one.
2. Protection of privileged memory regions against accesses from unprivileged mode.
3. System calls and upcalls to switch forth and back between unprivileged and privileged mode
   in a controlled manner.

These building blocks are briefly explained in the following paragraphs.

Execution modes Like most contemporary operating systems, AtroPOS regards privilege isolation
as a binary matter. That is, it distinguishes between two distinct modes of operation: privileged mode
(also called supervisor mode or kernel mode) and unprivileged mode (also known as user mode). Since
the kernel is very small, it does not make use of hierarchical, multi-layered protection rings [SS72]
as implemented, for instance, in Multics [Sal74]. The kernel runs in privileged mode at all times
and has full control over the entire system, including devices, privileged instructions and all memory
ranges. The processor mode in which user applications are executed is variable: isolated user code
runs in unprivileged mode, unisolated code in privileged mode.

Kernel-memory protection AtroPOS prevents all memory regions that belong to the kernel from
being read, written or executed by user-mode code. The critical component whose integrity and
confidentiality are essential is the kernel’s internal state, including its data, I/O buffers and memory-
mapped device registers. For the kernel’s program code, by contrast, it would generally be sufficient
to protect it against being manipulated through writes from user space. However, as recommended
by current security guidelines, AtroPOS restricts all types of access to the kernel’s text section, too,
in order to reduce the system’s attack surface. Doing so induces zero extra overhead but prevents
the operating system from supplying a vast complimentary repository of gadgets for conceivable
return-oriented-programming [Sha07; HHF09] exploits carried out in user space.

Mode changes Switching deliberately from user mode into kernel mode – a system call – is a
synchronous operation that traps into the kernel. Directing control in the opposite direction is
either a return from a preceding trap or an upcall into a predefined program entry point. For
security reasons, entries and exits are handled asymmetrically: control transfers from privileged
to unprivileged code are free to jump into arbitrary user code, whereas control transfers in the
opposite direction must end up at a specific kernel entry point that decides where to go next –
otherwise, it would be trivial for user programs to trigger the execution of arbitrary kernel code.

82
5.3 Configurable Privilege Isolation

Configurable vertical isolation does not pose any special hardware requirements, but it can make use of dedicated accelerating instructions if available. It builds upon the established mechanisms used by commodity operating systems that implement a fixed privilege-isolation model: processor support for (at least) two different execution modes, a means to mark certain memory areas as not accessible from user mode, and a synchronous mode-switching mechanism. Intel and AMD x86_64 CPUs offer these features in the form of hardware protection rings, MMU support for paged memory including a per-page supervisor bit, and special system-call and -exit instructions for entering and leaving supervisor mode in a controlled manner. On platforms where no such special instructions exist for mode switching, for example SPARC LEON3, a software trap can be used instead.

From a conceptual point of view, the sole difference to conventional, static privilege isolation is the fact that user code can now be executed in either unprivileged or privileged mode, based on the decisions made by the operating system. Hence, not every call into the kernel (or interrupt, for that matter) necessarily entails a processor-mode switch: unisolated user code may perform direct function calls instead. The details of how system calls are handled in the face of configurable vertical isolation are described in Section 6.3.

5.3.2 Static and Dynamic Configurability

Privilege isolation is added to the existing configuration infrastructure of AtroPOS as an optional feature. The systems integrator in charge of assorting a concrete configuration of the operating system is given a ternary choice: statically enabled, statically disabled or dynamic. If the first option value is selected, the behaviour of commodity operating systems is mimicked: applications unconditionally run isolated in user mode. The second choice causes AtroPOS to dispense with vertical isolation altogether and to execute all applications in supervisor mode. The third option, dynamic, allows the system to freely mix isolated and unisolated application code and to make all pertaining decisions dynamically at run time. All statements and descriptions made in the remainder of this section refer to that latter variant of the operating system.

Dynamic vertical isolation is managed on three separate levels of abstraction with different granularities and different temporal characteristics: application, elementary claim and core. With this, it reflects the three fundamental abstractions provided by AtroPOS:

- **Application level.** Every application binary is tagged with a static Boolean flag that specifies its initial privilege setting. The flag has global semantics and is not mutable over the lifetime of the application. When the operating system launches the application, it reads that flag from its application descriptor and then sets up the execution of the program’s entry code either in unprivileged or in privileged mode, based on the value of the flag.

- **Elementary-claim level.** To control vertical isolation on the claim level, every elementary claim holds a flag of its own in its descriptor. The privilege status can be changed dynamically at run time and affects all CPU cores associated with the elementary claim. An application’s initial claim inherits the flag from the application descriptor. By default, additional elementary claims that are subsequently staked via the *in invade* operation implicitly inherit the flag from the elementary claim that performs the *in invade* unless a privilege level is explicitly specified for the new elementary claim.

- **Core level.** Each core inherits the privilege level from the elementary claim it belongs to, but the core’s privilege level can also be adjusted at run time. This allows for short-term and fine-grained dynamic changes even for short sections of application code.
Changing a core’s own privilege level at run time is an inexpensive operation. Dynamically setting an elementary claim’s privilege status is costlier because it pertains to all cores belonging to the claim; hence, it offers itself to set the desired status already during invasion. As hierarchical claims are high-level entities that are managed not by ArroPOS but by the invasive runtime system, managing the privilege levels of a higher-level claim is a task for the runtime system: it needs to enumerate the corresponding elementary claims – including those residing on remote tiles – and set the correct level for each elementary claim individually. Likewise, no direct mechanism is offered to alter the privilege level of an entire application during its execution because such an event is assumed to be exceedingly rare; a program that wants to achieve this needs to iterate over all its elementary claims and adjust their privileges.

With this three-tier approach, it is possible to configure privilege isolation individually per application, both in advance and at run time, and in both a coarse-grained and a fine-grained manner. This is illustrated with the help of the following examples.

5.3.3 Example Scenarios

These five scenarios describe a number of exemplary use cases for the configurability aspect of privilege isolation in ArroPOS, in both its static and dynamic manifestations. While X10 and C++ are mentioned explicitly in the examples, both serve as placeholders for any type-safe or type-unsafe programming language, respectively. Note that, for reasons of simplicity, we focus here on cases where only a single application is present. As mentioned previously, every decision whether to grant or deny a concrete request for heightened privileges depends on the system’s overall state and on the combination of the demands expressed by all participating applications.

**Untrusted C++ application** In the simplest scenario, the application is written in a non-memory-safe language such as C++. This is assumed to be the default case. Since the program cannot be trusted not to corrupt or manipulate the operating-system state, or to exfiltrate sensitive kernel data, a barrier needs to be erected that protects the kernel’s memory against accesses from user space: the application is flagged to run in unprivileged user mode. All claims inherit that flag upon their creation; effectively, all cores belonging to the application execute its code in user mode.

**Trusted X10 application** It has been shown that applications written in a type-safe language such as Java or X10 may run unprotected while still guaranteeing both memory safety and high execution performance – either through just-in-time compilation [PVC01] or through ahead-of-time compilation [Erh+11; Erh+14]. Hence, for a memory-safe program that is certifiably trustworthy, the hardware-enforced protection barrier between user space and kernel space is not required because vertical separation is already constituted through software mechanisms. The application may be marked as privileged; its claims adopt that status, and consequently all cores run the application code in supervisor mode.

---

The sources cited refer to Java applications in the field of embedded systems, but their fundamental assertion that static application and system knowledge can be exploited by an ahead-of-time compiler is transferable to the context of invasive computing.
5.3 Configurable Privilege Isolation

Hybrid X10/C++ application Our application is trusted and mostly written in type-safe X10 but contains performance-critical functionality that is implemented as a C++ module. For the sake of argument, it is presumed that the latter code is compute-intensive and does not issue system calls in abundance. The program stakes a dedicated claim on which to run the unsafe code portion. The application itself is marked as privileged and its claims inherit that property by default, but the particular unsafe claim is made unprivileged. Consequently, the X10 code is executed without extra protection, whereas the C++ code is forcefully hardware-isolated from the operating-system kernel. All guarantees pertaining to the system’s overall integrity and confidentiality are preserved.

Trusted X10 application using an untrusted C++ library An X10 application that is trustworthy per se but that performs calls into type-unsafe library code effectively has a varying trust level over time. The program is launched in privileged mode; the runtime system – which is also assumed to be trustworthy – then drops the local CPU core’s privileges before every call to a type-unsafe function and reacquires them after returning.

Application with special requirements The final example is a program written in C++ that contains particular code paths which perform system calls and are especially sensitive to their direct and indirect costs. The system software may decide to permit a compromise between the application’s requirements and the security needs of others, and to relax the restrictions it imposes. The application and all its claims are unprivileged, but whenever a CPU core enters a sensitive code path, it requests getting promoted to supervisor mode. As soon as it leaves that path, it drops the privileges again.

In the first two examples, it is sufficient to configure vertical isolation statically and on the application level. The third scenario makes use of the per-claim setting. In the fourth and fifth example, the management of privilege isolation is highly dynamic and happens at the granularity of individual CPU cores. With this scheme, the isolation status can be controlled and adapted in a very fine-grained manner. Special care must be taken if there is a possibility that a user control flow with changed privileges might migrate to another core, for example through blocking or yielding: dealing with or preventing such situations is left to the runtime system. While the above example scenarios are deliberately kept straightforward and simple, real-world applications can contain arbitrary combinations of these elementary patterns.

5.3.4 Summary

The interplay of hardware-enforced horizontal and vertical isolation ensures the integrity and confidentiality of the overall software system, but neither of the two mechanisms comes free of charge. For this reason, AtroPOS offers applications to choose either hardware-based or software-based isolation between kernel and user space, or to opt for no isolation at all. The choice can be made both statically during integration and dynamically at run time, at different spatial and temporal granularities: from entire applications down to individual cores and from permanence down to the very short term. Compared to OctoPOS, the separation between the kernel and the user program drastically alters the way control flows are handled: now there are two different types of control flows on two distinct levels of the system-software stack that interact with each other.
5 The Design of AtroPOS and InvRT

5.4 Control-Flow Model

As a result of the introduction of vertical isolation, the control-flow model of AtroPOS exhibits significant differences to the original one in OctoPOS – but it is also vastly dissimilar from the traditional threaded model of commodity operating systems. Conditioned by the separation between user space and the kernel, user control flows and kernel control flows are decoupled from each other and have separate execution contexts. Scheduling of these control flows takes place in both the kernel and the user application’s runtime system, with each level being responsible for its own control flows. Elementary claims play an important role as the kernel entities around which much of the scheduling revolves.

5.4.1 Multi-Level Scheduling

A schematic overview of the control-flow model and its different layers is given in Figure 5.3. In the figure, a circular arrow represents a scheduler; the items located around it are its corresponding scheduling elements (schedels). Schedulers exist on three distinct levels of the software stack and manage different types of schedels.

Claim scheduler  This scheduler operates at a low level of the operating system. Its schedels are elementary claims and their respective entry or resumption code sequences. By default, the claim scheduler is only activated whenever the allocation of computing resources changes: when a CPU core is newly added to an elementary claim, it gets to execute that claim’s entry sequence; when the
5.4 Control-Flow Model

core is removed from the claim, it enters an idle loop. In the special case where claim virtualisation
is enabled, more than one elementary claim may be associated with the CPU core, and the claim
scheduler becomes preemptive: once the current elementary claim has spent its time quantum, it is
preempted and a context switch to the next elementary claim is performed. Since the cores of a
virtualised elementary claim are gang-scheduled, claim switches happen simultaneously on all its
cores. That is, even if a core becomes idle and another claim has ready-to-run work for that core,
no claim switch takes place until the time slice has expired.

User scheduler The user scheduler is part of the application’s runtime system. Its inner workings
are virtually opaque to the kernel. As part of its interface to the kernel, a user scheduler exposes a
predefined entry point that performs an initial jump into the dispatch loop. The kernel carries out
an upcall to that entry point when a freshly invaded core becomes ready to execute user code. The
application is free to implement its user scheduler in the way that best suits its particular needs. In
the extreme case, every elementary claim of the user program may operate its own specialised user
scheduler with custom schedels. User control flows can exist in the form of i-lets, Cilk tasks, X10
activities or fibres, just to name a few possibilities. They are decoupled from the kernel control
flows; there is no interaction with the kernel unless the CPU core receives an interrupt, a trap is
triggered or the user program explicitly issues a system call.

Kernel scheduler In the aforementioned three exceptional situations – interrupts, traps and system
calls – control is transferred to the kernel, which handles the respective event in the form of a kernel
job: a kernel-internal control flow that mirrors the interrupted user control flow. Kernel jobs perform
their work either on behalf of the user program (in the case of system calls and, arguably, traps) or
at its expense (in the case of interrupts). A kernel job shares certain conceptual characteristics with
an i-let: it is a self-contained piece of code – a handler function – that usually runs to completion; it
is scheduled cooperatively and executed successively with other kernel jobs.

The relationship between user control flows and kernel jobs is asymmetrical. A user task can
commission a new kernel job but not vice versa – the kernel always returns control to the interrupted
user task, resuming it at the point where it was interrupted. The only exception to that rule are
upcalls to the user scheduler’s entry point. Notifying the user application of an asynchronous
event in which it is interested, for example of the completion of a system operation, is generally
carried out asynchronously and in an indirect fashion, not via an immediate control transfer. The
event-notification mechanism is explained in Section 5.4.4.

Depending on the concrete scheduling strategy, when a user scheduler runs out of work on a CPU
core, it may try to steal jobs from another core. If the attempted theft proves unsuccessful, the user
scheduler issues a system call that puts the core to sleep until new work becomes available\(^2\). Hence,
the kernel needs minimal knowledge about the internal state of the user scheduler: while the core
is idle, the kernel must be aware whether there is potential work for that core, and it must notice
when new work becomes available so it can wake up the core. The user scheduler must provide
this information through a standardised interface, whose technical realisation will be presented in
Section 6.8.4. Besides that, the runtime system is given full freedom in the implementation of its
scheduling strategies.

\(^2\)Certain newer processors such as the Intel Xeon Phi ‘Knights Landing’ allow executing the mwait instruction, which
halts the CPU and waits for an event, from user mode, without the need to perform a system call [CC16]. However, as
that feature is currently not widely available in other processors, support for user-mode mwait is not implemented in
ArkoPOS and InvRT.
Every entry into the kernel – be it caused by an interrupt, a trap or an explicit system call – leads to a single kernel job being commissioned and immediately executed on the local CPU core. In the base case, that job runs to completion before leaving the kernel again; the kernel scheduler does not get involved. It is only when a kernel job blocks during its execution – and when it later gets unblocked – that (minimal) scheduling functionality is triggered. Immediately before every return from the kernel, the kernel scheduler is queried for unblocked jobs that are ready to run. These jobs are resumed and completed one after another; it is only after all of them have been processed that the kernel is exited. A thorough walkthrough of the implementation of the kernel’s job scheduler is given in Section 6.2.

5.4.2 Hybrid Kernel Execution Model

Based on its handling of kernel jobs, AtroPOS implements a hybrid kernel execution model that draws inspiration from Mach [Dra+91] and also mirrors some aspects of the i-let model that was presented in Section 2.3.1. By default, as long as no kernel job blocks, AtroPOS follows the interrupt model: a single kernel stack is held ready by the operating system per CPU core; all system calls that are issued on a particular core and run to completion there are processed on the same stack. System calls that need to block in the kernel have the choice to either create a continuation or to block according to the process model. This choice is made by the developer of the respective system function on the basis of factors such as the amount of pre-block state that must be transferred to the post-block code. In cases where that state is sufficiently small, it offers itself to bundle it together in a continuation.

By contrast, if a kernel job blocks using the process model, for example because the state to be preserved is large, the current stack is moved aside and bound to the job, and a new kernel stack is taken from a pool and attached to the CPU core. Future kernel jobs are then processed on that stack. Once the blocked kernel job is resumed, its stack is temporarily restored and the job is executed on that stack. As soon as the job either finishes or blocks again, its stack is discarded.

Of the three categories of exceptional situations that trigger a kernel job, the focal point of the interaction between user and kernel control flows lies on system calls. This is elaborated on in the following paragraphs.

5.4.3 Synchronous and Asynchronous System Calls

The control-flow model of AtroPOS is inextricably linked with the way system calls are handled. System calls can be executed synchronously or asynchronously to the calling control flow. In the former case, the system call has the same semantics as a regular function invocation; in the latter case, the system operation is concurrent to the user control flow and synchrony must be established manually via a suitable mechanism.

Listing 5.1 shows the usage of a synchronous system call. Such a system call has straightforward semantics: it returns once the system operation in question has finished. However, without additional measures such as scheduler activations [And+92], blocking in the kernel triggers the thread-blocking anomaly. As described in Section 3.4.2, scheduler activations have been criticised as being overly complex and inefficient in terms of both memory footprint and temporal overhead.

Asynchronous system calls are an alternative to synchronous ones; they are able to avoid the aforementioned blocking anomaly without requiring scheduler-activation support. An asynchronous call may return early, before the system operation has finished. In that case, the user program must
Listing 5.1: Usage of a synchronous system call.
The system operation finishes before control is transferred back to the caller.

```c
int retVal = syncSyscall(arg1, arg2, arg3);
if (retVal != 0) {
    panic("syncSyscall failed");
}
```

Listing 5.2: Usage of an asynchronous system call, synchronised with the help of a future.
The system call returns early even though the system operation may still be unfinished. The function `future_force()` blocks the user control flow until the operation has finished and the return value is available.

```c
future_t future;
future_init(&future);
asyncSyscall(&future, arg1, arg2, arg3);

// ...

int retVal = future_force(&future);
if (retVal != 0) {
    panic("asyncSyscall failed");
}
```

explicitly wait for the completion of the operation by using a dedicated synchronisation mechanism such as a future [BH77]. An example of how an asynchronous system call is synchronised with the help of a future is given in Listing 5.2. As indicated with an ellipsis in line 5, the application programmer has the option to conduct latency hiding by inserting arbitrary code – possibly including other system calls – between the call site and the place where the result of the system call is used for the first time. If designed properly, the future’s `force` operation can be implemented purely on the application level, without direct interaction with the kernel but with full knowledge about the state of the user-level scheduler at its disposal. This way, it is possible for the concurrency platform to avoid the thread-blocking anomaly by construction, even without support for scheduler activations.

In the context of this thesis, asynchronous system calls are a means to an end: they circumnavigate the thread-blocking anomaly for blocking system operations, thus preventing a needless temporary degradation of parallelism in the application. The focus is not on achieving performance or predictability benefits through the concurrent, possibly parallel, execution of arbitrary system operations. The asynchronous offloading of system functionality in OctoPOS, for example communication, onto dedicated CPU cores has previously been investigated in a separate dissertation [Sch18], with the objective of reducing the interference between the operating system and the user program. Moreover, there is ongoing research into the dynamic scheduling of asynchronous operations in multi-layered software systems that span both the user and the kernel level – structured as a stack of intercommunicating *asynchronous abstract machines* [Mai+19]. This goes far beyond the scope of this dissertation.

Hence, AtroPOS does not flatly make all system functions asynchronous as Symunix does optionally [ELS88] or as Akaros does compulsorily [Rho+11]; nor does it make all kernel calls non-blocking as Panda [Ass+93b] does. Instead, it opts for a pragmatic approach and implements a simple two-pronged system-call scheme as displayed in Table 5.1: non-blocking system calls have
Table 5.1: System-call types implemented by AtroPOS.

<table>
<thead>
<tr>
<th></th>
<th>Non-blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>✔️</td>
<td>✗️</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>✗️</td>
<td>✔️</td>
</tr>
</tbody>
</table>

Synchronous blocking system calls would require kernel and user-space support for scheduler activations. Asynchronous non-blocking operations are not in the scope of this work.

Figure 5.4: Synchronous system call in AtroPOS.

The system call is executed on a dedicated kernel stack. After its completion, control is transferred back to the calling application.

A synchronous interface; system calls that may block in the kernel – either according to the process model or by creating a continuation – are asynchronous. This way, it is possible for application programmers to distinguish at a glance which system operations can potentially block and which ones cannot, by looking at their interface. From the application’s point of view, an asynchronous system call can be given synchronous semantics through a user-space wrapper that forces the future internally. Such a wrapper is specific to the runtime system and its concrete implementation of user control flows and futures. Interestingly, this approach by AtroPOS is the exact opposite of how Go [Mor13b] and PEACE³ [Sch88b] view asynchrony: instead of assuming a blocking kernel interface and establishing concurrency in user space, AtroPOS offers a non-blocking interface and leaves the institution of synchrony to user space.

The typical sequence of synchronous and asynchronous system calls in AtroPOS is depicted in Figures 5.4 and 5.5, respectively. In the graphics, the dashed horizontal line indicates the interface between the kernel and the application. Depending on the application’s isolation status, it also marks the separation between the two privilege levels. The semantics of a synchronous system call are straightforward, as illustrated by Figure 5.4: control is transferred to the kernel, which executes the operation in question and subsequently returns control to the application. If the application is isolated, the system call involves two privilege-mode switches that include an implicit context switch: one from the user stack to a dedicated kernel stack, and one back. However, if privilege isolation is disabled, a regular function call into the kernel is used instead. There is opportunity for further optimisations of system calls from privileged application mode; for instance, it is possible to forgo the stack switch and execute the system operation directly on the user stack. Details on these optimisations are given in Section 6.3.

The execution of an asynchronous system call that blocks in the kernel, for example an I/O operation, is shown in Figure 5.5a. The default method of blocking in AtroPOS is analogous to the one

³For PEACE, this applies only in conjunction with IPC; all other system calls in PEACE are non-blocking.
When the system call blocks in the kernel, control is immediately returned to the application; the user control flow waits for the system operation’s completion by blocking on a future. Later, an interrupt triggers the resumption of the blocked system call. When the system operation finishes, that event is signalled to the application, which then wakes the blocked user control flow.

Figure 5.5: Asynchronous blocking system call in ArnoPOS.

When the system call blocks in the kernel, control is immediately returned to the application; the user control flow waits for the system operation’s completion by blocking on a future. Later, an interrupt triggers the resumption of the blocked system call. When the system operation finishes, that event is signalled to the application, which then wakes the blocked user control flow.

Employed in user space (see Chapter 4 for the fundamentals and Section 6.2 for details on the kernel implementation), working on the basis of fixed-size kernel contexts and signal data structures. When the system call blocks, the CPU core switches to a new kernel context, which is fetched from a kernel-managed pool of pre-allocated contexts. Subsequent system calls, traps and interrupts arriving at the local core will be handled on the new stack. Immediately after the stack switch, control is returned to the calling application control flow. The caller may execute arbitrary further user code (see Listing 5.2) until it needs the result of the system operation, at which point the user activity must be synchronised with the kernel job. If the result is not yet available by then, the user control flow blocks on the future and makes way for other user tasks that are ready to run. Again, a context switch is performed – this time in user space.

Later, the completion of the I/O operation is signalled to the kernel by the hardware via interrupt. The resulting sequence is outlined in Figure 5.5b. The application is interrupted; the kernel is
entered and the interrupt is handled on the kernel stack that is currently associated with the CPU core. The blocked kernel job is set ready in the process. Before the interrupt handler returns, the kernel scheduler becomes active, takes the now-ready kernel job from the run queue and resumes its execution on its original stack. When the operation completes, it asynchronously notifies the application; the concrete mechanism for sending the notification of kernel events to user space is detailed in Section 5.4.4. After that, the kernel job exits and hands control back to the interrupt handler, which in turn returns to user space. The application continues its execution and processes the event notification. As a consequence, it sets the blocked user control flow ready again and eventually resumes it.

If the kernel control flow blocks not according the process model but with the help of a continuation, a slightly altered variant of that procedure emerges. This can be seen in Figure 5.6. Since all state is saved explicitly and is preserved inside the continuation, the existing kernel context can stay in use on the CPU core; no additional kernel context needs to be touched. When the system operation is resumed, its continuation is executed directly on the interrupt handler’s stack. From the user program’s point of view, the observable behaviour is fully identical, regardless of whether the blocking system call is implemented according to the process model or the interrupt model.

Figure 5.6: Asynchronous blocking system call using a continuation in the kernel.

The sequence is analogous to that in Figure 5.5, but the use of a continuation eliminates the need to switch contexts in the kernel.
5.4 Control-Flow Model

Figure 5.7: Event-notification scheme.
In this figure (and in Figures 5.8 and 5.9), a thick black curved arrow with a flat head denotes a copy operation; a thin coloured angular arrow with a long head and a circle as its tail represents a reference. The event object is allocated in user space; a reference to it is passed to the kernel. To notify an elementary claim of the event, the reference is inserted into a shared queue, from which it is picked up by the user scheduler and processed.

5.4.4 Event Notification

One particular requirement of blocking system calls is a mechanism for communicating the completion of an asynchronous system operation back to user space. Such a mechanism should be both simple and generic in order to allow arbitrary runtime systems to implement their own dedicated user-level blocking data structures on top of it. The kernel dictates neither the layout nor the semantics of such synchronisation data structures; it merely provides a versatile foundation for them to build upon. This foundation is called a kernel event.

As depicted in Figure 5.7, a kernel event is a user-allocated object that resides in the application’s address space and is targeted at an elementary claim. By convention, all asynchronous system calls take as argument a reference to such an event object. When the user program needs to launch an asynchronous kernel job and wants to be notified of its completion, it allocates a kernel-event object and passes a reference to that object as argument to the system call. The kernel memorises the reference until the job finishes, at which point it inserts the reference into an event queue residing in a memory area that is shared between kernel and user space. Every elementary claim possesses its own event queue, which receives the events directed at that claim. In principle, any tile-local CPU core is able to insert items into an event queue, and any core that is part of the associated elementary claim can remove and process events. Consequently, the queue generally has MPMC semantics. The runtime system is expected to routinely poll the event queue as part of its regular dispatch loop. In other words, the runtime system merges the functionalities for control-flow dispatching and event processing into a single combined loop. Kernel events are thereby injected into the program’s regular stream of work packages.

As kernel events are passed around between the application and the kernel with by-reference semantics, the kernel needs only minimal knowledge about the internal structure of an event object. Essentially, the kernel is handed an opaque reference, retains it for a while and eventually gives it back to the user program, without knowing anything about the memory contents behind the reference. This makes the event-notification mechanism extremely flexible and suitable for all different types of application-level synchronisation and notification patterns. In Section 5.6.4, it

---

4Note that the event queue in ArnoPOS is semantically different from the homonymous concept that was proposed by Hansen [Han72] in the context of critical regions. Hansen’s event queue holds processes waiting for a condition on a shared variable; when a critical region is left, all processes in the associated event queue are woken up.
will be described how the invasive runtime system builds upon kernel events to implement futures, remote i-lets and notification i-lets for push data transfers.

The event queue residing in shared memory constitutes part of the contract between kernel and user space. It is designed in such a way that the user application needs to trust the kernel, but not vice versa. An application may accidentally or intentionally manipulate or destroy the event queue of one of its own elementary claims, but it must neither be able to access the queues of other applications nor to abuse an event queue as a gateway for exploiting the kernel (for instance via a dangling pointer). Provided privilege isolation and memory isolation are both enabled, the repercussions of any damage done by the application are restricted to the scope of the application itself.

### 5.4.5 Summary

Pursuant to the mantra that the application knows best how to represent and schedule its own concurrent control flows, AtroPOS leaves the implementation of all user-space scheduling aspects to the runtime system and exposes raw (physical or virtual) CPU cores to the upper layers of the software stack. The extent to which scheduling is performed in the kernel is kept small or even minimal (in the absence of claim virtualisation). AtroPOS implements a hybrid kernel execution model that follows the interrupt model in the default case but opportunistically becomes process-based by binding a kernel stack to a kernel job if that job blocks. In order to accommodate the potential of kernel jobs to block during their execution, system calls that trigger such jobs have asynchronous semantics from the caller's point of view. On the other hand, system calls that are guaranteed not to block in the kernel retain their synchronous semantics. The completion of an asynchronous operation is signalled to the application with the help of a kernel event – a reference to a generic, application-defined data structure that is inserted by reference into a per-elementary-claim event queue when the operation has finished. User schedulers are responsible for processing items from their respective event queue in addition to their regular dispatching of tasks. Event queues also play an important role in the kernel's communication interface, as we will see in the next section.

### 5.5 Inter-Tile Communication

AtroPOS retains the distributed-multikernel [Bau+09] approach of OctoPOS, where a separate operating-system instance runs per tile or per NUMA domain. User applications whose claims are spread out over more than one tile are logically split into parts that exchange data through message-based communication, with no assumption of inter-tile cache coherence from the application's point of view. We differentiate communication into two types: kernel-level communication – that is, the exchange of messages between two instances of AtroPOS that run on different tiles – and application-level communication between parts of a distributed user program.

#### 5.5.1 Kernel-Level Communication

This class of communication is performed whenever a local operating-system instance needs to access or modify a non-local portion of the distributed system state. A typical example of this is the low-level invade operation, which is a system call the user application makes in order to stake a new
elementary claim. When the kernel job determines that the tile to be invaded is remote, it sends a request to the kernel instance that is responsible for the other tile, which processes the request and sends back its reply. The result of the system call is then signalled back to the calling application.

Like OctoPOS, AtroPOS resorts to RPCs built upon sys-i-lets, or upon the software emulation thereof on x86_64 (using shared memory and IPIs), for inter-tile communication on the operating-system level. The kernel job that issues the RPC may subsequently need to wait for the result to become available until it can continue its execution. Consequently, all system operations that use blocking RPCs must be implemented as asynchronous system calls. Waiting in the kernel for the result of an RPC is achieved as described in Section 5.4.3, either by blocking or by creating a continuation that is executed when the response arrives. Either way, the system function operates asynchronously from the user program’s point of view; its completion or termination is signalled to the application with the help of a kernel event.

5.5.2 Application-Level Communication

For communication between separate parts of a distributed application, new low-level mechanisms are added that supersede the existing ones. In OctoPOS, exchanging data messages on the application level relies heavily on the forth-and-back transfer of i-lets. By contrast, AtroPOS has no representation of i-lets on the kernel level as it considers i-lets to be purely application-level entities. Thus, its new communication primitives, albeit aiming in the same general direction, operate on a lower level of abstraction.

An important design goal is that the kernel-provided mechanism for communication between elementary claims be kept generic. In particular, it should be independent of the concrete structure and implementation of the user scheduler – see Section 5.6.2 for further reasoning about this design aspect. Like PEACE with its messages and high-volume data transfers [Sch88b; GS89], AtroPOS offers two mechanisms with different semantics and degrees of complexity: messages and push data transfers. To ensure that only claims of the same application can exchange data with one another, both primitives are part of the privileged kernel, which performs the necessary checks.

**Messages**  A message is a small application-level object that is sent asynchronously to an elementary claim located on a remote tile. Its size does not exceed that of a cache line. The process of sending a message from one tile to another is shown in Figure 5.8. A flat copy of the sender’s message object is made and is stored in a destination-side buffer\(^5\) (step 1). That buffer is provisioned by the operating system and shared with the user program. A reference to the message copy is subsequently enqueued into the receiving claim’s event queue, from which it is eventually picked up by the claim’s user scheduler (step 2). This allows the application to react to an incoming message in the same way as to a kernel event. The contents of a message are entirely application-defined and are copied as is. With this, messages are a suitable basis for the implementation of higher-level application constructs such as remote i-lets.

**Push data transfers**  In scenarios where bigger amounts of data have to be transferred from one tile to another, messages quickly reach their size limit. For this reason, AtroPOS also offers a primitive for performing push data transfers. Its overall semantics are akin to those of the corresponding OctoPOS primitive: an arbitrary number of bytes are copied asynchronously from a source buffer into

\(^5\)The astute reader, whose mind may be circling around the limited capacity of that buffer, is hereby referred to Section 6.6 in the next chapter.
The message contents are copied into the remote tile’s memory and the receiving elementary claim’s event queue is used to notify the application. As soon as the data copy (step 1) is complete, a reference to the kernel event is pushed into the sending claim’s event queue and the message is sent to the destination claim using the primitive described above (step 2). The local event and the remote message are subsequently processed by the respective parts of the application (steps 3 and 4). The combination of a local kernel event and a remote message makes it easy for the runtime system to implement a higher-level wrapper that mimics the OctoPOS push-data-transfer function by using local and remote i-lets for signalling – but the application is free to use arbitrary other notification mechanisms, for example futures.

### 5.5.3 Summary

The mechanism for kernel-level inter-tile communication stays largely the same compared to OctoPOS, but it is more generic and accommodates the new control-flow model of AtroPOS. Application-level communication is based on messages and push data transfers; it is seamlessly integrated with the kernel’s generic event-notification mechanism. Dedicated hardware support is possible, but not necessary. Applications are given the opportunity to implement their own specialised communication system on top of these primitives.

### 5.6 Invasive Runtime System

The AtroPOS kernel covers only the minimal operating-system functionality required to build parallel applications for tiled hardware architectures: device drivers, coarse-grained memory allocation (at the granularity of pages), low-level resource management on the basis of elementary claims, communication primitives and others. Provided claim virtualisation is not enabled, disjoint subsets of the physical computing resources are made available to applications in the form of elementary claims. The kernel exposes a low-level system-call interface whose isolation characteristics
5.6 Invasive Runtime System

Figure 5.9: Push data transfer.
This operation asynchronously copies the contents of a source buffer to a given destination address on the receiving tile. Afterwards, the sending elementary claim is notified via kernel event (see Figure 5.7) and the receiving claim is notified via message (see Figure 5.8).

are configurable both statically and dynamically: either for protection or for predictability and performance.

What AtroPOS does not offer are higher-level abstractions for direct consumption by the programmers of concurrent applications. A runtime system is required to bridge the gap between the raw CPU cores exposed by the operating system and the user program’s fine-grained concurrent control flows. As this functionality is performance-critical, it is consequential to locate it entirely above the kernel–user demarcation line. This section discusses what such a runtime system must look like, using the example of InvRT, a runtime library that brings the invasive programming interface to AtroPOS. Applications originally written for OctoPOS can run unmodified on InvRT and AtroPOS. While certain aspects of InvRT are specific to the field of invasive computing, the fundamental mechanisms described are also directly transferable to other runtime systems.

A high-level overview of the invasive runtime system within a single tile is depicted in Figure 5.10. The runtime library is logically seen as part of the application. Its main purpose is to schedule and dispatch fine-grained tasks of the user program to the CPU cores exposed by the operating-system kernel. Scheduling is performed at the granularity of elementary claims. The cores within an elementary claim are treated equally and are fed with work by the scheduler; tasks that are directed at a claim can be processed by any of its cores but do not implicitly migrate to cores outside the claim. Every elementary claim can have its own custom scheduler implementation. A user scheduler may exchange tasks with other elementary claims of the same application through a standardised interface, and it may receive events and messages as described in the preceding section.

5.6.1 Resource Management

As mentioned previously, AtroPOS exposes accessible computing resources to the application in the form of elementary claims. Establishing more advanced resource-management concepts at a
Figure 5.10: Tile-local overview of the invasive runtime system.

The primary purpose of the runtime system is the scheduling and dispatching of fine-grained user tasks to CPU cores at the granularity of elementary claims. User schedulers may exchange tasks with other elementary claims of the same application (horizontal arrows) and may receive events from the kernel (vertical arrows).

A higher level of abstraction falls under the responsibility of the runtime system. In the simplest case, the runtime system uses only a single elementary claim that comprises all CPU cores available, and distributes the work among them. This scheme is applicable for non-invasive programs that are not resource-aware; it can serve as a basis for ports of established commodity concurrency platforms such as Cilk Plus, OpenMP or Go to single-tile hardware platforms.

InvRT, by contrast, implements the hierarchical cross-tile claim model of OctoPOS that was introduced in Section 2.3. Elementary claims of different resource types within the same tile are encapsulated into per-tile claims, which can in turn be bundled together as sub-nodes of an overarching cross-tile claim – resulting in a tree-like claim structure. The higher levels of that resource hierarchy are managed in user space by the runtime system.

5.6.2 User Scheduling and Dispatching

The user scheduler is the runtime system’s central component where all threads converge\(^7\). It takes the concurrent application’s fine-grained user tasks and dispatches them to the associated CPU cores. All scheduling functionality related to these user tasks is executed in user space, with no additional kernel-internal thread scheduler making independent and potentially adverse decisions. The runtime system instantiates a separate user scheduler per elementary claim, which grants programmers the possibility to perform specialised handling of control flows in different logical parts of their program. Figuratively speaking, the user scheduler can be regarded as a box that has interfaces in all directions, as shown in Figure 5.11: to the user application, whose tasks it is supposed to juggle; to the kernel, by which it is initially activated and from which it can receive events; and to other elementary claims, which may wish to communicate with it.

\(^7\)This is of course purely metaphorical – remember, there is no such concept as threads in ArnoPOS.
Figure 5.11: Interfaces of the user scheduler.

The user scheduler has to interact with, and receive input from, other software components in both vertical and horizontal dimensions.

By design and on purpose, zero prescriptions are made with respect to the scheduler's interface to the application. The runtime system is given full creative freedom over that interface and over the inner workings of its scheduler, including which strategies it implements, how it represents control flows and their execution contexts, and what synchronisation primitives it offers. This ensures maximal flexibility in terms of the kinds of concurrency platforms that can be ported to ArroPOS.

For the same reason, the contract between the runtime system and the kernel is kept minimal. It consists of three elements: the aforementioned event queue, a standardised entry point into the scheduler, and a method for the kernel to query whether there is any outstanding work. The entry point denotes the location where a newly invaded core is to commence its execution of user code. Every user scheduler must define such an entry function. When an application wishes to invade a set of computing resources, it passes the respective function pointer as argument to the `invade` system call, which upon completion triggers an upcall to that function on all freshly acquired CPU cores. The outstanding-work indicator is used by the kernel to decide whether to wake up cores that are currently asleep.

The prime element of the contract is the per-elementary-claim event queue, which is managed by the kernel but resides in a shared memory area that is both readable and writeable by the application. The structure and interface of the event queue is predefined by the operating system. The user scheduler is obliged to poll and empty the queue as part of its regular dispatching routine. When, how and with which priority events are handled relative to application tasks is at the runtime system's discretion.

In the horizontal dimension, different elementary claims belonging to the same application should be able not only to manage work packages each by itself but also to exchange work with one another. A work-sharing scheduler can offer a callable function interface for outside cores to push work items directly into its run queues; it has to ensure that all accesses to the queues are properly synchronised. For work-stealing schedulers, by contrast, such an interface would be undesirable because their efficiency largely relies upon the fact that a run queue is only ever written to by its owner and by no one else – a constraint that could obviously no longer be upheld. Since barging in the user scheduler's internals is clearly not the right way to go, a simple and generic solution is chosen instead: as anticipated in Section 5.5.2, the event queue also serves as a standardised interface for horizontal inter-claim communication. If the sending and the receiving elementary claim reside on different tiles, they make use of the kernel's message primitive. If they are located on the same tile, they communicate via shared memory and without direct kernel involvement: as both parties are within the same address space, it is sufficient for the sender to allocate a message object and to subsequently insert a reference to it into the receiver's event queue. Note that the
sending and the receiving claim do not necessarily have to share the same scheduling logic or even a compatible representation of their internal work packages; as the event queue decouples the two from each other, it is entirely sufficient to define a common format for inter-claim messages. In summary, the event queue serves as a simple standardised interface for inbound communication from both the kernel below and from other elementary claims sideways.

**InvRT** contains two different i-let schedulers to choose from: an adapted variant of OctoPOS’ original work-sharing scheduler that emulates the behaviour of the hardware CIC, and a randomised work-stealing scheduler that aims to be more efficient and permits load balancing within the confines of the elementary claim. Both schedulers manage per-core run queues with MPMC FIFO or work-stealing LIFO semantics, respectively. In addition, they poll a centralised event queue as part of their regular dispatching routine and call the appropriate handler for all events retrieved this way. Implementation details are given in Section 6.8.

### 5.6.3 Synchronisation

All the improvements and optimisations proposed in Chapter 4 are carried over from OctoPOS to the InvRT runtime system, including the efficiency enhancements in the management of execution contexts, the post-switch-hook technique, handoff scheduling and more. InvRT provides the same synchronisation data structures as OctoPOS, especially the wait-free binary signal and the new counting signal. The runtime system’s synchronisation primitives are tightly integrated with the respective user schedulers and operate purely on the user level. This makes them suitable for serving as a basis for the custom application-defined event-notification mechanisms described in the next subsection. In particular, the blocking future data structure is composed of a binary signal.

### 5.6.4 Event Notification on the User Level

From the AtroPOS kernel’s point of view, the generic event-notification mechanism centres around the circulation of references to mostly opaque user data structures. This section describes how the runtime system fills these data structures with life in order to realise a number of common use cases on the application level. The use cases cover both the notification of a kernel job’s completion (via future or completion i-let) and the transmission of messages between elementary claims (also via i-let).

The program logic for allocating, interpreting and handling kernel-event and message objects resides entirely within the runtime system. There can be different types of events, so the handler routine is responsible for interpreting the concrete type and executing the corresponding action for each event individually. Fundamentally, there are two possible ways to achieve this:

1. The event object is regarded as a sort of data packet – similar to a network packet. It contains a leading type field and optional serialised data. The handler function consists of a large switch-case statement that, depending on the value of that field, deserialises the contents and performs a certain action with it.

2. Following an object-oriented approach, the event is interpreted as a polymorphic object (or as a kind of active message [Eic+92], for that matter). It contains a number of member variables and a pointer to a method that operates on these variables. The runtime system’s event-handler function invokes that virtual method.
INVRT makes use of the second option because it is both more flexible and more extensible. The following paragraphs show how common elements of the invasive execution model – namely futures, completion i-lets and remote i-lets – are mapped to kernel events and messages.

**Future**  As described in Section 5.4.3, futures are the preferred way to wait for the completion of asynchronous system calls. In INVRT, a future object is a kernel event that encapsulates a binary signal (see Section 4.2.1 for more details). The event’s handler method calls the signal’s `signal` operation, and the future’s `force` operation calls `wait`. This way, an i-let execution blocking on `force` is unblocked as soon as the kernel event gets processed by the runtime system.

**Completion i-let**  As an alternative to using futures, the caller of an asynchronous system call may prefer a continuation-based style instead, where a `completion i-let` is set up to be dispatched once the system operation finishes. The completion i-let is wrapped into a kernel-event object whose handler method directly executes the i-let\(^8\). This data structure makes it possible for INVRT to provide a push-data-transfer function that is based on the aforementioned system call but whose interface and semantics are identical to the one originally offered by OctoPOS.

**Remote i-let**  Sending an i-let from one elementary claim to another is achieved by wrapping it into a message object and sending that message to the destination. Apart from minor implementation differences, remote-i-let messages are technically similar to completion-i-let events.

One semantic distinction between remote i-lets on the one hand and futures and completion i-lets on the other is the fact that the latter two, being means of notifying the application of the termination of a system call, must be able to convey the possible occurrence of a runtime error. The implications of this constraint are discussed in the next chapter in Section 6.4. No such constraint is imposed on remote i-lets: if the operation fails, the message is simply not sent and the would-be sender is responsible for handling the error; the recipient does not get involved.

A survey of existing invasive real-world applications reveals that in many cases where a push data transfer is used in combination with a completion i-let, no continuation-based approach is applied: often, the caller blocks on a future-like signal data structure and the sole purpose of the completion i-let is to notify that signal. With AtroPOS, the code of such applications can be simplified by replacing the construct of completion i-let plus signal with the future object proposed above. This requires only minor modifications to the program’s existing code base. It also has a slight positive impact on the efficiency of these bits of code because no dedicated i-let, cheap as it may be, needs to be allocated, initialised, passed around and dispatched. One aspect to keep in mind in this context is the priority with which events are processed relative to regular application i-lets: while OctoPOS treats all i-lets the same regardless of their origin, which may cause completion i-lets to queue up behind other i-lets, INVRT could opt to assign precedence to incoming kernel events.

### 5.6.5 Usage of the Invasive Hardware Components

On the invasive prototype platform, applications built on INVRT can make use of the hardware-accelerated scheduling, dispatching and communication facilities offered by the pertaining special hardware components, namely the GIC and the NoC adapter. There are several possible ways

\(^8\)It is possible to achieve semantically equivalent functionality via the future’s `onCompletion` operation (see Section 4.2.3), but a specialised completion-i-let data structure is a more efficient solution because it manages without atomic instructions.
to implement software support for this. Since the two components are controlled entirely via
memory-mapped registers, the easiest method is for the kernel to expose the pertaining address
pages to InvRT. This is both very simple and very efficient but has severe security implications:
as neither the CiC nor the network adapter were designed with security as a first-class concern, they
are open to potential abuse. For instance, a malicious application could exploit the CiC to inject
i-lets into another application; it could abuse the NoC’s push functionality to overwrite arbitrary
data on a remote tile; or it could elevate its own privileges through a crafted sys-i-let.

As an alternative, the device driver itself can be moved into AtroPOS, which then exposes a set
of system calls for the corresponding functionality. With this, the kernel has full control over all
arguments and can perform appropriate permission checking to prevent any abuse. An obvious
disadvantage of this approach is that every such operation must cross the user–kernel boundary –
which may be acceptable for communication but hurts performance when every single scheduling
and dispatch operation requires a system call.

A third option would be to modify the hardware implementation and interfaces of the CiC and the
NoC adapter in a way that only those portions of the functionality can be selectively exposed to
user space that do cannot violate security constraints. Overall, the number of changes required
would be manageable; a brief discussion of those modifications can be found in Appendix B.

5.6.6 Summary

InvRT, the invasive runtime system built upon the abstractions provided by AtroPOS, offers the
same programming interface as OctoPOS and is capable of serving as a drop-in replacement for
it. Its primary responsibilities are the higher-level resource management and the scheduling and
dispatching of i-lets in user space. InvRT makes use of the partly asynchronous system-call interface
and employs the generic event-notification mechanism of AtroPOS as basis for both inter-claim
and kernel-to-claim communication.

5.7 Applicability to Other Runtime Systems

How well are the design of AtroPOS and the principles on which InvRT is built applicable to the
runtime systems of other concurrency platforms such as Cilk Plus, Go or OpenMP? When porting
an existing runtime system that originally targets the POSIX API to AtroPOS, the three primary
differences to consider are the following:

• **Cores instead of threads.** Unlike a commodity operating system, AtroPOS directly exposes
  CPU cores to user space, without threads as an intermediary abstraction.

• **Asynchronous system-call interface.** Since not all system calls have synchronous semantics,
  one can no longer flatly assume that a system call does not return to its caller until it has
  finished. The runtime system must provide a mechanism for application code to handle such
  situations in user space, either through a blocking future-like data structure or with the help
  of continuations, or both.

• **Kernel-event queue.** The runtime system must be aware that there is an additional source of
  work that requires processing: it must poll the event queue as part of its regular scheduling
  procedure.
The first item on the list is relatively simple to take care of by creating a single big elementary claim that contains all computing resources assigned to the respective application and does not change over time, effectively crossing out the resource-aware aspect for that particular application. This implicates that such an application cannot span more than one tile without taking special measures that reflect that fact. One consequence of this is that the application will never make use of the facilities for inter-claim or inter-tile communication.

Coping with the different system-call interface is arguably the biggest challenge for bringing alternative concurrency platforms to AtroPOS because it affects not only the runtime system but potentially also the application program itself if the latter performs blocking system calls from within concurrent activities. In particular, this is a concern for Go because its runtime library is designed around the assumption that the kernel interface is synchronous. On the other hand, it must be stated that avoiding the thread-blocking anomaly by construction (and without scheduler activations) is a unique characteristic of AtroPOS that naturally entails a certain amount of fallout. The runtime system can make porting programs easier by offering a synchronising wrapper around system calls, giving them the familiar semantics from the user program's point of view.

In the single-claim scenarios touched upon above, where one elementary claim holds all the application's CPU cores, the central event queue belonging to that claim may become a scalability bottleneck as all cores constantly thrash the same queue. A clever runtime system can alleviate that thrashing by assigning the computing resources to multiple elementary claims as opposed to a single one. With each elementary claim having its own event queue, the user scheduler can be built in a way that it serves the union of the application's elementary claims but polls only the respective core's nearest event queue.

Two concrete alternative runtime systems are evaluated in the context of this dissertation: the existing invadeX10 library and the Cilk Plus runtime library, libcilkrt. Porting the invasive X10 library to AtroPOS requires no modifications to its source code because it is built on top of the OctoPOS interface, and InvRT is fully API-compatible. However, the X10 runtime library is enhanced to use the wait-free binary signal and the handoff-scheduling primitives where possible. By contrast, a dedicated port is conducted for the Cilk Plus runtime library. While it is not a full-blown and feature-complete variant of libcilkrt, it contains all the functionality required to get a selected set of Cilk Plus applications from Intel's official suite of sample programs to run on a single-tile AtroPOS system.

5.8 Summary

Motivated by the introduction of vertical isolation, the invasive system-software stack with OctoPOS at its core has been subjected to a major redesign in the context of this thesis. The results of said effort are AtroPOS, a stripped-down operating-system kernel that provides low-level abstractions and is intended as a foundation for concurrency platforms, and InvRT, a runtime system that implements the invasive application and execution model. One unique feature of AtroPOS is its optional, dynamically configurable vertical isolation: applications can be separated from the kernel through either hardware-based or software-based means, or isolation can be disabled altogether.

Alternatively, for cache-coherent NUMA systems, one could build a variant of AtroPOS that treats the entire hardware system as one large tile, irrespective of its topology. However, making such an operating-system variant competitive in terms of performance would require implementing memory-locality optimisations that are not necessary in a partitioned system, such as Linux' first-touch policy.
This choice can be made both statically and dynamically, and at different spatial and temporal granularities.

The AtroPOS kernel exposes computing resources to the upper layers of the software stack as is; unlike traditional operating systems such as Linux, it does not offer higher-level abstractions such as kernel threads. The kernel execution model is hybrid: it is event-based by default but becomes process-based on demand when a kernel job blocks. AtroPOS distinguishes between system calls that do not block and have synchronous invocation semantics, and system calls that behave asynchronously because they may block during their execution. The kernel notifies the user program of the completion of such an asynchronous operation by pushing a kernel event – a reference to a user-defined object – into a shared event queue. Communication between parts of a distributed application is carried out via messages and push data transfers; messages are handled like kernel events and use the same mechanism.

InvRT, the invasive runtime system, is a user-level implementation of the foundations of the invasive execution model. Its primary task is the efficient scheduling and dispatching of i-lets onto the CPU cores exposed by the kernel. InvRT is built on top of the low-level interface exposed by AtroPOS, making heavy use of the kernel-event mechanism to implement application constructs such as futures, completion i-lets and remote i-lets.

Furthermore, AtroPOS is designed to be able to serve as a generic foundation for various concurrency platforms, as demonstrated by the port of the Cilk Plus runtime system.
The design considerations discussed in the previous chapter have been cast into a prototypical implementation of the AtroPOS kernel and the InvRT runtime system. This chapter shines a light on a select set of particularly relevant focal points, on technical choices that were made and on the reasoning behind them.

Figure 6.1 gives a high-level overview of the resulting stack of software components. Note that, even though the kernel part is pictured at approximately the same size as OctoPOS is in Figure 2.5 on page 22, AtroPOS bundles less functionality and its interface is located at a lower level of abstraction. An overview of the kernel interface is given in Section 6.1. Subsequently, Sections 6.2 and 6.3 explain in detail with the aid of flow charts the kernel-internal handling of control flows, triggered by interrupts or system calls. The concrete implementation of the mechanisms for event notification and cross-tile communication is presented in Sections 6.4 and 6.5, and a solution to the issue of bounded buffer space is given in Section 6.6. Section 6.7 briefly explains the cross-cutting considerations that have flowed into the security model of AtroPOS. Finally, select aspects of the implementation of InvRT, the invasive runtime system, are detailed in Section 6.8.

All platform-specific code in AtroPOS is encapsulated into abstractions, allowing the kernel to retain its portability. Note that this chapter puts a strong focus on x86_64 as the default target platform. All descriptions are made from that perspective, and the portability aspect, albeit present in the source code, is not explicitly highlighted in the text. While the notion of global cache coherence is not passed on to applications, the software implementation of the kernel's communication primitives for x86_64 NUMA machines exploits the availability of cache coherence for efficiency reasons. This is not a limiting factor because, given appropriate hardware support, AtroPOS can operate equally well on tiled systems that are not globally cache-coherent.

6.1 Kernel Interface

The kernel interface is the gate through which user applications – or their runtime systems – interact with AtroPOS. It consists of an active component in the form of system calls, through which the application can direct execution into the kernel, and of a passive component in the form of shared
6 Implementation Details

Figure 6.1: Overview of ArroPOS (bottom) and InvRT (top).

The numbers of this chapter’s sections are annotated on the same level as the respective software components covered by them.

memory regions and data structures that allow the efficient exchange of information without explicit control transfers.

6.1.1 System Calls

System calls are the generic mechanism that permits application programs to actively and immediately transfer control to the kernel in order to have it execute an operation on the application’s behalf. System-call code is divided into three parts from top to bottom: a user-space stub – logically part of the application – that sets everything up and triggers the actual system call; a unique kernel entry point that gets activated in the process, saves (and later restores) the application context as necessary and redirects control to the desired operation; and the actual kernel-side implementation of the operation in question – referred to as kernel function.

System-call stub
A system-call stub is a callable function that obeys the compiler’s regular user-space calling conventions – for instance, the System V ABI [Mat+14] on x86_64 – and activates the appropriate mechanism for entering the kernel. In most operating systems, the stub is made available to programmers as component of the libc – but in ArroPOS, due to its optional privilege isolation, stubs can contain logic that is tangled with the configuration of the operating system. Hence, stubs are provided by the kernel itself, with a specialised stub manifestation existing per system operation. They reside in an executable memory section that is accessible from user space.

Entry points
The kernel has two separate entry points through which it can be entered by system calls: one for calls invoked from unprivileged user code and another for calls from privileged code. The entry points serve as the glue code that receives control from the calling application, saves its
processor state, switches to a kernel stack and sets up the kernel job in charge of executing the actual kernel function. After the kernel function returns, the entry point restores the user stack and processor state, and directs control back to the stub.

**Kernel function**  Kernel functions are regular callable functions that are part of the operating system's code base. A kernel function carries out a specific operation on behalf of the calling program. It is executed in the context of a kernel job and may potentially block the job during its execution.

Implementation details of the system-call mechanism are elaborated on in Section 6.3.

### 6.1.2 Shared Memory

Certain information held by the kernel is exposed to the application in the form of shared memory: kernel pages that are made accessible to user space. This allows user programs to efficiently query for information without having to issue a system call, but also to exchange data in both directions.

Mapping kernel pages into the user address space is a technique used extensively by Linux, known under the name *virtual dynamic shared object (vDSO)*. A vDSO behaves as if it were a shared library that is linked dynamically during application start-up. The dynamic linkage makes it possible to enable address-space layout randomisation (ASLR) for the Linux kernel, which hardens it against common attacks. By contrast, AtroPOS adopts the simpler memory-mapping scheme of OctoPOS as described in Section 2.3.3, where the kernel and the applications all have different views of the same single address space. An application's address-space view is restricted on a per-page basis by means of present- and supervisor bits: pages not belonging to the application are marked as absent; pages that are part of the kernel are present but have their supervisor bit set to one.

Shared memory regions can be roughly divided into two categories, determined by their lifetime and permissions: static data and dynamic per-claim information.

#### Static Data

These are memory pages that contain immutable system information and are mapped read-only (r--), allowing applications to efficiently retrieve basic architectural and topological data. Among this information are the local tile identifier (ID), the number of tiles in the system, the IDs of I/O tiles, the number of cores in the local tile and the calibrated frequency of the timestamp counter. Moreover, there are per-core pages, each of which holds the ID of the corresponding CPU core. The static-data category also comprises the aforementioned system-call stubs, which reside in pages mapped with permissions r-x.

#### Per-Claim Information

Elementary claims are the basic operating-system abstraction for representing a set of computing resources. They are managed by the kernel on behalf of the application and are referred to with a numerical tag. A read-only view of an elementary claim's state is exposed to user space for use by the application's runtime system. This state comprises the claim's identifying tag, the ID of the application to which it belongs, the claim's privilege status and a descriptor for the set of cores held by the claim, among others. This information is read-only from the application's point of view but
can be modified by the kernel. As x86_64 offers no direct way to express these dual permissions in a single page descriptor, the physical page is mapped twice: once for the kernel as rw- with the supervisor bit set, and once in the application’s address space as r--.

In addition to those read-only pages, the kernel exposes mutable data structures for event queues and message contents, which can also be written to from user space. Their usage is explained in Sections 6.4 and 6.5, respectively. Also included is status information concerning which of the claim’s cores are currently idle – required for selectively waking up sleeping cores. The user-level data structures and the role they play with respect to user scheduling are addressed in Section 6.8.

6.2 Implementation of the Kernel Execution Model

The implementation of the kernel control-flow model of AtroPOS bears similarities with that of the i-let model in several respects. The execution contexts of kernel jobs are called kernel stacks or kernel contexts; the two terms are used interchangeably in this chapter. Like i-let contexts, kernel contexts have a fixed size that is a power of two, and are aligned at a multiple of their own size. As described in Section 4.1.1, this makes computing the current kernel job’s context very efficient. Kernel contexts are preallocated and kept in a pool.

Despite the similarities with the i-let model, there is a major difference concerning the schedels that are handled by the respective scheduler. The schedels of the i-let model are runnable i-let incarnations that are stateless in the sense that they have no associated execution context (hence the wakeup-i-let construct as a means of restoring an existing context that holds a blocked i-let execution). By contrast, the schedels juggled with by the kernel scheduler are stateful kernel jobs that always carry an associated kernel context. This makes kernel jobs related more closely to fibres than to i-lets. By extension, this means that every dispatch of a kernel job is accompanied by a context switch, whereas many successive i-let executions can share the same user stack.

The primary synchronisation data structure in the kernel is a variant of the wait-free binary signal that was introduced in Section 4.2.1. The kernel-internal binary signal handles kernel-context pointers as opposed to i-let pointers. Like its application-level counterpart, it makes use of a post-switch hook to avoid race conditions. An explanation of how the blocking and unblocking of kernel jobs is implemented is given over the course of this section and the next.

6.2.1 Kernel Entry and Exit

Kernel jobs are triggered by system calls, interrupts or traps. As the handling of asynchronous interrupts and synchronous traps is implemented virtually identically in AtroPOS, the use of the term interrupt in the remainder of this section will refer to either of the two. A kernel job that is triggered by an interrupt or trap must not block using the process model; synchronisation aside, it is mandatory that control always be returned to the user program as soon as the kernel can no longer make immediate progress. If such a job cannot complete its work in one session, it must set up a continuation that later carries out the remainder of the work in a separate kernel job.

AtroPOS holds three kernel stacks ready for each CPU core in the system: a worker stack, a small (single-page-sized) interrupt-entry stack and an extra stack for handling non-maskable interrupts (NMIs). System calls and interrupts are generally processed on the worker stack. When a system call blocks according to the process model, another kernel stack is taken from the pool and becomes the new worker stack of the core in question.
6.2 Implementation of the Kernel Execution Model

<table>
<thead>
<tr>
<th>From unprivileged application</th>
<th>From privileged application</th>
<th>From kernel job</th>
</tr>
</thead>
<tbody>
<tr>
<td>System call</td>
<td>Interrupt</td>
<td>System call</td>
</tr>
<tr>
<td>syscall</td>
<td>interrupt request (IRQ)</td>
<td>call</td>
</tr>
<tr>
<td>syscall</td>
<td>interrupt request (IRQ)</td>
<td>ret</td>
</tr>
<tr>
<td>syscall</td>
<td>interrupt request (IRQ)</td>
<td>(in software(^1))</td>
</tr>
<tr>
<td>syscall</td>
<td>interrupt request (IRQ)</td>
<td>in software</td>
</tr>
</tbody>
</table>

Table 6.1: Kernel entry and exit on x86_64.

The kernel’s entry point is responsible for switching to a worker stack and back where necessary.

From a technical standpoint, we categorise system calls into two classes. A simple system call is either synchronous – it runs to completion without blocking – or it is asynchronous but never blocks according to the process model. If a simple system call cannot make further progress, it must block with a continuation. Conversely, an extended system call is one that has an asynchronous interface and can block in the kernel according to the process model, with an obligatory kernel-context switch. As we will see over the course of this chapter, simple system calls are more efficient to implement and offer more opportunities for optimisation.

An overview of how kernel entries and exits are carried out on x86_64, our reference platform, is listed in Table 6.1. System calls from unprivileged mode issue a mode switch via the syscall assembly instruction and return via sysret. The kernel’s entry- and exit-code sequence takes care of switching to a worker stack in software. By contrast, system calls from privileged application code are implemented as regular function calls. Simple system calls are executed directly on the user stack; only for system calls that may block must a switch to the dedicated worker stack be performed. More details on the processing of system calls are given in Section 6.3.

AtroPOS offers optional support for kernel page-table isolation (KPTI) – that is, separate sets of page tables for kernel and user space. KPTI in Linux was originally designed under the name KAISER to close side channels in conjunction with kernel ASLR [Gru+17], but it subsequently turned out to be an essential means to protect against the widely publicised Meltdown attack [Lip+18], which allows unprivileged applications to read from protected kernel memory by exploiting a catastrophic architectural flaw in a wide range of CPU families. All major operating systems have since adopted KPTI. Isolating the kernel’s page tables from those of the user application requires a page-table switch on every kernel entry and exit, including full TLB invalidations unless the TLB supports tagging.

Interrupts are not generally disabled during the execution of kernel jobs, so they may occur at arbitrary times. Consequently, kernel-job executions can be nested. Worker stacks must be large enough to accommodate the frames of numerous nested jobs. To be able to give hard guarantees, it is necessary to analytically determine the maximum stack usage of a kernel job. As a possible strategy to avoid kernel-context overflows, the kernel might temporarily disable interrupts when it cannot be guaranteed that the context has sufficient space left for yet another nested kernel job. Stack-usage analysis and overflow avoidance are outside of the scope of this thesis and are subject to future work\(^2\); in the prototype implementation of AtroPOS, the context size is determined empirically with a safety margin, and guard pages prevent silent data corruption.

The process of switching to the desired worker stack for the execution of a kernel job depends on whether the interrupted control flow is an unprivileged or a privileged user activity, or whether it is

\(^1\)The stack switch is only necessary for extended system calls but not for simple system calls; see below.
\(^2\)A static stack-usage analysis for AtroPOS is deemed not trivial but neither impossible: while there are a few instances of recursion in the kernel, they are all bounded. An example are iterations over limited-size red–black trees.
another kernel job. In x86_64 processors, optimised hardware support for interrupt handling is hard-wired for the following two situations:

- An unprivileged user control flow is interrupted; the processor performs a mode switch into kernel mode and activates a preconfigured kernel stack selected using the task-state segment [Int19].

- A privileged kernel control flow is interrupted; the processor remains in kernel mode and no stack switch is carried out in hardware.

Since the concept of a privileged user control flow is not reflected in the x86_64 architecture, there is no native way to make the processor automatically switch to the worker stack whenever a user control flow – privileged or not – gets interrupted\(^3\). Hence, switching to the worker stack is implemented by AtroPOS in software: when the processor is interrupted while in privileged mode, it saves its context on the user stack; in unprivileged mode, it pushes the context onto the dedicated interrupt-entry stack. The kernel then checks whether the interrupted control flow is a user task or a kernel job. In the former case, it switches to the core-local worker stack. In the latter case, we are already running on a worker stack; hence, the new job is executed on the same stack as the interrupted job.

### 6.2.2 Interrupt Synchronisation

For interrupt synchronisation, AtroPOS implements the two-phase prologue/epilogue scheme of PEACE [Sch94], PURE [Sch+98] and CiAO [Loh+07]. In this model, interrupt service routines are divided into two parts: a prologue and an epilogue. The prologue is non-interruptible and is intended to be short; it usually only saves the hardware state and acknowledges the external event. The prologue may optionally request the subsequent execution of an epilogue, which is interruptible and may access arbitrary portions of the operating-system state. Epilogues are processed with a lower priority than prologues; their execution is delayed until all prologues have finished on the local CPU core. For this purpose, runnable epilogues are collected in a core-local queue. The prologue/epilogue model is related to the concept of top and bottom halves (tasklets) in Linux [Rot15] and to deferred procedure calls in Windows [RSI12].

The processing of interrupt epilogues is orthogonal to the scheduling of kernel jobs; although the two interact with each other, they cover separate concerns.

### 6.2.3 Kernel Scheduler

Newly created kernel jobs that are triggered by a system call or interrupt are executed immediately. When a kernel job terminates or blocks, the kernel scheduler is activated and looks for further work to be carried out before the kernel is left. The kernel scheduler’s schedels are kernel jobs that have either yielded or that had previously blocked and have been set ready to run again. Scheduling is done in FIFO order and on a per-core basis: the CPU core that unblocks a kernel job is responsible for subsequently executing it. The scheduler manages an unbounded queue per core for this purpose. The queue is implemented as a linked list with single-producer, single-consumer (SPSC) semantics; it is based on an interrupt-safe wait-free algorithm proposed by Drescher and Schröder-Preikschat [DS15].

\(^3\)In principle, this could be achieved on x86_64 by letting privileged applications run in ring 1 or 2 – but that would in turn make system calls from privileged user code more complex and expensive because they would entail ring switches.
6.2 Implementation of the Kernel Execution Model

Figure 6.2: Unblocking a kernel job.
The job is unblocked by being inserted into the ready-queue.

Figure 6.3: Dispatching of kernel jobs from the core-local ready-queue.
The red and green arrows indicate the site where the kernel job temporarily hands control to another job and is later resumed.

As has been roughly outlined in the previous chapter in Figure 5.5b on page 91, the scheduling of unblocked kernel jobs works as follows. Only a kernel job can set another kernel job ready. In most cases, this is done by an interrupt job that gets triggered because the event another job is waiting for – for example the arrival of an Ethernet packet – has occurred. The interrupt job does its work – for example extract the packet from the Ethernet adapter's ring buffer – and appends the unblocked job to the local queue in the process, as illustrated in Figure 6.2. There is no such concept as handoff scheduling for kernel jobs. Right before the interrupt job finishes, it activates the kernel scheduler's dispatcher, which works off the ready-queue. The detailed procedure is visualised in flowchart form in Figure 6.3: the outgoing job removes the first element from the queue and appends itself to the end. It then saves its own context and performs a context switch to the dequeued job. As soon as that job finishes, it will in turn dequeue the next element and switch to it. This way, control will eventually return to the original interrupt job. Since more jobs may have been set ready in the meantime, we need to check once more whether there is further work left to do. If there is none, the interrupt-handling kernel job terminates and returns to user space.

As interrupts are allowed during the execution of a kernel job, kernel-job executions can be nested. In that case, only the outermost job – that is, the one that will eventually return control to the user application – is responsible for emptying the job queue and processing the items that may have accumulated. This behaviour is similar to that achieved by the asynchronous system trap used in BSD to force a process into the scheduler before it returns to user mode. BSD uses asynchronous system traps 'because the VAX architecture does not permit context switching from the interrupt stack' [Lef+89]; on non-VAX machines, BSD emulates them by provoking memory faults. AtroPOS is not built around the notion of such a hardware mechanism; it directly invokes the kernel scheduler with conditional function calls that are part of the regular code paths leading out of the kernel.

Figure 6.4 shows how interrupts are handled in AtroPOS. Page-table switches occur if the processor was interrupted while in unprivileged application mode. If the interrupted control flow belongs to a
Figure 6.4: Handling of an interrupt.

It is ensured that the interrupt handler is executed on a kernel stack and that only a non-nested interrupt job calls the kernel dispatcher afterwards. The violet box with double-struck vertical edges refers to Figure 6.3. Switching to a dedicated kernel page table and back is optional, depending on the KPTI configuration setting.

A user application – regardless of whether it is unprivileged or privileged – a switch to the dedicated worker stack is performed before the interrupt handler is executed. Likewise, it is only then that the kernel dispatcher is activated and the ready-queue is processed before returning from the interrupt. The privileged-call flag indicates that a privileged application was executing a simple system call on the user stack when it was interrupted, in which case the interrupt job does not activate the kernel dispatcher because the system-call stub itself will take care of working off the queue. More details on this are given in Section 6.3.2.

The sequence depicted in Figure 6.4 is entered and left with interrupts disabled. The step execute interrupt handler implements the aforementioned two-phase interrupt handling as per the prologue/epilogue model; it temporarily unblocks interrupts after the prologue and blocks them again after the epilogues have been processed.
6.3 System-Call Mechanism

System calls allow a user program to trigger the execution of a kernel job that operates on its behalf. Their implementation in AtroPOS differs from that of traditional operating systems in two ways:

1. A system call may or may not entail a processor-mode switch (and possibly an associated page-table switch), depending on whether it is issued by an unprivileged or a privileged application. This decision cannot be made statically because the privilege status of an application is dynamically changeable at run time and may vary over time.

2. A distinction is made between simple and extended system calls, which is reflected in the call semantics and the system-call stub. It also has profound ramifications for the kernel-internal control-flow model.

System-call stubs are very small and straightforward when privilege isolation is configured to be statically enforced, but they grow somewhat more complex in the case of disabled hardware isolation, and more so for dynamically configurable isolation. We first examine what a system call from unprivileged mode looks like, which constitutes the base case.

6.3.1 System Calls from Unprivileged Mode

The code path in AtroPOS that handles system calls made from an unprivileged application is identical for simple and extended system calls. It is largely equivalent in principle to the system-call implementation of Linux, with the distinction that in-kernel blocking is dealt with differently. The call arguments reside in registers as per the kernel ABI (see Table 6.2). The system-call number is also passed via register.

Figure 6.5 shows the sequence of an unprivileged system call. The kernel mode is entered with the help of the `syscall` instruction, which saves the current program counter and processor flags in `%rcx` and `%r11`, disables interrupts, switches into ring 0 and jumps to the preconfigured kernel entry point. The entry-point routine is then responsible for switching to the worker stack, saving the essential parts of the user context, and switching to the dedicated kernel page table if KPTI is enabled. Afterwards, it uses the system-call number to look up the address of the requested kernel function in the system-call table. It then executes the kernel job as explained in the next paragraph. Afterwards, the application’s page table is reinstated if necessary; the user context and the user stack are restored. Finally, control is returned into user mode to the caller via `sysret`.

The execution of the kernel job itself is depicted in Figure 6.6. The job invokes the kernel function that was retrieved from the system-call table. The kernel function may block according to the

---

<table>
<thead>
<tr>
<th>Entity</th>
<th>Register(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments</td>
<td><code>%rdi, %rsi, %rdx, %rax, %r8</code></td>
</tr>
<tr>
<td>System-call number</td>
<td><code>%r9</code></td>
</tr>
<tr>
<td>Return value</td>
<td><code>%rax</code></td>
</tr>
<tr>
<td>Saved program counter</td>
<td><code>%rcx</code></td>
</tr>
<tr>
<td>Saved processor flags</td>
<td><code>%r11</code></td>
</tr>
</tbody>
</table>

(dictated by `syscall/sysret`)

Table 6.2: AtroPOS kernel calling conventions on x86_64.

With the exception of the fourth argument, the mapping of arguments to registers is the same as for user functions according to the System V ABI [Mat+14]. This keeps the number of register moves in system-call stubs low.
6 Implementation Details

Figure 6.5: System call from unprivileged mode.

This sequence corresponds largely to the way system calls are implemented in commodity operating systems such as Linux. Crossing the dashed vertical line represents the transition between user and kernel mode and back.

Figure 6.6: Execution of a kernel job.

The kernel function may block the job during its execution; the blocking procedure is depicted in Figure 6.8. The singular red arrow indicates that the kernel job is terminated at that point and another job is resumed.

process model while it is executed, in which case the kernel job is suspended and another kernel context is activated that carries out the premature return to user space. That procedure is discussed in detail below in Section 6.3.3. Our blocked kernel job will eventually be reactivated as described in Section 6.2.3. After the kernel function finishes, we have to make a distinction: did the kernel job block during its execution? If it did not, we simply continue by processing the kernel scheduler’s job queue and subsequently returning to user space. However, if our job did block – as indicated by a Boolean flag in the kernel context – control had already been returned to the user application back then, so we cannot return to it once more. Instead, we remove the next kernel job from the queue, terminate our own job by discarding its context, and switch to the dequeued one. There must be at least one kernel job in the queue, namely the one that has unblocked us and yielded afterwards (see Section 6.2.3).

For a simple system call, checking whether the job has blocked always yields false. Since that check and its true-branch constitute the only section of the common code path that could be omitted, no specialised own entry point is provided for simple system calls. This is why the same code path is taken for both categories. Note, however, that user-space stubs for the two system-call categories differ in one detail. For an extended system call, the application must take care of explicitly saving its non-volatile registers (%rbp, %rbx and %r12-%r15 on x86_64) before the call and of restoring them afterwards. Although these registers do get spilled and restored by the kernel in principle, we cannot rely on their proper restoration in the face of blocking because a system call that blocks according to the process model returns to user space on a completely different code path. For this reason, the stub function pushes the non-volatile registers to the user stack and pops them after the
6.3 System-Call Mechanism

System call has returned. This ensures that the post-system-call register state is consistent even if the pertaining system operation has blocked and the system call has returned early.

For the sake of brevity, the flowcharts leave interrupt synchronisation unmentioned – but proper measures are in place to avoid a lost-wakeup problem where a system call's kernel job could be interrupted after it has finished working off the ready-queue. In such a scenario, the interrupt's job could unblock a third kernel job but would subsequently decline to process the queue because it would deem the interruptee responsible for that (see Figure 6.4). As a consequence, the third job would remain stuck in the ready-list indefinitely. This issue is addressed by the system-call job as follows: interrupts are suppressed before the last dequeue attempt and are delayed until the job has terminated. This way, the IRQ will not go through until the CPU core is back in user mode; it will then trigger an immediate kernel re-entry.

6.3.2 System Calls from Privileged Mode

When a privileged application invokes a system operation, there is plenty of leeway for optimisations. Not only is there no necessity for switching from user mode to kernel mode and back, but other sections of the procedure can be simplified as well. The degree to which the code can be optimised depends on the category of the system call: simple calls allow for more aggressive improvements than extended ones. Their different handling is shown in Figure 6.7.

As is visualised in Figure 6.7a, the overhead of executing a simple system call from privileged mode is kept exceedingly low. The user-space stub calls the kernel function directly, without even switching to a kernel stack first. Here it pays off to provide a dedicated stub for each system operation: since the address of the kernel function can be resolved statically at link time, the function invocation is expressed as an absolute call. By default, the only two sources of overhead are the setting and resetting of the core-local privileged-call flag, which has been explained in Section 6.2.3, and a subsequent check of whether there are jobs in the kernel scheduler’s run queue. Only if that condition applies do we have to temporarily switch to a kernel stack after all and process the run queue as depicted in Figure 6.3. This needs to happen on a dedicated kernel stack because the queue-processing routine performs context saves and switches, which makes it necessary to pretend to be a proper kernel job.

Extended system calls offer fewer optimisation opportunities, but their execution path is still streamlined compared to calls from unprivileged mode. It is mandatory that extended system calls be executed on a kernel stack because their context must be captured and moved aside in the case of blocking. The processing of an extended system call from privileged mode is shown in Figure 6.7b. As explained above, the caller's user stub is responsible for saving and restoring the set of non-volatile registers. It invokes a simplified kernel-entry function whose calling conventions are identical to those of the regular kernel entry point – with the difference that it receives as argument not a system-call number but the address of the kernel function itself. The simplified entry point switches stacks but, as the caller is already privileged, does not change privilege modes. It then executes the kernel job as normal (see Figure 6.6) but calls the function pointer instead of having to look up the kernel function in the system-call table first. Afterwards, the user stack is reactivated and control is returned to the stub, which restores the previously saved non-volatile registers.

As KPTI is only applicable when the application runs isolated, a system call from a privileged application does not entail switching to a dedicated kernel page table and back. This leads to a slight reduction in the system operation's direct costs (note that writing to the page-table base register is a serialising instruction), but it may also be noticeable in the indirect costs – especially so if the KPTI implementation makes no use of TLB tagging.
Figure 6.7: User-space stubs for simple and extended system calls from privileged mode.

Simple system calls are executed directly on the user stack; extended calls require somewhat more complex handling but are more straightforward than calls from unprivileged mode.

In summary, system calls from privileged mode are more efficient than their counterparts with respect to both their direct and their indirect costs. The primary contributors to the improvements are the omission of the mode switches into and out of the kernel, and the eschewal of toggling page tables if KPTI is configured in – but there is also a reduction of indirection such as the consultation of the system-call table.

6.3.3 Blocking a Kernel Job

The final ingredient necessary for implementing the control-flow model of ArroPOS is the blocking of a kernel job. The procedure for blocking according to the process model is outlined in Figure 6.8. Under normal circumstances, the kernel exit is executed on the same stack as the kernel entry – but when a job blocks, the responsibility for leaving the kernel is shifted to another kernel context because the current one is still holding the state of the blocked job. It is important to differentiate between two cases: is this the first time that the kernel job has blocked or has it already blocked
6.3 System-Call Mechanism

Figure 6.8: Blocking a kernel job.
If the job has not blocked before, a new kernel stack is set up and control returns to user space; otherwise, a switch to another kernel job is performed. The sequence following the dashed arrow is executed on the new kernel stack.

earlier? In the first case, control must be returned to the calling user application; in the second case, it must be handed back to the kernel job that has unblocked us, for the application itself has already resumed its execution after the first time the kernel job suspended itself.

In order to return to the user program (see also Figure 5.5a on page 91), a new kernel stack is taken from the pool of contexts and is initialised with a minimal return trampoline\(^4\). That context becomes the core’s new designated worker stack. The state of the blocked kernel job is saved on its stack and a switch is performed to the new kernel context. The return trampoline first processes the kernel-job queue and subsequently leaves the kernel; the concrete return sequence depends on whether the core is configured to execute the application in privileged or in unprivileged mode.

6.3.4 Dynamic Privilege Isolation

If privilege isolation is set to either statically enabled or statically disabled in the Atropos configuration, the system-call stubs are generated according to Section 6.3.1 or Section 6.3.2, respectively. Likewise, the decision made in Figure 6.8 which return path to take is resolved statically by the compiler. However, as soon as dynamic privilege isolation comes into play, the appropriate system-call mechanism can now change at run time. This can be resolved in two ways: either through an additional level of indirection that directs to the correct stub, or through a unified system-call stub that dynamically chooses the appropriate mechanism depending on the caller’s privilege level.

\(^4\)This mechanism is similar in principle to a scheduler-activation upcall – but in contrast to a scheduler activation, it ends up at the original call site, not at a dedicated scheduler entry point.
Implementation Details

Listing 6.1: Stub assembly code for a simple system call.

Depending on the processor’s privilege level, the stub either performs a regular system call or takes the fast path.

Inspiration for a possible indirection technique can be taken from the way the GNU libc implements system calls on i686 Linux: the Linux kernel chooses the best system-call mechanism for the CPU model at boot-up and exports the appropriate stub under the symbol __kernel_vsyscall() in the vDSO [Dry14]; the libc stores a pointer to that function in thread-local memory during initialisation. System calls are then issued through the instruction call *%gs:0x10, which retrieves the stub’s function pointer from thread-local storage. AtroPOS could use a similar technique; changing a CPU core’s isolation status would entail changing the corresponding function pointer.

As a more convenient alternative, the system-call stub can determine the privilege level the CPU core is running on, and choose the correct branch accordingly. On x86_64, this is achieved by examining the two lowermost bits of the code-segment selector register %cs, which contain the protection-ring number: a value of 3 indicates that the core is in user mode, 0 that it is in kernel mode. Since this test is very cheap, it suggests itself for x86_64 to rely on the unified system-call stub. Listing 6.1 gives an example of the resulting assembly code for the stub of the simple system call mem_map(),

```assembly
Listing 6.1: Stub assembly code for a simple system call.

Depending on the processor’s privilege level, the stub either performs a regular system call or takes the fast path.

Inspiration for a possible indirection technique can be taken from the way the GNU libc implements system calls on i686 Linux: the Linux kernel chooses the best system-call mechanism for the CPU model at boot-up and exports the appropriate stub under the symbol __kernel_vsyscall() in the vDSO [Dry14]; the libc stores a pointer to that function in thread-local memory during initialisation. System calls are then issued through the instruction call *%gs:0x10, which retrieves the stub’s function pointer from thread-local storage. AtroPOS could use a similar technique; changing a CPU core’s isolation status would entail changing the corresponding function pointer.

As a more convenient alternative, the system-call stub can determine the privilege level the CPU core is running on, and choose the correct branch accordingly. On x86_64, this is achieved by examining the two lowermost bits of the code-segment selector register %cs, which contain the protection-ring number: a value of 3 indicates that the core is in user mode, 0 that it is in kernel mode. Since this test is very cheap, it suggests itself for x86_64 to rely on the unified system-call stub. Listing 6.1 gives an example of the resulting assembly code for the stub of the simple system call mem_map(),

```
which allocates memory pages for the calling application. Being core-local data structures, both the privileged-call flag and the kernel-job queue are addressed relative to the FS segment. The method KernelScheduler::<code>processWorkersAfterPrivcall()</code> switches to a kernel stack, processes the queue and switches back to the user stack.

For extended system calls, the principal scheme is similar but the stub additionally takes care of saving and restoring the non-volatile registers. The system-call stub of the extended system call <code>claim_push_data()</code>, which implements a cross-tile push data transfer, is shown in assembly form in Listing 6.2. Here, the glue code for privileged calls is not inlined into the stub but exists in the form of an external entry point, named <code>privcall_body()</code>, through which all extended system calls from privileged mode enter and leave the kernel.

### 6.3.5 Changing a Core’s Privilege-Isolation Status

Switching forth and back between unprivileged and privileged application mode is implemented as a system call. For the local CPU core, it is a simple matter of entering the kernel through one
6 Implementation Details

entry point and leaving through the respective other. That is, if the kernel was entered by the syscall instruction, it is left via ret; if it got activated through a call to privcall_body(), the exit sequence uses sysret. To ensure that no register contents get overwritten in the process, the operation is formally expressed as an extended system call even though it does not block in the kernel. This way, the system-call stub saves and restores the appropriate register set.

To toggle the isolation status of another core, the system call sends an IPI to that core. The interrupt handler saves the relevant registers, sets up a return trampoline in a way analogous to the one described for system calls, and restores the application context.

6.3.6 Summary

The peculiar control-flow model implemented by AtroPOS bridges the gap between blocking kernel jobs and the partially asynchronous kernel interface. Applications can dynamically switch between unprivileged and privileged mode, which is reflected in the mechanism used to enter and leave the kernel. System calls from unprivileged mode work in a fashion similar to that of common operating systems such as Linux or Windows. Calls from a privileged application exploit the existing potential for optimisation – especially for simple system calls, but also for extended ones.

A complete list of all system calls currently implemented in AtroPOS, along with their respective categories, is given in Appendix A.

6.4 Event-Notification Interface

The event-notification mechanism is a central part of the interface that the AtroPOS kernel exposes to the runtime system. While its primary use case is to notify the user application of the completion of an asynchronous system call, the event queue is also used to transmit messages between elementary claims, both across tiles (with kernel support) and within a tile (without active involvement by the kernel). In this section, the requirements posed to such an interface are discussed and the concrete implementation in AtroPOS is presented.

6.4.1 Requirements and Considerations

As the primary use case of kernel events is to signal the completion of a system call, they must provide a means to convey the success or failure of the corresponding system operation to the caller. Just like in Linux, this is done with the help of a numerical status code. By convention, a value of 0 indicates success whereas a negative value is an error code such as -ENOMEM (out of memory), -EFAULT (bad memory address) or -EINVAL (invalid argument). Due to the asynchronous semantics of extended system calls, the function return value cannot be used for this purpose as the function may already return before the completion of the operation – it can only indicate an early failure. One possibility is the use of an extra integer parameter, passed by reference, that receives the status code. However, from an application programmer’s point of view, it is preferable to foster clarity by encapsulating the status in the event object itself. This way, waiting for an asynchronous system call with the help of a future can be achieved in the way shown previously in Listing 5.2 on page 89, with the status code and the future-object consolidated.

On the other hand, it can come in useful to split error reporting into two phases: a late phase whose success or failure is encoded in the event object, and an early phase indicated by the function’s
return value. This allows the kernel to catch error conditions such as invalid arguments early on and to signal them back to the caller without having to trigger the event. An obvious example where this makes sense is a situation where the caller passes an invalid event-object pointer: upon detecting the bad memory address, the kernel function returns the error code -EFAULT. With this, the system-call scheme must look as in Listing 6.3.

Another requirement is that the association of a given kernel-event pointer with its destined elementary claim must be memorised during the execution of the kernel operation so the kernel knows upon completion into which queue to push the event. This association can either be represented in the form of a separate kernel-internal key–value store or it can be stored directly in the event object itself.

It is not imperative for the event queue to provide a means for prioritising certain events over others; FIFO ordering is sufficient for our use case. The queue must have MPMC semantics: every core on the local tile may concurrently execute a kernel job or user activity that enqueues an event, and every core belonging to the elementary claim that owns the queue may concurrently fetch work from it. The kernel only acts as a producer; it will never remove elements from the queue. By contrast, the user application is able to both produce and consume events.

The trust model is asymmetrical with respect to the event queue: the user program needs to trust the kernel, but the kernel must not trust the user program. This has critical security implications as a malicious attacker must be prevented from gaining control over the kernel by exploiting bugs in the handling of the shared data structure, for example by setting up a carefully crafted dangling pointer. For this reason, the queue memory should be managed entirely by the kernel; it should not be under the user’s control.

### 6.4.2 Data Structures

With the above considerations in mind, the corresponding event-notification data structures – namely the event structures themselves and the shared event queues – are designed to fulfil the given requirements.

---

**Listing 6.3: Usage of an asynchronous system call, synchronised with the help of a future – improved variant with two-phase error reporting.**

This is the same code as in Listing 5.2, but the system call additionally returns an early status code.

Listing 6.3:

```c
future_t future;
future_init(&future);
int earlyRetVal = asyncSyscall(&future, arg1, arg2, arg3);
if (earlyRetVal != 0) {
    panic("asyncSyscall failed early");
}

int retVal = future_force(&future);
if (retVal != 0) {
    panic("asyncSyscall failed");
}
```
6 Implementation Details

Listing 6.4: Kernel-event base structure.

These two fields are used by the kernel; anything that lies behind them in memory is application-specific and opaque to the kernel.

Kernel Event

In Section 5.6.4, it was demanded that kernel-event objects be user-managed data structures that are mostly opaque to the kernel. Ideally, the kernel would need zero knowledge about their internals, but a pragmatic approach was chosen where each event object has a minimal header with a fixed layout. The declaration of this kernel-event structure is given in Listing 6.4. Event objects are allocated in the user application’s address space. They can be of an arbitrary size and can hold arbitrary data behind the `kernel_event_t` header; the kernel never accesses an event object beyond its header.

The object’s `retVal` field holds the status code of the asynchronous kernel operation. The kernel sets it to -EBUSY at the outset of the system call to indicate that the operation is ongoing. When the operation finishes (or aborts), the field is overwritten with the final status code. The `claimTag` field is used to memorise the destination claim of the kernel event. In the case of an asynchronous system call, the destination claim equals the claim that issues the call. For messages, which carry no return code and do not require the temporary memorisation of their destination claim, the `retVal` field is simply ignored and left uninitialised, and `claimTag` is set to the invalid tag value `MESSAGE_TAG` (−1) to identify the event object as a cross-tile message.

Since the kernel-event structure is under the control of the user application, the kernel must handle it with caution to avoid security loopholes. Without proper validation, the user could trick the kernel into overwriting critical data structures by passing as event pointer a crafted pointer to a kernel address, or it could illegally inject events into a different application by forging the `claimTag` field. Hence, the kernel verifies that the event object resides in the application’s portion of the address space, and it makes sure that the calling application equals the application the destination claim is associated with. Since the user program might concurrently overwrite the `claimTag` field during the execution of the system call, thus triggering a time-of-check-to-time-of-use bug, performing the latter check and pushing the event into the queue are combined into a logical transaction.

Event Queue

The fundamental question concerning the event queue is the following: should one use a bounded or an unbounded queue? An unbounded queue has the advantage that – memory exhaustion aside – capacity overrun is not an issue, and that no fixed amount of memory must be constantly kept available. On the other hand, it has the severe downside that it involves dynamic memory allocations and is generally implemented as some form of a pointered linked list, which complicates the interplay between kernel and user space. A variant of a linked list called UCQ is implemented by Akaros [Rho14]: a list of dynamically mapped memory pages that contain event arrays. Concurrent accesses to a UCQ are synchronised with the help of spinlocks.
6.4 Event-Notification Interface

ArroPOS opts for a simpler solution and makes use of a bounded queue with a fixed capacity and non-blocking synchronisation. Since every element consists only of a single pointer, the memory overhead of holding a not-too-small number of queue elements ready per elementary claim is tolerable. The strategy to prevent overruns is described in Section 6.6.

The queue is implemented as a bounded buffer with FIFO semantics that contains three fields: a pointer array, data, and two counters, putCount and getCount. A value of nullptr denotes an invalid, empty array entry; everything else is considered a valid event pointer. The buffer operates under the following fundamental principle: a producer writes an item into the array at the current write position and increments the write counter; a consumer reads from the array at the current read position and increments the read counter. Race conditions between multiple concurrent producers, or between multiple concurrent consumers, are dealt with by synchronising modifications of the put- and get-counter with the help of the atomic CAS instruction. The use of continuous counters instead of indices that wrap around when reaching the buffer’s capacity serves to avoid the ABA problem [Sto90] – but care must be taken to handle integer overflows correctly. The C++ source code of the bounded-buffer implementation is listed in Listing 6.5.

The put() method begins by pulling a local copy of the put-counter. The corresponding index into the array is computed by taking the counter modulo the buffer capacity. If the buffer slot referred to by the index is still occupied by a non-nullptr value, the buffer is currently full and the put() operation fails. Otherwise, the method carries on and computes the next put-counter value by calling increment(). It attempts to commit the new counter value via CAS, starting over if the put-counter was modified by a concurrent put(). Once the counter has been updated successfully, the produced item is written into the buffer slot. If the loop has been traversed MAX_ATTEMPTS times to no avail, put() gives up and returns, leaving the buffer in an unchanged state.

The inverse operation, get(), retrieves one previously produced item or returns nullptr if the buffer is empty. It is largely symmetrical to put(), but the check in line 29 causes the selected slot to be polled when it is detected that the buffer is currently empty but a concurrent put() is already in the process of filling the slot. That condition is fulfilled if getCount and putCount differ but the slot is still holding a nullptr value, meaning that the concurrent producer has passed line 10 but has not yet executed line 12. Once a filled slot has been found and the get-counter has been successfully incremented, the slot is overwritten with nullptr and thereby marked as free.

The helper method increment() ensures that a counter never overflows in an uncontrolled manner. Let us assume without loss of generality that the buffer’s capacity \( C = 100 \). Let us also assume that the counter has reached the maximum value representable by its data type, \( \text{uintptr}_t \), which is 18 446 744 073 709 551 615 on a 64-bit platform\(^5\) – hence, the corresponding buffer index is 15. Now, if we naively increment the counter once more, it will overflow and its new value will be 0, which corresponds to index 0 instead of the correct index 16! The computation in line 47 ensures that the counter is wrapped around correctly and ends up back in the interval \([0; C)\): in our concrete example, 16. Note that in the special case where the capacity \( C \) is a power of two, the manual wraparound is not strictly necessary because the overflow result, 0, will always be correct.

Producers and consumers act largely independently of each other. The checks in lines 6 and 24 prevent buffer over- and underruns, respectively. The atomic modification of the counter variables ensures that concurrent accesses of the same type operate on different slots. If multiple producers are active at the same time, slots are not necessarily filled in ascending order because one producer may overtake another producer. The same is true for multiple consumers trying to concurrently empty the buffer. However, as every slot bears an implicit tag that marks it as ready for consumption

\(^5\)While it is highly unlikely that a 64-bit counter will ever overflow in practice, the bounded-buffer implementation is designed to also work correctly on 32-bit platforms, where the limit of 4 294 967 295 is reached far more easily.
The buffer stores pointers to objects of type `T` and has a fixed capacity of `C` elements. It uses non-blocking synchronisation to cope with race conditions.

Listing 6.5: Generic MPMC bounded-buffer implementation in C++.
or ready for overwriting – indicated by whether its contents are valid or nullptr – it can be ruled out that a consumer reads a yet-unfilled slot or that a producer overwrites a yet-unread slot.

### 6.4.3 Putting It All Together

The code in Listing 6.5 is somewhat simplified. In reality, as can be seen in Figure 6.9, the bounded-buffer data structure is split up into two parts. The lower part, consisting of the put-counter and the data array, is managed by the kernel, allowing the kernel to insert event pointers but not to retrieve them. It is mapped into the application’s address space as a readable and writeable memory region. Since there is a 1:1 association between kernel-event queues and elementary claims, the data structure is allocated dynamically by the kernel during the `invade` system call. The get-counter constitutes the upper part of the event queue. It determines the current tail position of the bounded buffer and is needed for dequeuing. The get-counter is under the control of the application’s runtime system; there is no need to make it known to the kernel.

As the user program has write permissions on the entire data structure, it is able to directly insert additional work into the queue. This permits efficient intra-tile communication between claims of the same application without having to actively involve the kernel (see Section 5.6.2).

The kernel offers no protection against rogue user code that overwrites queue entries or corrupts the counters. However, the bounded-buffer data structure does not consist of any pointers (except for the elements themselves, which are never dereferenced by the kernel after insertion); its capacity is a constant and the modulo operation prevents out-of-bounds array accesses – consequently, the event queue cannot be exploited as a means to corrupt kernel memory. Assuming privilege isolation is active, the worst-case damage consists of dropped kernel events and user-memory corruption, both of which are restricted to the scope of the misbehaving application itself. Moreover, since the number of retries in the bounded buffer’s `put()` method is bounded, the kernel cannot be forced into an endless CAS loop by unprivileged code that thrashes the put-counter: it will eventually give up and resolve the matter by dropping the misbehaving application’s kernel event.

For all its benefits in terms of simplicity and efficiency, the bounded queue implementation suffers from one issue: it has a fixed limited capacity. Special measures have to be taken either to deal with...
6 Implementation Details

Listing 6.6: Communication interface of the AtroPOS kernel.

As system calls accept a maximum of five arguments, the push data transfer's source- and destination-buffer information is bundled together into a `push_params_t` descriptor. The `params->dest` field holds the address of the destination buffer, from which the destination tile ID can be trivially derived.

situations where the pressure on the event queue is too high, or to prevent such situations from arising in the first place. The mechanism that is chosen for AtroPOS is explained in Section 6.6.

6.5 Implementation of Inter-Tile Communication

The communication facilities AtroPOS offers for distributed applications that wish to exchange data across tile boundaries are implemented as system calls. As described in Section 5.5.2, the kernel exposes two different primitives: messages for transmitting small application objects, and push data transfers for asynchronously copying arbitrarily large memory buffers. Kernel involvement is necessary to perform adequate permission checking. This section focuses on the implementation of these operations on cache-coherent NUMA platforms such as contemporary large COTS x86_64 systems, where applications are supposed to use the communication primitives to exchange data across tile boundaries, while the operating-system kernel is still able to exploit the existing cache coherence by directly reaching into the memory regions of other tiles. In the design of the communication subsystem, the degree of coupling between kernel instances is intentionally kept moderate and the implementation is sufficiently generic that, given proper hardware support, porting AtroPOS to a non-cache-coherent tiled hardware platform is easily achievable. To provide an implementation for the invasive hardware prototype, the hardware primitives offered by the NoC adapter would have to be modified and enhanced to a certain degree; alternatively, a less efficient software-emulated variant could be built that uses only RPCs based on sys-lets.

Listing 6.6 shows the kernel's communication interface, which consists of two system-call functions: `claim_send_message()` and `claim_push_data()`.

6.5.1 Messages

Transmitting a message is implemented in the function `claim_send_message()`, which sends the given message, size bytes large, to the remote elementary claim identified by `claimTag` on tile `tileID`. The `message` parameter can point to an arbitrary user data structure that must be no
larger than a cache line and must contain a `kernel_event_t` structure as its first field at byte offset 0. The operation is a simple system call because all necessary checks and actions can be performed on the fly. Before proceeding to the actual message transfer, the function verifies that the given destination tile ID and claim tag refer to an existing claim on an existing tile, that the claim indeed belongs to the calling application and that the message does not exceed the maximum size.

One issue that comes into play is the question of where and how to buffer incoming messages on the destination tile. The message contents should be immediately accessible by the receiving application without necessitating further copying. On the other hand, management of the buffer memory should be achieved without having to consult the user application first. AtroPOS approaches this issue by defining a message-buffer data structure as part of its kernel interface. One message buffer is created per elementary claim and is mapped readable and writeable into the application's view of the address space. It consists of a fixed number of consecutive slots, each of them large enough to accommodate a single message; in other words, each slot has the size of a cache line (64 bytes on x86_64). A bitmap indicates which message slots are free (value 1) and which ones are occupied (value 0); finding a free slot is achieved with the `find first set` bit operation, which is compiled to the `tzcnt` instruction for x86_64.

The x86_64 NUMA implementation of `claim_send_message()` proceeds as follows: it retrieves the address of the destination claim's descriptor (see Figure 6.11 on page 132), which holds a reference to the associated event-manager object; that object encapsulates both the event queue and the message buffer. All these data structures are located in the memory of the destination NUMA domain. The sender searches for a zero-bit in the message buffer's bitmap and flips it to secure the corresponding slot. It then writes a flat copy of the sender's message object into the slot, marks the copy with the `MESSAGE_TAG` tag and ultimately inserts a pointer to the first byte of the slot into the receiver's event queue. Finally, if the destination claim has idle cores, a wakeup IPI is sent to one of those cores.

The message will subsequently be processed as part of the receiving claim's regular event handling. Its handler function is responsible for freeing the message object after processing it. It does so by setting the corresponding bit in the bitmap back to one. Due to the message buffer's straightforward structure, synchronising concurrent accesses – both from the kernel below and the application above – is achieved in a non-blocking manner with the help of the atomic CAS operation.

With this mechanism, message objects are sequenced into the recipient's regular application flow. Just like the hardware-accelerated remote-i(let primitive does on the invasive platform, it requires no explicit forth-and-back communication on the application level and is interrupt-free unless a sleeping core needs to be woken up. At the same time, it is less hard-wired and more generic, and poses no special hardware requirements other than cache coherence. Nevertheless, both in-memory data structures – a bitmap and a bounded buffer – would be straightforward to access and manipulate by a dedicated hardware unit. In other words, incorporating the message-sending primitive into the invasive NoC adapter is feasible.

### 6.5.2 Push Data Transfers

The system call `claim_push_data()` asynchronously copies `params->size` consecutive bytes from the sender's local address `params->src` to the remote address `params->dest`, then sends `message` (which is `msgSize` bytes large) to the claim identified by `claimTag` on the destination tile and

---

6If the processor supports the `mwait` instruction, sending that interrupt is omitted. More details on the idling mechanism can be found in Section 6.8.4.
6 Implementation Details

pushes the event pointer into the sender's local event queue. Push data transfers require more extensive permission checks than simple messages – in particular because the destination address needs to be verified.

The address-verification scheme is inherited from OctoPOS. It is based on the following program logic: an application's memory mapping is managed locally on a per-tile basis; in particular, for scalability reasons, changes to the local mapping – caused by the allocation (\texttt{mem\_map()}) or release (\texttt{mem\_unmap()}) of memory pages – are not actively propagated to other tiles. As a consequence of this scheme, the sending side of a push data transfer generally cannot decide locally whether the operation is permitted: it must ask the kernel instance on the destination tile via RPC if the address range in question belongs to the application that wants to push the data \cite{Dre+16}. To speed things up, OctoPOS manages a sender-side software cache that memorises the remote memory regions to which access has previously been granted. This way, subsequent push data transfers from the same tile to the same memory pages can do a simple local cache lookup instead of having to perform the costly check on the remote tile. The OctoPOS instance on the receiving tile maintains a registry of lookup results that are cached by other kernel instances. Whenever the memory mapping on that tile changes, invalidation messages for the permission-cache entries that are affected by the change are sent to those kernel instances \cite{Dre+16}.

Since the potential forth-and-back RPC communication causes the operation to block in the kernel, \texttt{claim\_push\_data()} is implemented as an extended system call in AtroPOS. It begins by performing the same set of checks as \texttt{claim\_send\_message()} does to ensure that the caller has the permission to send the completion message to the given remote claim. The kernel function then allocates a slot in the receiver's message buffer and copies the message into it, but does not push the associated pointer into the event queue yet. Afterwards, the permission check for the destination buffer is carried out as described above: first, a lookup is performed in the software cache; if that lookup fails, a blocking RPC to the remote kernel instance is initiated and the kernel job is suspended until the reply arrives. By the time the kernel job blocks and the system call returns early, the \texttt{message} object has already been copied out; hence, the caller may subsequently recycle the object. This makes it legal for the calling program to allocate messages on the user stack\textsuperscript{7}.

If permission for the transfer to the destination address is not granted, the message-buffer slot is released without having achieved anything\textsuperscript{8}, and the caller is notified of the failure with the help of the \texttt{retVal} field in the local event object. Otherwise, the data is copied from the local source buffer to the remote destination buffer, the message is enqueued into the receiver's event queue and the completion of the operation is signalled to the caller via the event.

6.6 Dealing with Buffer-Space Exhaustion

Message buffers and event queues both have a fixed limited capacity, so there is a risk of space exhaustion that has to be dealt with. Note that, even if AtroPOS made use of theoretically unbounded queues and buffers, it would still be susceptible to the same problem in practice – albeit to a lesser degree – because memory is never available in infinite quantities. The fundamental approach followed by AtroPOS is to shift the responsibility for dealing with back-pressure out of the kernel and into user space, back to the origins of messages and events. This makes work easier for the kernel and puts the onus on the application and its runtime system instead.

\textsuperscript{7}Not allowing this would be a questionable design decision and a sure source of hard-to-debug data-corruption issues caused by dangling pointers.

\textsuperscript{8}The prior copy to the message buffer has been useless in that case, but users that pass an illegal argument have no standing to complain about minor inefficiencies in the error-handling path.
6.6 Dealing with Buffer-Space Exhaustion

6.6.1 Message Buffers

An elementary claim’s message buffer can fill up if too many participants try to send messages, or to perform push data transfers, to that claim at the same time so the recipient is unable to keep up. As long as the buffer is full – indicated by an all-zero bitmap – a sender cannot proceed until the receiver has completed processing at least one message and has released the pertaining message-buffer slot again. In such a situation, we could block and retry in the kernel, but this would require establishing a feedback channel through which blocked send operations can be woken up once a slot becomes available again. In other words, coupling would be increased and a burden to notify blocked senders would be put on the receiver.

Instead, the sender simply aborts the system call and returns immediately to user space with the error code -EAGAIN (resource temporarily unavailable). The calling application is responsible for retrying the operation until it succeeds. To avoid deadlocks caused by two or more parties waiting for one another to empty their message buffers, the caller should yield and process kernel events off its event queue before retrying, thus ensuring that global progress is made. Ideally, the runtime system wraps the claim_send_message() and claim_push_data() system calls into functions that implicitly do the yielding in a retry loop. InvRT implements this behaviour.

6.6.2 Event Queues

The situation with respect to event queues is more complex because there are three different sources of kernel events: completion events of asynchronous system calls, work packages pushed by another elementary claim from the same tile, and incoming messages from another tile. Each of these three event sources must be able to deal with resource exhaustion, and the three have to cooperate. Again, the strategy is to detect a would-be overflow condition in advance and to abort the pertaining operation in a way and at a stage where it is still convenient for the caller to handle the situation.

As an example of an inconvenient stage, imagine an extended system call that does not detect that the event queue has filled up until after the kernel job has blocked: in such a situation, it is no longer possible to easily communicate that fact to the application.

The fundamental idea is to make creators of kernel events reserve event-queue slots in advance, with a reservation being not for a particular slot but serving as entitlement to exactly one slot. If the number of slots currently pre-booked equals the capacity of the queue, further reservations fail until a slot becomes available again. A relevant observation herein is the fact that, thanks to the message-buffer management described above, there is already an implicit reservation mechanism for incoming messages. Provided the capacity of the event queue is greater than that of the message buffer, we assign a quota to communication operations that equals the message buffer’s capacity, and leave the rest to the other two event sources. That second quota is managed with the help of an atomic counter, called reservedSlots, that is stored alongside the kernel part of the bounded-buffer data structure. The resulting two-pronged quota-management scheme is depicted in Figure 6.10.

The reservation counter is initially zero; it is incremented on reservation (by the kernel or the runtime system, depending on the operation) and decremented upon event retrieval (by the runtime system) – but only if the retrieved event is not a message (which is identified by the aforementioned MESSAGE_TAG). Asynchronous system calls make a reservation attempt right at the beginning of the kernel function. The reservation fails if the counter has already reached the quota; in that

---

9 This is another reason why the slot is reserved early on in claim_send_message(), before the kernel job potentially blocks: being notified via asynchronous event that the system call was cancelled and needs to be repeated would make for a very cumbersome programming interface.
In this example, the event queue has a capacity of eight slots, five of which are allotted to events created locally and three to incoming messages from remote tiles. The event queue is currently holding one event-object pointer and two message pointers. One extended system call has not terminated yet but has already reserved a slot for its completion event.

In the case, the kernel function returns the error code -EAGAIN. The calling user activity is supposed to retry at a later point in time; until then, it should yield. Yielding gives the user dispatcher the opportunity to consume work items from the local event queue, thus relieving the queue of pressure and freeing up reserved slots. This increases the likelihood that the subsequent reservation attempt will be successful. If the runtime system is uncooperative and refuses to decrement the counter, the application will eventually fail to make progress, but this will have no effect on other applications.

The same counter is used for the tile-internal exchange of work items between elementary claims. Remember that this exchange exploits the availability of shared memory; work packages are allocated locally by the sender and pushed into the receiving claim’s event queue. Like for an extended system call, the sender is responsible for reserving a buffer slot – however, not in the event queue of its own claim but in that of the receiver. Also, the entity that performs the reservation is not a kernel job but a user activity. Again, the caller is expected to yield if the reservation attempt proves unsuccessful – with the difference that yielding here has no immediate effect on the availability of event-queue slots because it is not in the caller’s hands to consume events off another elementary claim’s destination queue.

### 6.6.3 Summary

The communication and event-notification mechanisms in ArnoPOS are implemented in a way that any potential back-pressure will build up on the part of the creators of events, in user space. The kernel avoids being put into situations where it needs to block or, worse, to silently drop events. As event-queue slots are cheap memory-wise, consisting of a single pointer word, it is feasible to dimension the event queues with a large enough margin that the occurrence of space exhaustion is reasonably unlikely in practice under regular load.
6.7 Security Considerations

So far in this chapter, security has only been mentioned occasionally as a side node. In this section, we will collect and briefly discuss a number of considerations related to security in AtroPOS, with a particular focus on the differences to commodity operating systems.

Being a descendant of OctoPOS, AtroPOS shares certain characteristics with a single-address-space system [Mur+93] in that all applications and the operating system share a global memory mapping; it is only the presence status and the access rights that differ from application to application and between application and kernel. The delineation between kernel memory and user memory is optionally strengthened with the help of KPTI.

For the kernel, this means that it must be wary of any user-provided pointers, including event objects and source or destination buffers. Among others, a malicious application could attempt to carry out the following attack schemes:

- Pass a pointer to a kernel address or to another application’s memory region as source address to a system call – for instance, as the source buffer of claim_push_data() – in a bid to illegitimately exfiltrate confidential data.
- Pass a pointer to a critical data structure belonging to the kernel or to another application as destination address to a system call – for example, as the destination buffer of claim_push_data() or as a kernel-event pointer – in an attempt to trick the kernel into overwriting that data structure with attacker-controlled values.

In short, the kernel has to verify all user-supplied source and destination addresses – that is, all pointers that are passed as arguments to system calls, plus the corresponding length arguments – and has to make sure that they refer exclusively to memory regions legally accessible by the calling application. To further complicate the matter, a cunning would-be attacker might also seek to trigger a use-after-free vulnerability by passing a pointer to a valid memory page, then concurrently releasing that page via mem_unmap() and speculating that it will be claimed by another application – thus tricking the kernel into writing to a dangling pointer. AtroPOS nips this type of attack in the bud by temporarily marking as pinned all user pages that are involved in ongoing system calls; it rejects any attempts to unmap a pinned page. These measures are activated for unprivileged user applications; they are omitted for trusted user programs that run in privileged mode.

To prevent internal data from leaking into user space, the kernel scrubs all caller-saved registers before returning from privileged mode to user mode. When returning to a privileged, trusted application, this step is deemed unnecessary and is omitted.

In addition to the essential first-level protection mechanisms – that is, gap-less horizontal and vertical isolation – many modern operating systems implement a second line of defence that is supposed to make it considerably harder to exploit breaches of the first line. Such techniques include supervisor-mode execution and access prevention (blocking the kernel from executing or touching user memory [Cor12]) and kernel ASLR (randomised kernel address space with the user program having zero knowledge about the kernel’s internal layout [Edg13]). In its current state, AtroPOS is unable to activate these hardening mechanisms for two reasons: on the one hand, accessing user memory from privileged mode is perfectly legal for unisolated applications; on the other hand, the coupling between user space and kernel space is relatively tight and the application is dependent on knowledge about the kernel layout: for instance, remember that a simple system call issued on

\[\text{pinned}\] Note that the term pinned has a different meaning here than in Linux, where it describes a memory page that cannot be swapped out.
an unisolated CPU core directly invokes the corresponding kernel function. However, it is possible to construct a dedicated security-focused variant of AtroPOS that implements all these features provided vertical isolation is force-enabled and cannot be disabled statically or dynamically.

### 6.8 Implementation of the Invasive Runtime System

After we have made a foray into the concrete implementation of various aspects of the AtroPOS kernel, this section highlights select implementation details of InvRT, the invasive runtime system. A focus is put on user scheduling and dispatching in the scope of elementary claims. For its management of claims, the invasive runtime system builds upon the shared elementary-claim data structures exposed by the kernel that were introduced in Section 6.1.2. An overview of these data structures – both on the kernel level and on the user level – is given in Figure 6.11. The layout displayed in the lower half of the figure is predetermined by the operating system; the upper half shows what the data structures look like in InvRT. Alternative runtime systems are free to organise the information differently.

On the kernel level, the claim information is divided into two parts: the aforementioned claim descriptor, which bundles all the relevant data over which the runtime system has no immediate influence, and an event-manager component that encompasses the data structures related to kernel events and messages. These data structures, namely the event queue and the message buffer, are both readable and writeable by user space. Also included is CPU-core status information, which is relevant for the handling of halted cores (see Section 6.8.4). The runtime system generally only
6.8 Implementation of the Invasive Runtime System

Listing 6.7: Excerpt from the resource-management interface of the AtroPOS kernel.

Other functionality not listed here includes the invasion of resources on a remote tile, adding cores to and removing them from an existing elementary claim, and the destruction of a claim.

reads and never writes this status information – but it is strongly linked to the kernel side of event management, which is why it is part of the event-manager data structure. In the worst case, the write permissions on this field empower a rogue user program to sabotage the liveness of its own cores but not the liveness of cores belonging to another application.

The elementary-claim data structure stewarded by InvRT on the user level holds references to the claim descriptor, to the user scheduler and to the user part of the event-queue structure introduced in Section 6.4, which in turn holds a pointer to the corresponding kernel portion. In addition, it contains a map of cores currently looking for work, and – to avoid excessive dereferencing chains in hot code paths – a direct reference to the event manager's field where the halted cores are noted.

6.8.1 Resource Management

The following paragraphs give a brief overview of the way computing resources are managed by the runtime system with the help of the kernel's low-level invasion interface. The portion of the interface that is relevant for the construction of a new elementary claim is listed in Listing 6.7. The claim_desc_t structure is the aforementioned claim descriptor. The core_mask_t data type is a bitmap that is defined by default as uint64_t, which allows a maximum of 64 CPU cores per tile; larger core counts could be supported by setting it to the built-in type __uint128_t or by defining it as a struct similar to cpu_set_t on Linux. The claim descriptor's retreatMask field is a bitmap of cores that are slated for retreat, which is achieved cooperatively by the runtime system as described later in Section 6.8.3.

In order to create a new elementary claim with a given number of cores, the runtime system invokes the asynchronous system call claim_construct_with_cores(). This system call constructs the elementary claim, writes the corresponding claim tag into the event object's retVal field and performs an upcall into user space on exactly one of the invaded cores. The upcall executes the function specified by initFunc and passes as arguments a pointer to the claim descriptor plus an arbitrary user-specified pointer. The initialisation function bears the responsibility for
allocating and initialising the runtime system's elementary-claim data structures. Afterwards, it
heaves up the elementary claim's remaining cores – referred to as subordinate cores – by call-
ing claim_startup_subordinate_cores(), which in turn carries out upcalls on all those cores
(triggered by IPIs) to the function pointed to by startFunc, again with an optional pointer argu-
ment. Once all of the claim's cores are up and running, they are ready to enter the dispatcher and
to begin executing user code.

Additional system calls exist for invading cores on a remote tile, for modifying an existing claim by
adding or removing cores, and for retreating and destroying a claim.

6.8.2 User Scheduling

The implementation of execution-context management, stack switching and synchronisation prim-
itives in InvRT is largely identical to the one in OctoPOS. For details, the reader is referred to
Chapter 4. The main differences with respect to the handling of concurrent user control flows are
present in two areas: the integration with the AtroPOS interface, especially with its system-call and
event-notification mechanism, and the availability of modular user scheduling. InvRT provides two
different i-let schedulers that can be selected on a per-elementary-claim basis: an adaptation of the
pre-existing CIC-like work-sharing scheduler, and a new work-stealing scheduler. Both schedulers
implement the same unified interface, which makes it possible to decide for each claim individually
which scheduler suits the application program's needs the best.

The existing work-sharing scheduler emulates the behaviour of the CIC component present in the
invasive hardware prototype. It distributes i-lets to per-core FIFO run queues with MPMC semantics,
with no designated way to rebalance the load after the fact. Setting an i-let ready for another
elementary claim on the local tile is taken care of by the work-sharing scheduler itself; there is no
need to go through the event queue.

The newly introduced work-stealing scheduler makes use of a textbook work-stealing deque per
CPU core, allowing for dynamic load balancing within an elementary claim at run time. The queue's
design is the one proposed by Chase and Lev [CL05]. The scheduler follows the child-stealing
principle: newly spawned i-lets are inserted at the hot end of the local core's run queue; the
dispatcher dequeues i-lets from the hot end in LIFO order. Stealing is done by picking a random
victim core from the thief's elementary claim and attempting to snatch an i-let from the cold end of
its deque. Since the synchronisation scheme does not permit the insertion of work into a foreign
run queue, an i-let that is destined for another elementary claim on the same tile is wrapped into
an event object that is then pushed into the receiving claim's event queue.

6.8.3 Dispatching

The main dispatch loop is kept generic and is independent of the concrete scheduler implementation.
It is the central point in the runtime system where user work – in the form of i-lets and kernel
events – is picked up and dispatched. The C++ source code of the dispatch function is shown
in Listing 6.8. In each iteration, the loop attempts to perform a dispatch operation, ordered by
descending priority:

1. Process an event from the elementary claim's central kernel-event queue and run an i-let
from the core-local run queue in alternating order.

2. Repeat step 1 until both queues are empty.
6.8 Implementation of the Invasive Runtime System

```c
[[noreturn]] void Dispatcher::start(Context *currCtx) {
    iLet *iLet = currCtx->getCurrentILet(); // Pointer to i-let storage in context
    Claim *claim = ResourceManager::Inst().getClaimForOwnCore();
    KernelEventQueue &eventQueue = claim->getEventQueue();
    InvasiveScheduler &scheduler = claim->getScheduler();

    for (;;) {
        // Alternate between processing kernel events and i-lets
        bool hasWork;
        do {
            hasWork = false;
            KernelEvent *event = eventQueue.get();
            if (event != nullptr) {
                event->process();
                hasWork = true;
            } else {
                if (!scheduler.dequeue(*iLet)) {
                    Executor::runILet(*iLet, *currCtx);
                    hasWork = true;
                }
            }
        } while (hasWork);
        // Check if we need to retreat from this core
        claim->checkRetreat();
        // As a last resort, try to steal work from another core
        uintptr_t oldMonitorValue = eventQueue.getMonitorValue();
        if (scheduler.steal(*iLet)) {
            Executor::runILet(*iLet, *currCtx);
            continue;
        }
        // Poll for work a number of times
        uintptr_t oldEventCount = eventQueue.getEventCount();
        if (!scheduler.hasWork() || eventQueue.isEmpty()) {
            continue;
        }
        for (uintptr_t i = 0; i < IDLE_LOOP_COUNT; ++i) {
            if (scheduler.getMonitorValue() != oldMonitorValue
                || eventQueue.getEventCount() != oldEventCount) {
                hasWork = true;
                break;
            }
            Atomic::busy();
        } else {
            // If there is no more work for us at all, go to sleep
            if (!hasWork) {
                claim_idle(oldMonitorValue, oldEventCount);
                break;
            }
        }
    }
}
```

Listing 6.8: Main dispatch loop in InvRT.

The loop processes the available work, then tries to steal from other cores and eventually puts the core to sleep.
3. Check if the local CPU core is slated for removal from the elementary claim.
4. Try to steal an i-let from the run queues of the elementary claim’s other cores and run it.
5. Poll both the user scheduler and the event queue for work a number of times.
6. If all the previous steps have proved unsuccessful, go to sleep until work is available again.

By interlacing the processing of kernel events and i-lets, the two are effectively treated with quasi-equal priority. Every iteration of the dispatch loop starts off with an event, which is a precondition for the efficacy of the yield described in Section 6.6. The dequeue operation in line 18 and the theft in line 27 are the spots where the i-let is bound to the core’s current execution context by being moved into the context’s i-let-storage area, thus establishing a temporary 1:1 association between the two.

When a core is to be removed from an elementary claim, the kernel sets a flag in the claim descriptor’s retreatMask field. After the runtime system has processed all elements off the local core’s run queue, it polls that flag as part of its regular dispatch routine (see line 24). This cooperative approach helps to ensure that the core is not detached from the claim while there are still i-lets in its run queue. It is possible to implement a more sophisticated mechanism that facilitates earlier retreat by redistributing the core’s pending work to other cores, but that does not fall within the scope of this thesis.

The dispatcher moves on to the queues that belong to other cores of the same elementary claim and tries to steal an i-let from them, provided the user scheduler supports work stealing – if it does not, its steal() method returns false and does nothing more. If the theft fails as well, the core becomes idle until work becomes available again. It first actively polls for work a number of times before ultimately going to sleep in order to conserve energy. This procedure is described in detail in the next paragraphs.

### 6.8.4 Dealing with Idle Cores

Every CPU core has two sources of work: its own run queue for user schedulers – plus, by extension, other queues belonging to the same user scheduler if the latter supports work stealing – and the event queue of its elementary claim. While the event queue is known to the kernel because it is managed by it, the user scheduler and its internals are not. Consequently, a standardised interface is needed through which the runtime system can communicate to the kernel that new user schedulers are available so the kernel can decide to wake any idle cores. This is achieved with the help of a per-elementary-claim monitor word – a memory word that resides at a pre-agreed address and whose value conveys the availability of work in the user scheduler. A change of the monitor word’s value indicates that new work has been submitted to user scheduling. Similarly, the event queue’s put-counter is used as an indicator of new kernel events. For practical reasons – cue mwait – the monitor word resides in the header of the event-queue data structure, right next to the put-counter.

#### Waiting for Work

A dispatcher that runs out of work invokes the claim idle() system call, which puts the CPU core into an energy-saving mode. The core remains asleep in the kernel until the monitor word or the put-counter changes its value, at which point it resumes and the system call returns. Putting the processor to sleep is typically a privileged instruction that can only be executed in kernel mode.
On i686 and x86_64, the traditional way to suspend one's own core is the \texttt{hlt} instruction, which stops 'instruction execution and places the processor in a HALT state' [Int19] until it is interrupted; virtually all CPU architectures provide a similar instruction. Modern x86_64 processors offer an alternative with the \texttt{monitor/mwait} instruction pair: \texttt{monitor} advises the processor to watch a given address range for stores; \texttt{mwait} makes it enter an idle mode until either the address range is written to or an interrupt is received. The length of the range is hard-wired and can be queried via \texttt{cpuid}; 64 bytes are a common value. AtroPOS supports both sleep mechanisms but prefers the latter. With the event queue's put-counter and the user scheduler's monitor word adjacent in memory (with proper padding to prevent spurious wakeups caused by writes to nearby addresses), both can be monitored together with a single \texttt{monitor/mwait} sequence. This eliminates the need for wakeup interrupts on processors that offer pertaining support. The \texttt{claim_idle()} system call chooses the appropriate method shown in Listing 6.9: it calls \texttt{EventManager::idleMWait()} if supported, and falls back to \texttt{EventManager::idleHlt()} otherwise.

The fallback function implements a classic wait loop. To avoid the lost-wakeup problem, where a wakeup interrupt arrives right after the decision to sleep has been irrevocably made, the code section that begins with checking the wait condition (line 15) and ends with actually going to sleep (line 18) is made atomic with respect to interrupts. This atomicity is ensured by calling \texttt{CPU::dozeOff()} beforehand, which disables interrupts, and \texttt{CPU::safeSleep()}, which atomically re-enables them and executes \texttt{hlt}. If the program logic decides not to go to sleep after all, interrupts are re-enabled by \texttt{CPU::cancelSleep()}\footnote{On certain exotic architectures such as SPARC LEON3, there is no mechanism for enabling interrupts atomically with the subsequent instruction. On such architectures, the operating system resorts to a software workaround instead of blocking interrupts: the interrupt handler detects whether the processor was within the sleep function, and resets the program counter accordingly if necessary.}. When an interrupt arrives while the processor is idle, its handler may unblock a suspended kernel job that will ultimately signal a kernel event. Consequently, it is important to process any unblocked jobs immediately as they arise, which is done in line 23 within the loop. As this is accompanied by kernel-stack switches, \texttt{claim_idle()} is invoked as an extended system call to ensure that it runs on a proper kernel stack.

In order to avoid triggering an excessive number of unnecessary wakeup IPIs, the \texttt{hltStatus} field is used as an indicator of which cores are currently halted. Every core whose corresponding bit is set to one in that bitmap expects to be woken up via interrupt; if all bits of \texttt{hltStatus} are zero, no cores are halted and no interrupt is necessary. To minimise wakeup latencies, the interrupt handler for the wakeup IRQ provides only the absolute bare minimum functionality required, implemented in handwritten assembly: it acknowledges the interrupt to the Advanced Programmable Interrupt Controller (APIC) and returns immediately afterwards, without going through the regular kernel-job path described in Section 6.2.

The preferred sleep mechanism, implemented in \texttt{EventManager::idleMWait()}, makes use of \texttt{monitor} and \texttt{mwait} to conditionally put the core into idle mode. The general structure of the wait loop is identical to that of \texttt{EventManager::idleHlt()} – but instead of blocking and unblocking interrupts, the monitor is armed by \texttt{CPU::monitor()}, and no mask of halted cores needs to be maintained because no sending of wakeup interrupts is ever necessary. To prevent a lost-wakeup problem related to kernel jobs unblocked by interrupt handlers, the section in lines 39–40 is made atomic with respect to interrupts.

\textbf{Notification of New Work}

The sleep and wakeup mechanism follows the work-first principle [FLR98] in that it aims to minimise the overhead paid by regular creators of work and to shift the bulk of the burden to the respective
```cpp
bool EventManager::hasNewEvent(uintptr_t oldMonitorValue,
                                uintptr_t oldEventCount) const {
    return (this->manager.eventQueue.monitorValue != oldMonitorValue
            || this->manager.eventQueue.putCount != oldEventCount);
}

void EventManager::idleHlt(uintptr_t oldMonitorValue,
                            uintptr_t oldEventCount) {
    // Mark the current core as asleep
    core_mask_t mask = 1UL << CPU::getCPUID();
    Atomic::bitFetchOr(*this->manager.hltStatus, mask);
    KernelScheduler &kernelScheduler = KernelScheduler::Inst();
    // Loop until a new event has arrived
    CPU::dozeOff();
    while (this->hasNewEvent(oldMonitorValue, oldEventCount) == false) {
        // Put the core to sleep
        if (kernelScheduler.hasUnblockedWorkers() == false) {
            CPU::safeSleep();
        } else {
            CPU::cancelSleep();
        }
        // Process any kernel jobs that were unblocked while we were asleep
        kernelScheduler.processWorkers();
        CPU::dozeOff();
    }
    CPU::cancelSleep();
    // Mark the current core as awake
    Atomic::bitFetchAnd(*this->manager.hltStatus, ~mask);
}

void EventManager::idleMWait(uintptr_t oldMonitorValue,
                              uintptr_t oldEventCount) {
    KernelScheduler &kernelScheduler = KernelScheduler::Inst();
    // Loop until a new event has arrived
    CPU::monitor(*this->manager.eventQueue.monitorValue);
    while (this->hasNewEvent(oldMonitorValue, oldEventCount) == false) {
        // Put the core to sleep
        CPU::dozeOff();
        if (kernelScheduler.hasUnblockedWorkers() == false) {
            CPU::safeMWait();
        } else {
            hw::hal::CPU::cancelSleep();
        }
        // Process any kernel jobs that were unblocked while we were asleep
        kernelScheduler.processWorkers();
        CPU::monitor(*this->manager.eventQueue.monitorValue);
    }
}
```

Listing 6.9: Idle-loop implementation in the ArtoPOS kernel.

There are two variants of the idle function: a fallback alternative that uses hlt (or similar) to put the CPU core to sleep, and a preferred one for processors that support monitor/mwait.
### 6.8 Implementation of the Invasive Runtime System

#### Listing 6.10: Notifying the own elementary claim of the submission of an \textit{i}-let to the user scheduler.

If the processor supports \texttt{monitor/mwait}, changing the monitor value is sufficient for waking sleeping cores. Otherwise, a system call takes care of sending an IPI.

A producer of work is supposed to notify its idle peers, but the overhead for doing so should be minimal, and the overhead should be negligible in the default case where all cores are busy. This is reflected in the method \texttt{Claim::notifySelf()}, which is part of InvRT and is invoked by the user scheduler whenever an \textit{i}-let has been submitted to it. The code of that method is listed in Listing 6.10. The monitor word is changed to a new value simply by incrementing it; if it reaches the maximum, it is wrapped around to zero. There is no need to make that addition an atomic operation because the lost-update problem is immaterial in this context: if two or more control flows execute \texttt{notifySelf()} concurrently, what matters is that the monitor word does indeed change its value, but it is irrelevant by how much it is incremented. If waiting cores use \texttt{monitor/mwait}, the addition itself is sufficient to wake them up and no further action is needed. In the other case, which is determined by comparing a memory word with zero, the system call \texttt{claim_notify_self()} is issued. It sends an IPI to one of the sleeping cores – a privileged operation that has to be carried out by the kernel.

Pushing an event pointer into the kernel-event queue works in an analogous way: the accompanying increment of the put-counter implicitly wakes CPU cores that are in \texttt{mwait}; any cores that are stuck in \texttt{hlt} are reactivated via IPI. In the scenario where an event is put into the event queue of another elementary claim, a more generic system call named \texttt{claim_notify()} is used, which needs to carry out more permission checks such as making sure that the receiving claim belongs to the same application. Work-sharing user schedulers such as the CIC emulator call \texttt{claim_wake_core()} instead, which directs an IPI to a particular core.

#### 6.8.5 Events and Messages

All kernel-event types in InvRT are derived from the common abstract base class \texttt{KernelEvent}, whose definition is given in Listing 6.11. A \texttt{KernelEvent} object encapsulates the \texttt{kernel_event_t} structure from Listing 6.4, which is part of the runtime system's contract with the operating system. An overloaded cast operator is provided to absolve the user from having to perform a cumbersome explicit cast when passing a \texttt{KernelEvent} object to a system call.

A concrete subclass of \texttt{KernelEvent} may add more fields and must define a handler function. Under normal circumstances, this would be a clear-cut use case for C++ polymorphism, with \texttt{process()} as a virtual method – but the resulting object layout would be at odds with the requirements posed by the kernel interface, namely that the \texttt{kernel_event_t} structure must be placed at offset zero. This guarantee cannot be given for a polymorphic C++ object, whose first element is generally a pointer to its virtual-method table (vtable). Hence, the \texttt{KernelEvent} class builds a custom poor
class KernelEvent {
private:
    kernel_event_t event;
    void (*func)(KernelEvent *); // Poor man's vtable

protected:
    template<class T>
    KernelEvent(void (*func)(T *))
    : func(reinterpret_cast<void (*)(KernelEvent *)>(func)) { }

public:
    void process() {
        this->func(this);
    }

    intptr_t getReturnValue() const {
        return this->event.retVal;
    }

    bool isMessage() const {
        return (this->event.claimTag == MESSAGE_TAG);
    }

    operator kernel_event_t *() {
        return &this->event;
    }
};

Listing 6.11: Kernel-event base class in InvRT.

This class wraps the kernel_event_t structure (defined in Listing 6.4) into a generic base class. Futures, remote i-lets and all other user constructs that are implemented as kernel events are derived from it.

man's vtable implementation, holding an explicit function pointer that it receives from the subclass as constructor argument. The process() method, which is invoked by the dispatcher (see line 15 in Listing 6.8), calls that function pointer.

We now examine the implementation of two kernel-event subclasses that serve different purposes: one as a future that facilitates waiting for the result of an asynchronous system call, the other as a vessel for remote i-lets.

Future

As an example of a user-space synchronisation construct implemented as a kernel event, Listing 6.12 shows the Future class. It wraps a BinarySignal that represents the future's internal state. The object can be used in two mutually exclusive ways: either as a blocking future through force() or in spawning mode with the onCompletion() method. The two operations are mapped to the binary signal's wait() and onCompletion() methods, respectively. The future's handler function,
class Future : public KernelEvent {

private:

  BinarySignal sig;

public:

Future() : KernelEvent(doProcess) { }

void force() {
  this->sig.wait();
}

void onCompletion(iLet &iLet) {
  this->sig.onCompletion(&iLet);
}

private:

static void doProcess(Future *future) {
  future->sig.signalFromKernelEvent();
}

};

Listing 6.12: Future class in InvRT.

The class wraps a BinarySignal object. The user can either await the availability of the return value by using force() or specify a completion i-let with onCompletion().

doProcess(), signals the binary signal, waking up the blocked i-let or setting the completion i-let ready (if it already exists). The signal’s method signalFromKernelEvent() is a specialised and streamlined variant of signal() that carries out handoff scheduling if possible.

Remote i-let

In order to transmit an application i-let to an elementary claim on a remote tile, the i-let is wrapped into a RemoteILet object, which is then sent using the claim_send_message() system call. The definition of the RemoteILet class is shown in Listing 6.13. The object is copied into the receiving claim’s message buffer and a pointer is enqueued into the claim’s kernel-event queue as described in Section 6.5.1. Its handler method first extracts the i-let from the message and copies it into the execution context. After that, the message object is no longer needed and its message-buffer slot can be reused; it is marked as free in the bitmap. Finally, the i-let is executed in the context; the method subsequently returns into the dispatcher.

6.9 Summary

In this chapter, we have made a foray into the implementation of the AtroPOS kernel and the InvRT runtime system. We have explored in detail the kernel’s internal execution and control-flow model,
A RemoteILet object wraps an i-let and is sent to another tile as a message. The doProcess() method takes care of releasing the message-buffer slot as soon as it is no longer needed.

which shares certain characteristics with the i-let model but also manifests significant differences. In particular, kernel jobs are related more closely to fibres than to i-lets. The kernel interface exposed by AtroPOS consists of shared memory pages on the one hand and system calls on the other hand. The concrete invocation mechanism for system calls is dependent on whether the application runs isolated or unisolated from the operating system; in the latter case, optimised entry and exit paths are chosen.

The event-notification interface of AtroPOS is built around shared memory pages that contain a fixed-size bounded buffer using non-blocking synchronisation in order to asynchronously pass kernel events to the runtime system. Applications that need to communicate across tiles employ the same mechanism, combined with a specialised memory allocator, to exchange messages with one another. Communication is implemented in the form of two system calls: claim_send_message() and claim_push_data(). Overruns of the event queue or of the message buffer are avoided by design and it is ensured that any back-pressure builds up on the sending side, where it belongs.

Furthermore, select implementation aspects of InvRT, the invasive runtime system, were examined, with a focus on the runtime system’s interaction with the kernel. InvRT incorporates different modular scheduling strategies but offers a common dispatching routine. An efficient cross-layer mechanism puts idle CPU cores to sleep and wakes them up once work becomes available again. Finally, we have seen how InvRT implements user-level constructs such as futures and remote i-lets on top of kernel events as their generic foundation.
Evaluation and Discussion

In this chapter, the execution-model improvements described in Chapter 4 as well as the cross-layer operating-system design developed in Chapter 5 and implemented in Chapter 6 are appraised with respect to the objectives of this thesis. For this purpose, a number of evaluation scenarios comprising both micro-benchmarks and real-world applications are executed on a set of x86_64 multicore machines and the results are analysed.

The chapter is structured as follows. The hardware and software givens of the test environment are presented in Section 7.1. The static memory footprints of OctoPOS and AtroPOS are contrasted in Section 7.2. Afterwards, a detailed examination of select system operations is carried out in Section 7.3 with the help of micro-benchmarks. Section 7.4 provides insights into the impact of the system design on real-world applications. Finally, Section 7.5 discusses the evaluation results and summarises the findings.

7.1 Evaluation Environment and Methods

The benchmarks are executed on four COTS machines with x86_64 processors: three large multi-CPU-socket servers with the hostnames faui49big[1-3] and one workstation named fastbox with a single-socket multicore CPU clocked at a comparatively high frequency. Table 7.1 provides an overview of the evaluation systems. The three Intel-based platforms feature simultaneous multithreading [TEL95] (also referred to as hyper-threading), which provides twice as many logical CPU cores as physical ones. Overall, the test machines possess between eight and 96 logical cores. For easier distinction, the subsequent text uses not the hostnames but structured aliases that encode the manufacturer, the number of logical cores and the presence of multiple NUMA domains if applicable, for instance AMD48n or Intel8.

The executable ELF images of the operating systems and all benchmark applications are produced by the GCC 7.4.0 C/C++/Fortran compiler using the assembler and linker from the GNU Binutils in version 2.31.1. Furthermore, the AspectC++ 2.2 compiler is employed for building a set of variants of the OctoPOS and AtroPOS kernels with different static configurations.
7 Evaluation and Discussion

<table>
<thead>
<tr>
<th>Processor</th>
<th>AMD Opteron</th>
<th>Intel Xeon</th>
<th>Intel Xeon</th>
<th>Intel Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock frequency</td>
<td>2.5 GHz</td>
<td>2.2 GHz</td>
<td>2.1 GHz</td>
<td>3.5 GHz</td>
</tr>
<tr>
<td>CPU sockets</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Cores (physical/logical)</td>
<td>48/48</td>
<td>40/80</td>
<td>48/96</td>
<td>4/8</td>
</tr>
<tr>
<td>L3 cache</td>
<td>10 MiB</td>
<td>20 MiB</td>
<td>30 MiB</td>
<td>8 MiB</td>
</tr>
<tr>
<td>NUMA domains</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>RAM</td>
<td>64 GiB</td>
<td>128 GiB</td>
<td>512 GiB</td>
<td>32 GiB</td>
</tr>
</tbody>
</table>

Table 7.1: Technical characteristics of the evaluation systems.

All machines are based on the x86_64 CPU architecture. The three big boxes are cache-coherent NUMA systems with multiple processor sockets.

The runtime measurements are conducted using a software stopwatch built with the timestamp counters available in the processor hardware. A timestamp counter is a core-local register that is incremented at a constant rate equalling the processor’s designated clock frequency, which makes it suitable for measuring wall-clock times. The counters of all cores start at zero when the system is switched on or reset. Since frequency scaling is disabled, the timestamp value accurately reflects the number of clock cycles elapsed since the start of the system, and is fit for performance measurements in micro-benchmarks.

As rdtsc, the i686 and x86_64 instruction to read the current counter value, is not particularly well designed, using it correctly is somewhat tricky: due to the out-of-order architecture of modern processors, special care must be taken to prevent the CPU from reordering code around the start and end points of the measurement, which would have the potential to skew the results in arbitrary ways. Following the authoritative Intel white paper on that topic [Pao10], serialisation of the instruction stream is ensured through a careful combination of rdtsc, cpuid and the extended rdtscp instruction – with separate instruction sequences for starting and stopping the stopwatch. This mechanism works in both privileged and unprivileged mode and is suitable for benchmarking kernel and user code alike. The corresponding library functions, stopwatch_start() and stopwatch_stop(), are inlined into the calling code to avoid including function-call overheads in the measurements.

All in all, five distinct configurations of AtroPOS are compared with one another, with OctoPOS and with Linux. An overview of these configurations and of the differences between them is given in Table 7.2. They are chosen in such a way that the difference between neighbouring variants consists of a single feature in most cases. Not all benchmarks are executed in all variants; instead, a subset containing the respective relevant configurations is selected on a case-by-case basis.

7.2 Memory Footprint

In order to get a notion of the effects of shifting responsibilities from the kernel to a dedicated runtime system, we examine the memory footprint of the kernel and other system-level building

---

1 The timestamp counters in certain older processor models could not be used for wall-clock time measurements as they ticked at a variable rate that depended on the core’s current frequency and sleep state. All evaluation systems presented above implement a constant-rate counter.
Table 7.2: Operating-system variants used in the evaluation.

Generally speaking, an AtroPOS configuration variant differs from its neighbours with respect to exactly one feature each. The colours in this table correspond to the bar colours in the charts that follow.

<table>
<thead>
<tr>
<th>Runtime</th>
<th>Scheduling</th>
<th>Comm. prot.</th>
<th>Priv. isolation</th>
<th>KPTI</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>—</td>
<td>sharing</td>
<td>✗</td>
<td>✗ (static)</td>
</tr>
<tr>
<td>AtroPOS</td>
<td>bare (work-sharing)</td>
<td>InvRT</td>
<td>sharing</td>
<td>✗ (static)</td>
</tr>
<tr>
<td>AtroPOS</td>
<td>bare (work-stealing)</td>
<td>InvRT</td>
<td>stealing</td>
<td>✗ (static)</td>
</tr>
<tr>
<td>AtroPOS</td>
<td>isolated</td>
<td>InvRT</td>
<td>stealing</td>
<td>✓ (dynamic)</td>
</tr>
<tr>
<td>AtroPOS</td>
<td>Cilk Plus</td>
<td>—</td>
<td>stealing</td>
<td>✓ (dynamic)</td>
</tr>
<tr>
<td>Linux</td>
<td>—</td>
<td>—</td>
<td>steal</td>
<td>✓ (dynamic)</td>
</tr>
</tbody>
</table>

Figure 7.1: Code-size comparison.

The graph depicts the text-section sizes of OctoPOS and AtroPOS on x86_64. See also Table C.1 on page 187.

blocks, once for OctoPOS and once for AtroPOS plus InvRT. The comparison, shown in Figure 7.1, restricts itself to the text sections, that is, to the compiled machine code contained in the ELF images. On the operating-system side, it distinguishes between the actual kernel and the collection of system-call stubs (present in AtroPOS only). On top sit a trimmed-down libc and a malloc library developed by Gabor Drescher [Dre20] that provides heap-allocation functionality, including optional support for garbage collection (which is used by X10 applications). The last component is the InvRT runtime system.

Both systems are configured without support for agent-based resource bargaining and with no Ethernet stack. Compared to OctoPOS, the AtroPOS kernel is 37.4 KiB smaller, which corresponds to an atrophy by 10%. On the other hand, the InvRT runtime library weighs twice that amount, so there is an overall increase in size. While the C library remains virtually unchanged, the memory-management library shrinks by 8.8 KiB or 15%; this is conditioned by the fact that the former version contains a considerable amount of OctoPOS kernel code that was aggressively inlined by the compiler, whereas with AtroPOS the kernel code is strictly segregated and cannot be inlined into user-space code.

The size of the special kernel-provided text section that is mapped into user space and contains the system-call stubs of AtroPOS (plus a small set of system functions that are either callable directly from user space or serve as entry points for upcalls) depends on the privilege-isolation mode configured for the kernel. With dynamic isolation enabled, that section is 7.2 KiB large as depicted in Figure 7.1. If isolation is statically off, the stubs take up 5.9 KiB; if it is statically on,
they occupy only 3.7 KiB. These differences amount to less than one memory page, and the text section of the kernel itself is virtually identical in size in all three variations.

In summary, the combined code of AtroPOS and InvRT is larger than that of OctoPOS, but the difference is in no way significant. At a total extent of little over 600 KiB, the basic system software is very lightweight and easily fits into the processor's last-level cache.

### 7.3 Micro-Benchmarks

Up next is a series of micro-benchmarks, whose purpose is to show the minimal overheads of select system operations under controlled conditions and assuming a warm system state (caches, TLBs, branch predictors et cetera). Each measurement is executed 100 000 times in a loop; the values are collected in an array in RAM. After the measurements have finished, the benchmark application sends the bundled results to a host computer in a defined textual format via serial interface. This way, the transmission does not interfere with the measurements. Only the results of the last 1000 loop iterations are transmitted; the previous iterations merely serve the purpose of warming up the system and are discarded.

The execution-time measurements are occasionally affected by the processor's system-management interrupts, which are a built-in hardware mechanism for the sporadic execution of firmware programs with high privileges and priority, independently from the operating system. The occurrence of such interrupts cannot be prevented by software. It is functionally transparent to all software layers but can incur noticeable temporal delays in the range of tens or even hundreds of thousands of clock cycles: in the micro-benchmarks, this can produce outliers several orders of magnitude larger than the actual execution times. In the following micro-benchmarks, measured times that exceed a threshold of 31 000 cycles – an empirically determined value – are considered such outliers and are filtered out.

In order to retrieve meaningful numbers, the constant overhead incurred by the measurement itself is subtracted from the times measured. That measurement overhead is determined through a micro-benchmark that starts and stops the stopwatch in immediate succession on core 0; its results on the different machines are listed in Table 7.3. Two evident observations can be made:

1. The overheads vary significantly between different processor manufacturers and models.

2. Although the recommendations given in the Intel white paper [Pao10] are followed, there is still a non-zero variance in the measurement overheads on three out of four machines. This is especially glaring on Intel8, where a considerable portion of the values measured is over twice as large as the rest. As a result, the standard deviation in the low-cost micro-benchmarks is noticeably larger on Intel8 than on the other machines.

Table 7.3: Measurement overheads on the evaluation systems.

<table>
<thead>
<tr>
<th>Measurement overhead (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD48n</td>
</tr>
<tr>
<td>79 (60%) / 80 (29%) / 81 (10%)</td>
</tr>
<tr>
<td>Intel80n</td>
</tr>
<tr>
<td>48 (25%) / 52 (75%)</td>
</tr>
<tr>
<td>Intel96n</td>
</tr>
<tr>
<td>32 (100%)</td>
</tr>
<tr>
<td>Intel8</td>
</tr>
<tr>
<td>24 (79%) / 53 (21%)</td>
</tr>
</tbody>
</table>

The results are shown along with their relative frequencies. Upward outliers on AMD48n that occur less than 0.5% of the time are omitted. The variation is reasonably small on all machines except Intel8.
In all micro-benchmarks that follow, the respective evaluation machine's minimal measurement overhead (highlighted yellow in Table 7.3) is subtracted from the times measured; afterwards, their arithmetic mean is computed. These mean values are displayed mostly in the form of bar charts, with error bars representing one standard deviation. The variance is generally low, signified by small to miniscule error bars. Most micro-benchmarks are executed on all four machines, but for brevity not all graphs are necessarily shown for all machines. In particular, in cases where several machines behave very similarly – which is often the case for Intel80n and Intel96n – only one representative is picked. A complete collection of all aggregated measurement results is provided in Appendix C in tabular form.

In the micro-benchmarks, the configuration constant IDLE_LOOP_COUNT in InvRT (see Listing 6.8 on page 135) is set to a very high value in order to prevent the processor from going to sleep. This way, the latencies measured are not distorted by the additional overhead that would be required to wake a core from its sleep state.

### 7.3.1 Scheduling Operations on the User Level

This micro-benchmark determines a number of key figures with respect to the processing of i-lets. It examines the efficiency of the user scheduler and dispatcher, and also exposes architectural properties of the hardware. The system software is evaluated in three separate configurations: on one side OctoPOS with its work-sharing scheduler; on the other side AtroPOS combined with a work-sharing and a work-stealing variant of InvRT. Details about these three configurations can be retrieved from Table 7.2. The micro-benchmark aims to answer the following questions:

- **Dispatch latency.** How long does it take from the time an i-let execution finishes until the next i-let taken from the run queue becomes active?
- **Infect overhead.** What is the cost of scheduling an i-let for execution?
- **Infect latency.** How much time does it take for an i-let that is submitted to the scheduler to become active on the destination core?
- **Steal latency.** How long does it take to steal an i-let from the run queue of another core?

The two latter items are further subdivided into core-local (scheduling work for one’s own CPU core) and inter-core operations (scheduling work for another core on the same tile). The benchmark enforces this division by staking two elementary claims with one core each: infecting one’s own claim causes the i-let to be scheduled from core 0 to the same core; infecting the other claim makes it go from core 0 to core 1. All other CPU cores in the system are unassociated and idle.

Figure 7.2 shows the measurement results for the core-local operations. Graphs are displayed for AMD48n, Intel96n and Intel8; Intel80n is left out because the two large Intel machines expose relatively similar behaviour. The values for all machines are aggregated in Table C.2 on page 188. To examine the potential effects of hyper-threading on the Intel systems, the benchmark is also run with the second elementary claim on core 2 instead of core 1, thus establishing a set-up where the producer and the consumer of i-lets are not hyper-thread siblings. That variant is sometimes slightly slower and sometimes slightly faster than the former, but never by a significant margin. Hence, those numbers are not discussed further here.

As described in Section 6.8.2, scheduling work for another core behaves differently depending on the choice of scheduler. With the work-sharing scheduler, an i-let is inserted directly into the ready-queue of the destination core; with the work-stealing scheduler, pushing an item into another
Figure 7.2: Overheads and latencies of core-local scheduling and dispatching operations.

The times are measured on InvRT with the two different schedulers, and also on OctoPOS for comparison. See also Table C.2 on page 188.
core’s deque is illegal, so the i-let is wrapped into a kernel-event object and travels through the targeted claim’s event queue. The resulting overheads and latencies are given in Figure 7.3.

### Dispatch Latency

The work-sharing scheduler code of OctoPOS and InvRT is nearly identical. The sole distinction between the two is the dispatch loop, which is slightly more complex in InvRT because it additionally processes kernel events – see Listing 6.8 on page 135. A minor difference in the compiled code stems from the fact that the OctoPOS kernel is built without support for SSE (SIMD and other extended instructions) to prevent clobbering of the SIMD floating-point register state by the operating system, whereas InvRT, being a user-space library, is built with SSE enabled. Thus, InvRT transfers the i-let from the run queue into the execution context using two 128-bit `movdqa/movups` memory-access instruction pairs while OctoPOS relies on a regular cascade of eight 64-bit `mov` instructions.

On AMD48n, the dispatch latency is extremely low with OctoPOS and slightly higher with InvRT. On the three Intel machines, the situation is exactly the other way round, which hints at superior branch prediction or a more efficient SSE implementation in these processors, but the baseline latency is noticeably higher. The work-stealing scheduler has a significantly worse dispatch latency on all machines, which is owed to the fact that the `popBottom()` operation of the work-stealing queue requires a full memory barrier with the expensive `mfence` instruction – see Section 3.2.2 – whereas the MPSC bounded buffer used by the work-sharing scheduler needs only a cheaper read barrier (`lfence`) and its `get()` operation is otherwise unsynchronised.

### Self-Infect Overhead

The i-let-submission code in InvRT’s work-sharing scheduler, too, is slightly more complex compared to that in OctoPOS due to the modular scheduler structure and the addition of the monitor word described in Section 6.8.4. This leads to roughly comparable overheads with a certain variation. It can be observed that the work-sharing `infect` operation, which entails the execution of several atomic CAS instructions in short succession, is vastly more expensive than a dispatch – so the producer of work pays a higher price than the consumer. Moreover, that cost varies considerably from machine to machine: from an average of 186 cycles for OctoPOS on AMD48n to 865 cycles for InvRT on Intel96n. Executing CAS instructions appears to be very costly on the Intel processors; since the overhead increases with a larger number of cores per CPU die, it hints at a possible scalability bottleneck rooted in the hardware. The AMD processor seems to be more benign in this respect.

By contrast, pushing an i-let into the local work-stealing queue is an exceedingly cheap operation across the board. This is by no means surprising because the data structure is optimised for frequent insertions and removals at its bottom: the `pushBottom()` operation requires no synchronisation except for a single write barrier (`sfence`).

### Self-Infect Latency

The self-infect latency is measured as follows: an i-let starts the stopwatch, infects its own elementary claim with a new i-let and returns immediately; that new i-let then stops the stopwatch. This way, the scheduler’s end-to-end latency is determined. Intuitively, this latency is roughly equal to the sum of the infect overhead and the dispatch latency, which is confirmed by the measurement results.
Figure 7.3: Overheads and latencies of scheduling work for another claim.

Scheduling work for another claim, and thus for another CPU core, is more expensive than for one’s own core. See also Table C.3 on page 189.
7.3 Micro-Benchmarks

Infect Overhead

The price paid by the producer of an i-let for making the i-let available to another core is considerably higher than to its own core because it entails the inter-core exchange of cache lines. For both work-sharing scheduler implementations, the infect overhead is roughly the same on the three big server machines and only about half as high on Intel8. This is likely caused by suboptimal scalability on the part of the cache-coherence protocol, even though the cache lines involved are only bounced forth and back between two cores and there is no interference by the other cores. The InvRT scheduler's overhead is either the same or marginally larger than that of the OctoPOS scheduler.

With the work-stealing scheduler of InvRT, an i-let is submitted to another elementary claim by being wrapped into an event object, which is then inserted into the claim's event queue. This procedure is costlier on AMD48n than using the work-sharing scheduler – but it is substantially cheaper on the three Intel machines, especially on the two server systems. Evidently, the simplicity of the event-queue data structure compared to the higher complexity of the work-sharing scheduler's bounded buffer outweighs the extra cost of wrapping the i-let on the Intel machines but not on the AMD system. As a deduction from the observations made so far in this chapter, executing several CAS instructions in short succession should be avoided on Intel processors as a general rule.

Infect Latency

The infect latency largely follows the trends observed for the overhead. The added complexity in InvRT is primarily paid for by the consumer of work, which makes the difference compared to OctoPOS somewhat more pronounced. Again, it depends on the processor model whether work stealing involves lower latencies than work sharing or whether it does not.

Steal Latency

Measuring the steal latency is somewhat tricky because of the randomised nature of work-stealing scheduling. It is achieved as follows: an elementary claim is staked that comprises CPU cores 0 and 1. Core 0 repeatedly spawns i-lets and submits them to the user scheduler; core 1 steals them from the queue of core 0. The stopwatch is started as the last action of the current i-let and stopped as the first action of the next i-let. This is analogous to how the dispatch latency is measured, but the work is taken from a foreign queue and there is interference between the CPU cores.

The results for the four different hardware systems are displayed in Figure 7.4 in the form of violin plots. That representation is chosen because there is a considerable variance. It strikes that the distribution is unusually wide on AMD48n, while it is compact with few low-end outliers on Intel8 and bimodal with low-end outliers on the two large Intel machines.

Compared to the work-stealing scheduler's previously analysed core-local dispatch latency, the steal latency is higher by a factor of five to ten because it requires inter-core communication on the cache-coherence-protocol level. The end-to-end latency of stealing is not measured explicitly, but a very rough estimate can be obtained by adding up the self-infect overhead (the time it takes the producer to push an i-let into its run queue) and the steal latency (the time it takes the consumer to steal it from the producer's queue). With this heuristic, the estimated mean end-to-end-latency is approximately 1200 clock cycles on AMD48n, 700 on Intel80n, 800 on Intel96n and 550 on Intel8.

Looking at Figure 7.3, it can be inferred that the end-to-end latency of work stealing is generally...
7 Evaluation and Discussion

![Figure 7.4: Steal latencies of the work-stealing scheduler.](image)
The boxes and whiskers represent the quartiles and the outlines represent the frequency densities. The white dots signify the mean values.

both lower than that of work sharing and lower than that of pushing an i-let to another claim via the event queue.

Discussion

The work-sharing i-let scheduler of OctoPOS and InvRT are comparable in their overheads and latencies. Although the code of the InvRT version is slightly more elaborate due to its incorporation into a generic scheduling infrastructure, the added complexity has only minor repercussions on the efficiency of the implementation. The work-stealing scheduler is generally more efficient than its work-sharing counterpart even though it makes dispatching costlier. These efficiency gains are especially pronounced on the two Intel servers, where the bounded-buffer implementation of the run queue is conspicuously slow and the work-stealing code performs particularly well. Still, micro-benchmarks alone are not sufficient for predicting the performance of real-world applications, where much depends on the applications’ concrete concurrency patterns.

As a side note, it is not unsurprising how vastly differently the various processor families behave. For the work-sharing infect operation, which atomically modifies only non-shared memory locations in the benchmark, the AMD Opteron is faster by a factor of more than four than the biggest Intel Xeon processor tested. On the other hand, the Intel CPUs win over AMD by a considerable margin when it comes to inter-core communication within the processor die, both in terms of sharing work with another core and in terms of stealing from it.

7.3.2 Synchronisation Primitives

The next set of micro-benchmarks covers the runtime system’s synchronisation primitives that have been presented in Chapter 4. The goal is to assess two aspects: on the one hand the differences between the various signal data structures; on the other hand the possible benefits of handoff scheduling. For this purpose, the latencies and overheads of the relevant synchronisation operations are measured. The measurements are conducted on ArroPOS with InvRT and its work-stealing user scheduler.
7.3 Micro-Benchmarks

These are the times it takes for `wait()` to decide to block, save the context of the current i-let execution, switch to a new stack and start executing the next i-let. See also Table C.4 on page 189.

The comparison covers four types of synchronisation data structures: the pre-existing blocking simple and spawning infect signal, and the new unified counting and binary signal. The following metrics are incorporated in the juxtaposition:

- **Blocking latency.** When an i-let blocks, how much time goes by from the invocation of `wait()` until the next ready-to-run i-let becomes active?

- **Activation latency.** How long does it take for a signal to unblock or spawn an i-let until that i-let becomes active on its destination core? This is arguably the most relevant metric.

- **Signal overhead.** What is the cost of the `signal()` operation for its caller?

### Blocking Latency

The first set of measurements concerns the `wait()` operation, which is offered by all signal types except the infect signal. We differentiate between the method's fast path and its slow path as introduced in Section 4.2.1. The fast path is taken if the signal is already in the signalled state, in which case `wait()` does not need to block the calling i-let and returns immediately. The overhead of the fast path is negligible: about four clock cycles on all machines and for all signal types.

The execution of the slow path puts the calling i-let to sleep, which entails a context switch and the dispatching of a new i-let from the scheduler. The corresponding latencies are plotted in Figure 7.5. Regardless of the signal type and the processor architecture, the blocking latency amounts to approximately 210–230 clock cycles, with no statistically significant deviations. Roughly half of that time is spent dispatching the next i-let (confer Figure 7.2); the other half is consumed by the preceding steps of switching contexts and atomically changing the signal's state.

### Activation Latencies

The `wakeup latency` signifies the time that goes by from a call to `signal()` until the unblocked i-let becomes active; its equivalent for a spawning signal is the `spawn latency`. We subsume the two under the hypernym activation latency, which we examine with a two-dimensional parametrisation: locality (same core and separate cores) and handoff (disabled and enabled).
The locality dimension is split up into two figures; the results for unblocking or spawning work on the same core that calls the `signal` operation are visualised in Figure 7.6. Since the three Intel machines behave very similarly, only the charts for `AMD48n` and `Intel96n` are shown here. All latencies are well below 200 clock cycles. The bars in the left half of the plot concern the regular `signal()` method; the right half refers to handoff scheduling with `signalAndExit()`. It can be seen that, despite its greater genericness, the unified counting signal’s wakeup and spawn latencies are nearly identical to those of the respective specialised data structures. The binary signal induces lower latencies than the counting one by a constant margin of approximately 30 cycles on the AMD hardware and a mere 10 cycles on Intel. It is also evident that spawning a new i-let is more efficient than resuming a blocked one because it performs no context restoration: consequently, the binary signal’s spawn latency equals the work-stealing scheduler’s self-infect latency\(^2\) (confer Figure 7.2).

Circumventing the user scheduler with a direct handoff from the signalling i-let to the unblocked or spawned i-let brings down the latency by a factor of approximately three for unblocking and four

\(^2\)The observant reader may notice that the binary signal’s spawn latencies actually appear to be slightly lower than the self-infect latencies displayed in Figure 7.2, which goes against intuition. This deviation is likely explained by measurement tolerances and by minimal differences in the experimental set-ups of the two micro-benchmarks that trigger minor microarchitectural effects, for example the machine-code layout.
7.3 Micro-Benchmarks

Figure 7.7: Activation latencies for another core.

These measures are defined as in Figure 7.6, but the unblocked or spawned i-let is executed on another core of the same tile. Consequently, a handoff is attempted but does not succeed. See also Table C.6 on page 191.

for spawning. The handoff is especially efficient with the binary signal, where it involves no atomic read–modify–write operations and no memory barriers. Note that with the work-sharing scheduler, whose self-infect latencies are higher, the expected relative efficiency gain would be even larger.

Figure 7.7 shows the measurement results for the scenario where the elementary claim for which the wakeup/spawn i-let is destined differs from the signalling i-let’s claim. The i-let is dispatched to another CPU core, which is why the activation latency is largely dominated by the infect latency. Depending on the machine, it takes between 1000 and 3000 clock cycles, with the AMD machine trailing the Intel systems. In blocking mode, the simple signal and the unified counting signal fare equally. In spawning mode, the specialised infect signal is slightly less expensive (in the single-digit percent range) than its unified counterpart. The crucial difference between these two lies in the initialisation of the signal’s i-let field, which happens just before the measurement interval but still has an effect: the unified signal variant uses an atomic swap, which drains the store buffer and subsequently causes more work for the cache-coherence mechanism. In both blocking and spawning mode, the binary signal is around 10% faster on the AMD machine but hardly so on the Intel boxes, hinting at the possibility that an atomic swap is cheaper than a full CAS with AMD’s cache-coherence protocol while both cost the same in the Intel implementation.

The signalAndExit() operation cannot perform a handoff from one i-let to another unless both are slated for the same elementary claim, which is not the case in this scenario. This is why the
right-hand side of Figure 7.7 is essentially a mirror image of the left: the implementation falls back to a normal infect, which does not affect the latency in comparison to signal().

**Signal Overhead**

The cost paid by callers of the signal() method is depicted in Figure 7.8. In the benchmark, the wakeup or spawn i-let is scheduled to the signaller’s CPU core, so all memory accesses to the data structure come from the same core. As the behaviour of signal() differs based on the signal’s state – including the current value \( v \) of its internal counter – we distinguish between three cases. In all three cases, the counter is decremented, but additional steps may be prompted:

- \( v > 1 \): no further action is taken.
- \( v = 1 \) but \( \text{wait()} \) or \( \text{onCompletion()} \) has not yet been called: no further action is taken.
- \( v = 1 \) and a wakeup or completion i-let is already there: that i-let is scheduled.

In the binary signal, the counter value is implicit: \( v = 1 \) means that the binary signal is in the INITIAL or waiting state; \( v = 0 \) represents the SIGNALED state; \( v > 1 \) is not possible.
7.3 Micro-Benchmarks

Simple blocking signal  In the simple signal, the counter and the reference to the blocked context are mangled together into one memory word and all state changes are carried out via CAS. Hence, calls to signal() that do not unblock a sleeping i-let – either because $v > 1$ or because nobody is waiting yet – take an equal amount of time. If the fast path is taken and the i-let gets unblocked, the expectation is that we additionally have to pay the self-infect overhead (see Figure 7.2) – but the overall cost turns out to be lower than that sum, especially so on the Intel machines. This behaviour is arguably caused by the fact that the write barrier within the work-stealing queue’s pushBottom() operation is executed very shortly after the atomic read-modify-write instruction that updates the signal’s internal state; hence, the store buffer is flushed twice but is almost empty the second time, which makes the sfence faster than if it were executed in isolation.

Infect signal  The implementation of the infect signal operates very similarly to that of the simple signal, which explains the near-identical overheads. As the infect signal’s interface does not allow initialisation without setting the i-let field, invoking signal() with $v = 1$ always entails an unconditional spawn, hence the gap in the graph.

Unified counting signal  The unified counting signal manages the counter and the i-let pointer separately, which leads to partially different temporal behaviour. In the first case, signal() performs only an atomic decrement on $v$ and returns; this is equally fast as for the simple signal’s CAS. In the second case, the state is additionally switched to signalled using an atomic swap instruction, causing the overhead to be roughly doubled3. In the third case, signal() atomically decrements the counter and then takes the fast path as shown in Listings 4.3 and 4.4: it non-atomically sets the state to signalled and infects the destination claim with the i-let; this is not the same instruction sequence as for the simple signal but exhibits almost identical temporal behaviour.

Wait-free binary signal  As explained above, the binary signal supports only the pseudo-counter values $v \in \{0, 1\}$. If there is no i-let to wake up, signal() takes the slow path, atomically exchanging the state variable with SIGNALLED and returning. This proves to be as fast or marginally faster than the simple signal’s equivalent operation. If an i-let needs to be unblocked, the fast path is taken, which non-atomically sets the state to SIGNALLED and submits the i-let to the user scheduler. Curiously, the mean overhead of this path is slightly lower than the mean self-infect overhead even though it performs minimally more work, but this can arguably be attributed to measurement tolerances. Overall, the binary signal is more efficient than the counting simple signal and even more so than its unified counting counterpart.

Discussion  

A number of conclusions can be drawn from these results. They are discussed here by way of juxtaposing competing concepts. The recommendations derived from the discussion are taken into account in the implementation of InvRT and of the MPI and X10 runtime libraries, which are used by the application benchmarks that follow in Section 7.4.

3The slow path also contains a case differentiation for conditionally setting the wakeup i-let ready if signal() gets overtaken by a concurrent wait(). Since the micro-benchmark is constructed in a way that no such concurrency pattern can occur, that case is never triggered.
7 Evaluation and Discussion

**Spawning versus blocking**  Conditionally spawning a new i-let from a signal is more efficient than blocking and waking an existing i-let execution. Not only are the activation latencies lower by a significant margin, but the blocking step is dispensed with entirely. However, if the state that needs to be passed to the spawned i-let does not fit into the i-let’s two argument words, the explicit allocation, saving and retrieval of that state may eat up the cycles gained. Hence, it must be decided on a case-by-case basis which of the two mechanisms is more appropriate.

**Unified signal versus specialised blocking and spawning signals**  The unified counting signal that supports both the blocking and the spawning mode of operation is virtually identical to the simple blocking signal in terms of latencies; the only noticeable difference is in the signal overhead when \( v = 1 \) and the slow path of \( \text{signal()} \) is taken, which contains two atomic read–modify–write instructions instead of one. In spawning mode, the unified signal’s temporal behaviour is the same as that of the infect signal, except for a mild surplus in the cross-core activation latency that is caused by one additional atomic memory access. Hence, it can be established that the unified signal is able to combine the advantages of both signal modes without entailing any noteworthy disadvantages. As an added bonus, its use is guaranteed to be wait-free as long as the number of activities to wait for is predetermined as described in Section 4.2.2.

**Binary signal versus counting signal**  The binary signal is more efficient than the counting signal in almost all respects. The magnitude of the achievable improvement depends on the concrete use case and on the processor, memory and cache architecture. It is especially pronounced when the binary signal is used in spawning mode with handoff scheduling. The implementation of all operations is fully wait-free. This makes the binary signal the preferred choice for the notification of single events; in InvRT, it serves as the basis upon which futures are built.

**Handoff scheduling versus regular wakeup/spawn**  Handoff scheduling with \( \text{signalAndExit()} \) brings down the activation latency by a factor of three to four. If no handoff can be performed, the operation is no less efficient than \( \text{signal()} \). Consequently, the recommendation is to prefer \( \text{signalAndExit()} \) over \( \text{signal()} \) wherever applicable unless there is a striking reason not to bypass the user scheduler.

7.3.3 System Calls

We now turn from InvRT to AtroPOS and analyse the overheads induced by a system call. To determine these overheads, an empty kernel function is invoked with privilege isolation enabled and disabled – both statically and dynamically – using the simple and the extended invocation mechanism. The measurement results are plotted in Figure 7.9 in the form of overlaid bars for two metrics: the entry latency and the system-call overhead. The entry latency is measured by starting the stopwatch immediately before calling the system-call stub and stopping it within the body of the (otherwise empty) kernel function. The system-call overhead constitutes the direct cost of executing an empty system call; to determine it, the benchmark application starts the stopwatch, makes a system call with a completely empty kernel function, and stops the stopwatch right after control has returned from the stub. The entry latency constitutes approximately half of the system-call overhead across the bench, so it is sufficient to examine the latter in detail.
7.3 Micro-Benchmarks

The overheads are measured separately for simple and extended system calls, with different isolation settings. See also Table C.8 on page 193.

Enabled versus Disabled Privilege Isolation

When privilege isolation is enabled, both simple and extended system calls take between 100 and 125 clock cycles on all processor models. The extra work carried out by the extended stub – spilling the non-volatile registers to the user stack before the mode switch and restoring them afterwards – costs extremely little extra time. The system-call overhead is so low thanks to the use of the \texttt{syscall/sysret} instruction pair for entering and leaving the kernel. For comparison: entering the kernel in the legacy i686 way via trap – by executing \texttt{int $0x80} – makes the overhead of an empty system call surge to approximately 400–500 clock cycles on the AMD machine and 900–1000 clock cycles on Intel.

Simple system calls from privileged mode, being implemented as direct function invocations, cost next to nothing at a mere 11 clock cycles or less\footnote{The results on \textit{Intel} are at 16–17 clock cycles, but this value is very likely skewed upward by the aforementioned unstable measurement latency on that machine.}. At 32–40 clock cycles, extended system calls – which are register-indirect function invocations with register spilling and stack switching – are still very cheap, being faster than their mode-switching equivalents by a factor of three to four.

\textbf{Figure 7.9: System-call overheads.}

\textit{The overheads are measured separately for simple and extended system calls, with different isolation settings. See also Table C.8 on page 193.}
Static versus Dynamic Isolation

The micro-benchmark shows no noticeable difference in the system-call overhead between statically configured and dynamically configured privilege isolation. Essentially, the cost of the additional run-time mode check included in the system-call stub is so low that it disappears in the measurement noise. As examined in Section 7.2, there is a mild increase in the size of the text section because the kernel and the stubs need to provide both entry paths, but the overall difference in memory consumption amounts to less than a single page. For this reason, subsequent direct comparisons related to vertical isolation are done with a dynamically configured AtroPOS kernel image (see Table 7.2).

Discussion

In summary, system calls in AtroPOS are already efficient by default, and they become almost free of charge when the application is executed in privileged mode. The distinction between simple and extended system calls pays off in the latter case but not in the former.

For comparison, the getpid() system call on Linux — explicitly invoked using the syscall instruction, not the vDSO-accelerated variant that avoids switching into kernel mode — costs about 300 cycles on AMD48n. On the Intel machines, where Linux force-enables KPTI to protect itself against the Meltdown attack, getpid() takes between 620 and 820 cycles\(^5\). With an experimental variant of AtroPOS that includes an unoptimised KPTI implementation, a system call takes 700–1300 clock cycles — but this comparison is skewed in favour of Linux because AtroPOS currently does not use TLB tagging; hence, a costly full invalidation of the core-local TLB is triggered whenever the AtroPOS kernel is entered or left.

Note that the benchmark provides a notion of the minimal overheads under isolated conditions. The actual direct and indirect costs caused by system calls that carry out useful work are generally greater, as can be seen in the next micro-benchmark. The degree to which a real-world application can benefit from deactivated privilege isolation depends largely on the frequency at which system calls occur: while the potential time saving for a single system call is modest in terms of absolute numbers because the system-call overhead itself is only moderate in the first place, the accumulated savings have the potential to add up to a perceptible speedup.

7.3.4 Communication Operations

The last in the series of micro-benchmarks analyses the overheads and latencies of the cross-tile communication primitives offered by AtroPOS: messages and push data transfers. No particular focus is put on throughputs because the throughput is dictated predominantly by the memory and cache architecture, not by the operating-system configuration. Since the Intel8 workstation is not a multi-domain NUMA system, the communication benchmark is run only on the three servers. The micro-benchmark considers these aspects:

- **Message/push overhead.** How much effort is it for the sender to transmit a message or a buffer plus a message?
- **Message/push latency.** How long does it take from the moment the send is triggered until the recipient starts processing the incoming data?

\(^5\)The latest Linux security patch that mitigates the Intel-specific Microarchitectural Data Sampling vulnerabilities [Sch+19] by clearing the CPU’s store and fill buffers on kernel exit further increases the system-call overhead to up to 1320 cycles.
7.3 Micro-Benchmarks

These graphs compare the costs of the original OctoPOS x86_64 implementation with those of the AtroPOS communication primitives, the latter in three versions with varying degrees of protection. See also Table C.9 on page 194.

In the benchmark, messages have the obligatory three-word header and an application-defined i-let-sized payload, which adds up to total of 56 bytes on x86_64, one word short of the message-size limit. Push data transfers transmit a message and a small buffer of eight bytes. All communication goes from tile 0 to tile 1; the NUMA topology is not considered. For an in-depth look at the effects the topology has on inter-domain communication, the reader is referred to Chapter 5 of Jens Schedel’s dissertation [Sch18].

The measurements are executed on OctoPOS and on AtroPOS + InvRT with varying degrees of protection and isolation: a bare variant with no communication protection or privilege isolation, an unisolated one where communication protection is enabled (see Section 6.5.2), and an isolated flavour where all protection and isolation mechanisms are in place. InvRT is configured to use the work-stealing scheduler, but the communication system serves the kernel-event queue and thus bypasses the regular user scheduler anyway.

The overheads and latencies measured are displayed in Figure 7.10. We analyse the results with respect to a number of aspects by means of pairwise comparison (as shown from left to right in the graphs).
7 Evaluation and Discussion

AtroPOS versus OctoPOS

Both operating-system kernels in this comparison – OctoPOS and a bare flavour of AtroPOS – have memory protection enabled but offer no communication protection or privilege isolation. The OctoPOS variant of the benchmark makes direct use of the remote-infect and push-data-transfer primitives offered by the OctoPOS kernel. The software implementation of these primitives on x86_64 is relatively naive and uses a rather high degree of indirection (see Section 2.3.2): sending an i-let is achieved by wrapping it into a syscall i-let, which is then written into a shared buffer; an IPI is triggered on the receiving tile and the syscall i-let is executed by the interrupt handler; the syscall i-let unwraps the i-let and submits it to the scheduler. The InvRT library exposes the same communication API as OctoPOS but implements it as a thin wrapper around the AtroPOS kernel’s tailored communication primitives that have been presented in Section 6.5. AtroPOS unburdens the receiving side at the expense of the sender, which carries out all the work; since all four evaluation systems support mwait, the operation is interrupt-free.

One observation that leaps to the eye is that the error bars are roughly comparable across the different machines with OctoPOS – but with AtroPOS, they are far larger on the Intel systems than on AMD. That is, with the AMD Opteron, sending interrupts across NUMA domains is less noisy than writing to a non-local memory address; with the Intel Xeon, the opposite is the case. These properties are determined in large part by the NUMA interconnect and the cache-coherence protocol.

Despite the significant implementation differences, OctoPOS and AtroPOS perform similarly on the AMD system. Message latencies are slightly reduced in AtroPOS while the message overhead and the push latency stay roughly the same. In turn, the push operation becomes more expensive for the sender by a factor of 1.5, caused by the circumstance that the sending CPU core must perform a greater number of atomic memory accesses across NUMA-domain boundaries than on OctoPOS, which is time-consuming on the particular hardware.

On the two Intel-based servers, the picture looks rather different: while the overheads of atomic memory writes across NUMA domains are lower than on AMD48n, sending an IPI is more expensive and the interrupt latency is considerably higher as well. As a consequence, AtroPOS fares far better than OctoPOS in terms of latencies – by a factor of approximately two – because it elides the costly IPI. These performance gains are more pronounced on (the newer-generation) Intel96n than on Intel80n thanks to architectural improvements.

In summary, the AtroPOS implementation performs approximately equally well as OctoPOS on the older AMD machine, and significantly better on the newer Intel processors. As a general observation, there is only little difference with OctoPOS between remote i-lets and push data transfers in terms of overheads and latencies, meaning that the eight-byte buffer copy causes little extra cost. Those differences are more conspicuous with AtroPOS, where the former operation is a simple system call and the latter an extended one.

Influence of Communication Protection

Communication protection is not a contribution of this dissertation, but it is an essential building block that is needed to give full security guarantees. We briefly examine its influence here to put things into perspective. The baseline configuration of AtroPOS in this comparison is the bare variant from before, lined up against a variant that is not privilege-isolated either but where the address-validation checks described in Section 6.5.2 are enabled.
7.4 Application Benchmarks

Since these checks are only part of the data-push operation, the times measured for sending a message are identical. Push data transfers, on the other hand, perform a lookup in the software-managed permission cache, which costs 200–300 cycles on AMD and 600–850 cycles on Intel for a cache hit. A permission-cache miss and the subsequent remote lookup – not shown in Figure 7.10 – increase the overhead of a push data transfer to approximately 4000 clock cycles and the latency to roughly 10 000 clock cycles on all machines.

Cost of Privilege Isolation

Finally, we analyse the differences between the above unisolated AtroPOS and a variant that comes with vertical isolation enabled. The additional fee induced by system calls has two origins: on the one hand the direct cost of entering and leaving kernel mode (confer Section 7.3.3); on the other hand the extra validation of input parameters (see Section 6.7).

The message overhead rises by roughly 350–400 cycles while the message latency is increased by only 90–200 cycles. This is because the overhead subsumes the effort for both entering and leaving the kernel whereas the latency from the recipient’s point of view is affected only by the sender’s kernel entry.

The push data transfer is implemented as an extended system call that takes as input a total of four pointer arguments – unlike claim_send_message(), which accepts only a single pointer. As all pointer arguments are checked by the kernel, the added cost is higher: it lies between 750 and 950 cycles for the overhead and between 650 and 900 cycles for the receiver's latency.

The communication operations are two representatives of system calls in AtroPOS. Privilege isolation increases their cost by a margin of between 24% and 42%. Or, viewed the other way round, letting the application run in privileged mode brings down the overheads of these operations by 19–29%. It is worth noting that, even with all security mechanisms enabled, the latencies of remote infection and push data transfers on AtroPOS are still lower (with the sole exception of the push latency on AMD48n) than their equivalents in OctoPOS, where neither communication protection nor vertical isolation are available.

7.4 Application Benchmarks

The micro-benchmarks serve the purpose of analysing the cost of specific system operations in isolation, and their results, albeit relevant, have only limited expressiveness concerning the behaviour of the overall system in real-life application scenarios. This section examines three benchmark suites and one application program that cover different aspects of the system:

- the NAS Parallel Benchmarks, a suite of parallel numeric programs built on MPI with a focus on communication,
- IMSuite, a collection of concurrent graph algorithms implemented in type-safe X10,
- a real-world geometric multigrid solver written in X10 that stems from the context of Invasive Computing, and
- a number of Cilk Plus benchmarks to compare AtroPOS against Linux.
The benchmark applications measure their own execution time and print it out along with their other (scarce) regular output. Just like for the micro-benchmarks, the textual output data is channelled over the serial console to a companion computer. With the exception of the Cilk Plus benchmarks, which are run only five times for technical reasons\(^6\), each application is executed ten times in a row; all graphs that follow visualise the arithmetic means and standard deviations of those runs. Unlike for the micro-benchmarks, measurement latencies are not subtracted and no system-management interrupts are filtered out because both factors are negligible in the face of execution times in the dimension of seconds to minutes. Furthermore, the InvRT constant `IDLE_LOOP_COUNT` is left at its default value so idle cores do go to sleep eventually.

In the comparisons between OctoPOS and AtroPOS, an implementation aspect has to be highlighted that has so far gone unmentioned: the kernel's TLB-invalidation strategy. OctoPOS follows a rather simplistic approach: whenever the current page-table hierarchy is modified, shutdown IPIs \(^6\) are sent to all cores on the local tile that belong to the affected application. AtroPOS uses the same strategy for modifications that restrict the mapping (by revoking permissions or the page's presence bit), but it behaves lazily when it comes to enhancing the mapping. That is, when AtroPOS makes a memory page available to an application, it invalidates only the corresponding TLB entry on the local core but does not notify the application's other cores. If one of those other cores subsequently tries to access the newly allocated page, it may run into a page fault; the kernel's trap handler then takes care of updating the core's TLB. As we will see, this lazy invalidation strategy is effective for programs whose memory-allocation behaviour causes the page mapping to change frequently.

### 7.4.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks\(^7\) are a standard benchmark suite developed and maintained by the NASA Advanced Supercomputing Division. They are intended for assessing the performance of parallel supercomputers. The benchmarks consist of a set of programs that mimic the typical computation and data-movement characteristics of applications in the field of computational fluid dynamics \[^5\]\. An overview of the benchmark applications is given in Table 7.4. The NAS Parallel Benchmarks come in several flavours that use different programming models for parallelisation and communication, for example MPI, OpenMP or Java. The evaluation with AtroPOS is conducted using version 3.3.1 of the best-known variant, which is written in Fortran and C and is structured into parallel processes that communicate via MPI \[^5\]. The programs are linked against the MPI library described in Section 2.3.4 that was originally developed for OctoPOS, which in turn builds upon the programming interface provided by OctoPOS and InvRT.

The applications can be configured to run with one of eight different classes of problem sizes: S (small, for testing purposes), W (for ancient workstations), A–C (standard sizes) and D–F (large sizes) \[^5\]; not all applications support classes E and F. As shown in Table 7.5, classes C and D are used in this evaluation, based on the following selection logic: choose class D for an application if its data fits into main memory and a single run takes no more than 1000 seconds; pick class C otherwise. The FT benchmark in class D runs out of memory on AMD48n (which has only 8 GiB RAM per tile); BT, LU and SP exceed the 1000-second time limit, so these four applications are configured as class C. The number of MPI processes is set per machine: it generally equals the number of logical cores available in the system, but most benchmarks impose additional constraints in that they require the process count to be a square number or a power of two.

\(^6\)Most of the Cilk Plus programs contain a built-in loop in their source code that executes the measurements five times.

\(^7\)https://www.nas.nasa.gov/publications/npb.html
7.4 Application Benchmarks

<table>
<thead>
<tr>
<th>Name</th>
<th>Full name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Kernels</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>Conjugate gradient</td>
<td>Conjugate-gradient solver for unstructured sparse linear systems.</td>
</tr>
<tr>
<td>EP</td>
<td>Embarrassingly parallel</td>
<td>Generator of pairs of Gaussian random variates.</td>
</tr>
<tr>
<td>FT</td>
<td>Fast Fourier transform</td>
<td>3-D fast-Fourier-transform solver for partial differential equations.</td>
</tr>
<tr>
<td>IS</td>
<td>Integer sort</td>
<td>Parallel sort over small integers.</td>
</tr>
<tr>
<td>MG</td>
<td>Multigrid</td>
<td>Approximate multigrid solver for 3-D discrete Poisson equations.</td>
</tr>
<tr>
<td><strong>Apps</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BT</td>
<td>Block tridiagonal</td>
<td>Three different solvers for a system of nonlinear partial differential equations.</td>
</tr>
<tr>
<td>LU</td>
<td>Lower–Upper Gauß–Seidel</td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>Scalar pentadiagonal</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: NAS Parallel Benchmarks.

The benchmarks consist of five kernels and three pseudo-applications with different memory-access and communication patterns.

<table>
<thead>
<tr>
<th>Class</th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD48n</td>
<td>C</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>Intel80n</td>
<td>36</td>
<td>32</td>
<td>48</td>
<td>32</td>
<td>32</td>
<td>48</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>Intel96n</td>
<td>64</td>
<td>64</td>
<td>80</td>
<td>64</td>
<td>64</td>
<td>80</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td># Procs</td>
<td>81</td>
<td>64</td>
<td>96</td>
<td>64</td>
<td>64</td>
<td>96</td>
<td>64</td>
<td>81</td>
</tr>
</tbody>
</table>

Table 7.5: Configurations of the NAS Parallel Benchmarks.

The table shows for each application the problem-size class and the number of parallel MPI processes for which it is configured.

The MPI library stakes one elementary claim per MPI process. Each such claim comprises a single CPU core, giving the user scheduler zero freedom in terms of work distribution. The degree to which the NAS Parallel Benchmarks use fine-grained concurrency is very low – there is none from the applications themselves, only some induced by the internal structure of the MPI runtime system. Hence, these benchmarks primarily serve as a test bed for communication efficiency and not for user-scheduling strategies. Most of the memory is either allocated statically or reserved right at the outset of the program; dynamic memory allocations are limited in amount and number. It is expected that AtroPOS + InvRT perform better than OctoPOS, that the choice of user-scheduling strategy is unpertinent and that the applications run slightly faster in privileged mode than in unprivileged mode.

Since the focus is on inter-tile communication, the measurements are performed on the three NUMA server systems only. The mean times and standard deviations are plotted in Figure 7.11. Due to an unknown bug, the OctoPOS version of LU crashes on Intel80n and Intel96n and is exceedingly slow on AMD48n (over 10 min as opposed to less than 60 s), which is why the corresponding data points are left out. The absolute execution times – annotated on top of the bars for the baseline – vary between 24 s and 14 min. In the light of the absolute times on the three different machines, it can be asserted that the benchmarks generally scale well to the number of cores available.

**AtroPOS versus OctoPOS** In the direct comparison between OctoPOS and AtroPOS, the latter wins across the board thanks to the superior implementation of its communication subsystem: on average, the applications are 1% faster on AMD48n, 2.6% on Intel80n and 3.1% on Intel96n. These numbers reflect the results for the communication latencies from Section 7.3.4. The greatest improvements can be observed in the IS and SP applications: up to 3.2% on the AMD machine and
Figure 7.11: Execution times of the NAS Parallel Benchmarks.

The measurement results on OctoPOS serve as the baseline; their absolute time values are annotated at the top of the respective graph. For better legibility, the bars are cut off below 0.85. See also Table C.10 on page 195.
up to 8.3% on Intel hardware. Both benchmarks are communication-intensive: IS features a lot of all-to-all communication while the SP processes exchange data in a point-to-point fashion but everyone still talks to everyone else [TS99].

**User scheduling** As anticipated, the choice of user scheduler has a very limited impact for MPI applications. In relation to the work-sharing scheduler, its work-stealing counterpart is either on a par or minimally faster on average. The only notable outlier is the LU benchmark on Intel96n (but not on the other two servers), where the work-stealing scheduler performs 3% better.

**Cost of communication protection** With the extra security checks for push data transfers enabled, the slowdown compared to the unprotected variant is all but unnoticeable. As the applications’ memory layouts are largely static and the protection cache is big enough, its hit rate reaches nearly 100% after a brief initial warm-up phase. The cost of these cache lookups increases the overall mean execution time by only 0.1% (on Intel96n) to 0.3% (on AMD48n); none of the applications are slowed down by more than 1% on any machine.

**Effect of privilege isolation** The effect that privilege isolation has on the application performance depends greatly on the processor model: on the AMD Opteron, running the benchmarks isolated from the kernel as opposed to unisolated increases the execution time by an average of 0.8%, with the MG benchmark being slowed down by a whole 4.4%. By contrast, the mean slowdown on the Intel Xeon machines amounts to a mere 0.1% or even less. In other words, the potential for performance improvements by switching off privilege isolation is extremely narrow for the NAS Parallel Benchmarks.

All in all, a completely isolated application running in user mode on top of an AtroPOS kernel that provides full security guarantees for the system is highly competitive with its counterpart running on OctoPOS with memory protection but no other security guarantees. On AMD48n, the difference in the execution times is only marginal when averaged over all benchmarks. On the Intel machines, AtroPOS is actually faster by 2.6% and 3%, respectively. The lion’s share of that speedup can be attributed to the superior communication subsystem implemented in AtroPOS.

### 7.4.2 IMSuite

Being written in Fortran and C, the NAS Parallel Benchmarks are representatives of non-type-safe user programs. As an example of a collection of applications that may be executed with loosened isolation properties without sacrificing security because they are type-safe and memory-safe by design, we examine IMSuite\(^8\), the *IIT Madras benchmark suite for simulating distributed algorithms* [GN15]. It implements twelve distributed algorithm kernels, mostly of common graph algorithms used in networking applications. Table 7.6 contains an itemisation of the individual programs. IMSuite comes in four variations, three of them type-safe: two written in X10 (targeting the language versions 2.3 and 2.5, respectively), one written in Habanero-Java [Cav+11] and one implemented in C using OpenMP.

A modified version of the X10-FA 2.3 IMSuite, which manages concurrency with the `finish`, `async` and `atomic` constructs, is used in this evaluation. It is compiled using the x10firm toolchain and linked against the invadeX10 runtime library described in Section 2.3.4, which consumes the

\(^8\)[https://www.cse.iitm.ac.in/~krishna/imsuite/](https://www.cse.iitm.ac.in/~krishna/imsuite/)
Table 7.6: IMSuite benchmarks.
The suite comprises twelve distributed graph algorithms, most of them related to the problem field of networking.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BF</td>
<td>Distance computation through breadth-first search using the Bellman–Ford approach [Bel58].</td>
</tr>
<tr>
<td>DST</td>
<td>Creation of a breadth-first-search tree using Dijkstra’s algorithm [Di59].</td>
</tr>
<tr>
<td>BY</td>
<td>Consensus building in the face of malicious nodes (Byzantine generals’ problem [LSP82]).</td>
</tr>
<tr>
<td>DR</td>
<td>Routing-table creation using Dijkstra’s algorithm.</td>
</tr>
<tr>
<td>DS</td>
<td>Probabilistic determination of a graph’s dominating set.</td>
</tr>
<tr>
<td>KC</td>
<td>Graph partitioning into committees of size at most 𝑘.</td>
</tr>
<tr>
<td>DP</td>
<td>Leader election in a general network using David Peleg’s algorithm [Pel90].</td>
</tr>
<tr>
<td>HS</td>
<td>Leader election in a ring network using the Hirschberg–Sinclair algorithm [HS80].</td>
</tr>
<tr>
<td>LCR</td>
<td>Leader election in a ring network using the Le Lann–Chang–Roberts algorithm [CR79].</td>
</tr>
<tr>
<td>MIS</td>
<td>Randomised computation of a graph’s maximal independent set.</td>
</tr>
<tr>
<td>MST</td>
<td>Determination of a graph’s minimum spanning tree.</td>
</tr>
<tr>
<td>VC</td>
<td>Vertex colouring of a tree with three colours.</td>
</tr>
</tbody>
</table>

programming interface provided by either OctoPOS or InvRT. The evaluation is carried out with a patched software version of IMSuite; the patch was developed by Manuel Mohr [Moh18] and resolves a major performance issue caused by excessive copying of objects between places. That issue stems from a pitfall in the X10 language, where an activity implicitly captures the entire this-object even if only a field of this is accessed from within the activity, leading to a needlessly inflated closure object and far too much deep copying. With said patch, the required fields are explicitly copied into local variables, which are then captured by the activity, thus preventing any inadvertent capture of this.

The applications are generally structured in such a way that a simple iterative loop spawns large quantities of async activities. This way, a single CPU core at a single place is responsible for the creation of concurrency and all cores are consumers of that work. There exist official alternative variants of the BF, DST, BY, DR and MST benchmarks that initiate concurrency in a recursive, tree-like pattern – but these variants suffer from a problem of their own: since the user scheduler’s run queues are bounded in capacity, their space is quickly exhausted when all cores simultaneously attempt to enqueue a massive number of i-lets. Such a situation is easy to reproduce with those five applications and is currently not handled gracefully in OctoPOS and InvRT: after a fixed number of unsuccessful retries, the scheduler gives up and discards the i-let in question. As a possible graceful recovery mechanism, the scheduler could decide on a case-by-case basis to dispense with additional concurrency and to execute the i-let immediately – but this is not currently implemented in InvRT. As a consequence, the time measurements are carried out with the iterative variants of the benchmarks.

Even with the patch described above, the IMSuite benchmarks scale surprisingly badly on x86_64 NUMA systems – so badly that the performance gets worse with higher numbers of CPU cores. This behaviour is likely related to the concurrency pattern described in the previous paragraph. Meaningful results can only be retrieved on the single-tile Intel8, which has eight logical cores. For unclear reasons, the DP application reproducibly crashes on that machine when executed on OctoPOS; the same problem does not occur with AtroPOS.

The benchmarks are run with maximum-sized predefined input parameters. Three applications allocate large quantities of memory, triggering several garbage-collection runs because of memory

---

9Implementing this feature would require support for partial stack unwinding in order to handle calls to signalAndExit() correctly.
exhaustion: *BY* three times per program execution, *MST* four times and *DR* 31–33 times. The other nine benchmarks terminate as normal before their heap reaches the size of the available RAM. The *IMSuite* programs are generally smaller and shorter than the NAS Parallel Benchmarks, with the minimum execution time only 45 ms and the maximum approximately 100 s. The results are visualised in Figure 7.12, divided up into three pairwise comparisons.

### AtroPOS versus OctoPOS
All *IMSuite* applications take less time to run on AtroPOS than on OctoPOS, with a difference of up to 10% in the *BY* benchmark. On average, the speedup amounts to 6.5%. This is owed largely to the aforementioned lazy TLB invalidation. This optimisation has a big impact because the memory-management library, instead of preallocating a large user heap during application start-up, acquires pages from the kernel on demand. This leads to frequent modifications of the page table by the `mem_map()` system call, resulting in frequent TLB updates; lazy invalidation reduces the cost of these updates and increases scalability. As the measurements are done on a single-tile system, the different communication operations have no relevance here.

### Work stealing versus work sharing
One would intuitively expect that the work-stealing scheduler performs better than its work-sharing counterpart – after all, it facilitates dynamic load redistribution. However, this presumption is not backed up by the findings for *IMSuite*: with the work-stealing user scheduler, the benchmarks are 0.2% slower on average. In the best case (*DR*), the application performs 0.8% better; in the worst case (*MIS*), it takes a hefty 3.4% longer to finish. This is the result of an unfavourable combination of the user scheduler’s theft strategy and the concurrency-creation pattern explained above. The user scheduler employs child stealing instead of continuation stealing (see Section 3.2.2). This way, a lot of pressure is put on the run queue of the core that spawns the activities, which becomes a major bottleneck; as a consequence, the spawning i-let is frequently forced to temporarily back off from creating new work because the thieves cannot keep up with the insertion rate. This performance issue could be alleviated in InvRT either by implementing continuation stealing or by providing an intelligent mechanism for dealing with run-queue exhaustion in order to better support the aforementioned recursive benchmark variants.

### Cost of privilege isolation
The difference between statically and dynamically configured privilege isolation – not shown explicitly in Figure 7.12 – is mostly negligible, but there are three outliers (*DST*, *LCR* and *VC*) where the dynamic variant is actually up to 1.2% faster. It is unclear to what this unexpected result can be attributed; one may speculate that the slightly different code layout happens to be better suited for the instruction cache. In the comparison between *IMSuite* running in privileged mode and the suite running in user mode, the latter benchmarks are either equally fast or take up to 1.5% longer. Averaged over all applications, the mean speedup that can be gained by switching off vertical isolation amounts to 0.6%.

In summary, *IMSuite* is considerably faster with every single configuration variant of AtroPOS than it is with OctoPOS. Even when fully isolated from the operating system, the benchmarks run 5.8% more efficiently on average than with OctoPOS. The primary contributing factor to the efficiency gains is lazy TLB invalidation, whose impact is amplified by the frequent page allocations issued by the heap-management library. The work-stealing scheduling strategy fails to live up to its potential because the *IMSuite* applications’ unfavourable concurrency patterns prevent the scheduler from playing to its strengths. Being type-safe and memory-safe by design, the *IMSuite* benchmarks are suitable candidates for loosened isolation – but while relinquishing privilege isolation does lead to lower execution times, its practical effect is not world-shaking overall.
Figure 7.12: Execution times for the IMSuite benchmarks on Intel8.

The measurement results for different operating-system variants are compared pairwise; their absolute time values are annotated at the top of the respective graph. For better legibility, the bars are cut off below 0.85 and 0.96, respectively. See also Table C.11 on page 195.
7.4 Application Benchmarks

Figure 7.13: Execution times of the multigrid application.
The graph shows the absolute execution times on the four test machines. See also Table C.12 on page 196.

7.4.3 Geometric Multigrid Solver

The final benchmark in the comparison between OctoPOS and AtroPOS is a geometric multigrid solver that was originally developed in the context of invasive computing [Bun+13]. It performs a time-dependent simulation of the heat distribution on a metal plate that is being engraved by a laser. In each time step, the solver is applied recursively on a discrete grid that is first coarsened and then refined again – running through a so-called V-cycle. The multigrid application is written in X10 and was originally developed to demonstrate how multiple invasive applications sharing the same system can benefit from continual bargaining for computation resources. The V-cycle is an ideal candidate for showcasing this because the varying grid resolution induces gradually changing resource demands. However, since resource bargaining is not in the focus of this work, the evaluation is based on a simplified use case where a single multigrid application invades all CPU cores during start-up and does not release any as its grid becomes coarser. The program operates with a discrete $1024 \times 1024$ raster image as input that describes the stencil for the laser engraving; the stopwatch determines how long it takes to simulate 1000 time steps.

The mean execution times on the four evaluation systems are plotted in Figure 7.13. It can be observed that the application scales reasonably well with the available parallelism – but in relative terms, the performance on the single-tile eight-core Intel8 is considerably higher than on the multi-tile 48-, 80- and 96-core servers. This is not least attributable in part to the significantly higher clock frequency of that machine’s processor, being clocked at 3.5 GHz as opposed to 2.5 GHz or less. Garbage collection is triggered 29 times per program execution on AMD48n, which has 8 GiB RAM per tile, four times on Intel80n and Intel8 (32 GiB per tile), and never on Intel96n (128 GiB per tile).

AtroPOS versus OctoPOS  On AMD48n, the multigrid benchmark is only insignificantly faster with AtroPOS than with OctoPOS. By contrast, on the three Intel machines, AtroPOS yields a speed advantage that lies between 1.2% (on Intel8) and 2.2% (on Intel80n). The origin of this advantage is twofold: on the one hand, the application benefits from the optimised communication operations provided by AtroPOS on the multi-tile systems; on the other hand, lazy TLB invalidation makes the memory-allocation operations more efficient. Since the multigrid application is neither
as communication-intensive as the NAS Parallel Benchmarks nor as memory-intensive as IMSuite, the overall effect is somewhat attenuated compared to those two.

**User scheduling** With one large elementary claim per tile, the multigrid-solver application grants full freedom to the user scheduler, thus providing a good environment for comparing the work-sharing and work-stealing schedulers with each other. In line with expectations and in contrast to the IMSuite results, work stealing yields higher performance because of its ability to dynamically rebalance the i-let queues as needed. While the extent of the efficiency gains is moderate on the Intel systems, ranging from 0.6% on Intel8 to 1.3% on Intel96n, it is enormous on the AMD machine: with work-stealing scheduling, the benchmark runs a whole 19% faster. This is caused by the work-sharing scheduler reproducibly triggering a pathological case where the i-lets are distributed so unevenly that a small number of cores still have many i-lets in their run queues while the remaining cores are all idle – a situation that is successfully avoided through stealing.

**Dynamic privilege isolation and communication protection** The comparison between the bare and the unisolated AtroPOS configurations yields results that are counter-intuitive. The latter variant differs from the former in two aspects: communication protection is enabled and privilege isolation is dynamic, albeit disabled. The intuitive expectation would be that there is a performance penalty. This is true on the single-tile Intel8, where no communication takes place and the benchmark is minimally slower – but the exact opposite is the case on the other three systems: there is a speedup by up to 2%. No immediate handy explanation can be given for this phenomenon; the presumption is that communication using push data transfers is sufficiently infrequent that it carries no weight, and that – as previously conjectured for IMSuite – the code layout happens to be more benign for the instruction cache, the branch predictor or other microarchitectural entities.

**Cost of privilege isolation** As the application is implemented in type-safe X10, it is a suitable test bed for optionally disabled privilege isolation. When the simulation runs unisolated instead of isolated from the operating system, it finishes 0.7% to 1% earlier on the multi-tile systems. On the single-tile system, where no communication takes place and system calls are predominantly related to memory management or idling, the efficiency gain is insignificant. Again, the multigrid application running fully isolated on AtroPOS is faster across the bench than it is on OctoPOS with weaker security properties. The multigrid benchmark is an excellent showcase for the advantages of work-stealing scheduling, imposingly demonstrating the benefits of being able to rebalance the user scheduler’s run queues during execution. Besides, the application steers a middle course between communication and shared-memory parallelism; it is neither as communication-centric as the NAS Parallel Benchmarks nor as synthetic as IMSuite. Even so, the multigrid solver confirms that the communication infrastructure and other design and implementation aspects of AtroPOS are superior to those of OctoPOS, and that the elective renunciation of hardware-enforced privilege isolation can yield minor, yet noticeable, efficiency improvements.

### 7.4.4 Cilk Plus Benchmarks

To showcase the suitability of the AtroPOS kernel’s design and interface for arbitrary concurrency platforms, the AtroPOS port of the Cilk Plus runtime system is evaluated with the aid of a number of application benchmarks. The programs comprise two official Cilk Plus example applications,
Table 7.7: Cilk Plus application benchmarks.

The benchmarks execute generic algorithms and a set of specialised algorithms from the fields of finance and geology.

<table>
<thead>
<tr>
<th>Name</th>
<th>Full name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BL</td>
<td>Binomial lattice</td>
<td>Option-price computation using the binomial-lattice model for Brownian motions.</td>
</tr>
<tr>
<td>FI</td>
<td>Fibonacci</td>
<td>Computation of the 39th Fibonacci number using naive recursion.</td>
</tr>
<tr>
<td>KA</td>
<td>Karatsuba</td>
<td>Multiplication of large polynomials using the Karatsuba algorithm.</td>
</tr>
<tr>
<td>MA</td>
<td>Mandelbrot</td>
<td>Computation of the Mandelbrot set.</td>
</tr>
<tr>
<td>MC</td>
<td>Monte Carlo</td>
<td>Valuation estimation for swaptions by applying the Monte Carlo algorithm.</td>
</tr>
<tr>
<td>MS</td>
<td>Merge sort</td>
<td>Recursive merge sort on an array of 100 million integers.</td>
</tr>
<tr>
<td>RS</td>
<td>RTM stencil</td>
<td>Stencil computation for the Reverse Time Migration algorithm in seismic computing.</td>
</tr>
</tbody>
</table>

Figure 7.14: Relative execution times of the Cilk Plus application benchmarks on Intel8.

The results are normalised to the respective execution times on Linux. The absolute run times are annotated on top of the pertaining bars. See also Table C.13 on page 196.

Cilk Fib and Karatsuba10, plus a subset of the Intel C++ Compiler sample-code collection11, which contains algorithms for financial and geological applications. A list of the programs is provided in Table 7.7. For the comparison, the benchmarks are executed on AtroPOS and on Linux. Their source code is left unmodified and the relevant compiler flags are kept identical for both builds.

The primary purpose is to demonstrate that it is possible to run Cilk Plus applications on AtroPOS, and efficiently so. Since the ported runtime system is currently not integrated with AtroPOS’ communication system, only single-tile operation is supported. Hence, the benchmarks are run only on Intel8. To execute their Linux variants, the machine is booted into Ubuntu 16.04.5, which ships the Linux kernel in version 4.4.0, glibc 2.23 and libcilkrts 5.4.0; the ELF binaries are linked dynamically against those libraries. Linux is configured to use the performance CPU governor, which pins the clock frequency at 3.5 GHz. For each application, two dry runs are performed to warm up the system, followed by five measured runs. The results are presented in Figure 7.14. As there are no major differences whether the applications on AtroPOS run in privileged or in unprivileged mode, only the numbers for the latter are shown. The run times on Linux and on AtroPOS are largely comparable with each other, but there are three significant outliers: while KA is 8.2% slower on AtroPOS, BL and MC are both over 25% faster.

---

10https://www.cilkplus.org/download#block-views-code-samples-block-1
7 Evaluation and Discussion

The KA application creates a vast number of Cilk activities, which causes many dynamic memory allocations and deallocations from within the Cilk Plus runtime library. For this, the runtime system calls the heap-management functions `memalign()` – a variant of `malloc()` that honours user-specified alignment constraints – and `free()`. As mentioned earlier, the heap-management library provided with AtroPOS fetches additional memory pages from the kernel by invoking the system call `mem_map()` when necessary, and eagerly releases unused pages with `mem_unmap()`. By contrast, the Linux libc expands and shrinks its heap with the `brk()` system call, whose internal implementation is simpler and thus more efficient.

The enormous performance gains in the BL and MC benchmarks are attributable to the handling of idle cores. In the Linux version of libcilkfts, a worker thread that runs out of tasks does not go to sleep; it merely issues a `sched_yield()` system call to make way for other runnable kernel threads. The library does include a mechanism for putting workers to sleep with the help of POSIX semaphores, but that mechanism is used only during start-up and shut-down of the runtime system. Consequently, even in phases when the application cannot fully exploit the available parallelism, the system constantly remains under 100% load. An unusually high frequency of calls to `sched_yield()` can be observed especially in MC but also in BL, which are both relatively short at only a few hundred milliseconds. With AtroPOS, the runtime system is able to implement – in close cooperation with the kernel – a more straightforward and efficient idling mechanism that does put idle worker cores to sleep (see Section 6.8.4).

All in all, the evaluation demonstrates that executing Cilk Plus applications with the AtroPOS port of libcilkfts not only works but is more than competitive with Linux. The kernel interface of AtroPOS is well suited as a foundation upon which the runtime systems of arbitrary concurrency platforms can be built.

7.5 Discussion and Summary

After AtroPOS and InvRT have been subjected to an extensive evaluation with code-size measurements, selective micro-benchmarks and real-world application-benchmark suites, the core findings are summarised and briefly discussed here.

The overall memory footprint of the new system does not differ significantly from that of OctoPOS. In the comparison of code sizes, the AtroPOS kernel and the InvRT runtime library combined are several kibibytes larger than the OctoPOS monolith – but this overhead can be considered completely negligible on a machine with gibibytes of memory. On the other hand, the size of the kernel’s text section – that is, the extent of the trusted computing base – is reduced by 10%.

To analyse the runtime behaviour and the performance of the system design, AtroPOS was pitted against OctoPOS. The duel was conducted with communication-centric MPI benchmarks, an X10 benchmark suite and an invasive simulation application also written in X10. In the direct comparison with an equal feature set and identical protection settings, AtroPOS is faster than OctoPOS across the bench, often by a considerable margin; it is only in a few rare cases that both are neck and neck. Even when the application runs in unprivileged mode and the extra checks for communication protection are enabled in AtroPOS, most of the time it still outperforms the same application running with less protection on OctoPOS. The two factors mainly responsible for these efficiency improvements are the streamlined and better structured subsystem for cross-tile communication and the lazy TLB-invalidation technique. This bears witness that AtroPOS is not only a mere redesign of OctoPOS that adds new functionality, but that it also places a strong emphasis on the non-functional properties on the efficiency front.
The invasive runtime system, InvRT, was evaluated with the work-sharing and work-stealing user schedulers. The micro-benchmarks establish that, in addition to the obvious advantage of dynamic i-let redistribution, work stealing can generally be expected to yield better performance figures. This expectation is largely confirmed by the application benchmarks – but it hinges on apt behaviour on the part of the application. That is, in order to reach the best results, the programmer needs to be aware of the concrete scheduler implementation – especially of whether it employs continuation stealing or child stealing – and structure the program’s creation of concurrency accordingly.

The new and enhanced synchronisation data structures were analysed in micro-benchmarks. The benchmarks confirm the intuition that the spawning signal mode is more efficient in terms of overheads and latencies than the blocking mode of operation, provided the state to be transferred is small or non-existent. The newly introduced unified counting signal is on par with the corresponding specialised blocking and spawning synchronisation structures in almost every respect, so it is suitable as a drop-in replacement for either, given its superior predictability. Moreover, the binary signal, which supports both modes of operation as well, is not only fully wait-free but also outperforms its counting peer in every way. It is demonstrated that the signal and exit operation, which was implemented for all signal types, is indeed able to severely bring down activation latencies by circumventing the user scheduler; consequently, there is no reason not to use it in places where signalling is the last action an i-let performs before terminating. All in all, although no explicit pre–post comparison was carried out, the wait- and activation latencies prove to be very low thanks to the efficiency improvements for user-level context switches described in Chapter 4.

One of the unique selling points of AtroPOS is its configurable privilege isolation. The primary motivation for letting the application run in privileged mode is the prospect of being able to cut down on the direct system-call costs. In fact, the micro-benchmarks prove that an enormous reduction of the system-call overheads in relative terms – down to almost zero for simple system calls – can be reached this way. However, in practice, for most system calls the amount of work done in the body far outweighs the entry- and exit overhead. Still, a clear efficiency gain is measurable for certain classes of programs. Hence, if used purposefully, the facilities offered by AtroPOS can indeed make a difference. Interestingly, with dynamically configurable privilege isolation, the application benchmarks perform better in the majority of cases than with support for privilege isolation statically configured out. This is a surprising outcome because intuition would suggest the opposite and the micro-benchmarks indicate equality. For now, the blanket recommendation for application programmers is to use a variant of AtroPOS that offers reconfigurable privileged isolation, as it provides both maximal flexibility and superior performance.

The port of the Cilk Plus runtime system demonstrates the versatility of AtroPOS and its aptitude to serve as a basis for arbitrary concurrency platforms. Cilk Plus applications running on AtroPOS are more than competitive with those running on Linux. One particular advantage over Linux is the straightforward kernel interface for idling, although other assets include higher specialisation and tailoring explicitly for concurrent programs and better support for multiple applications through resource exclusivity.

The bottom line of the evaluation is that the goals specified at the outset have been largely met. Despite the extensive redesign that enhances the system’s security, flexibility and configurability, the platform consisting of AtroPOS and InvRT is capable of outperforming its predecessor OctoPOS. With this, AtroPOS is a solid foundation for building efficient and scalable concurrent applications.
Conclusion

This thesis has presented the design and implementation of AtroPOS, an atrophied parallel operating system that aims to specifically support concurrency platforms on multi- and manycore systems. The work is motivated by the insight that threads are not the best-suited abstraction for writing software able to exploit a vast degree of available parallelism: threads are too heavyweight to serve as high-level building blocks for fine-grained concurrent programming, and they have undesirable properties that make them inept as a low-level foundation for concurrency platforms. Among these issues are a lack of cooperation between the operating system's thread scheduler and the scheduler of the runtime system, uncoordinated competition among multiple applications for computing resources, and the thread-blocking anomaly.

This chapter summarises the work done and the results that were achieved, and provides an outlook into possible future work.

8.1 Summary

The fundamental idea behind AtroPOS is to construct a lightweight operating system that offers a low-level foundation on top of which the runtime systems of arbitrary concurrency platforms can reside. The processor's cores are made available to the application as is; the virtualisation of cores is optional and off by default. AtroPOS is derived from OctoPOS, the parallel operating system for invasive computing, and adopts many of its guiding principles: hardware centricity, the exclusive issuance of resources in the form of claims, static and dynamic configurability and tailorability – but it is a major redesign of the operating-system architecture, with greater genericness and a reworked execution model. One core feature of AtroPOS is configurable and dynamically reconfigurable privilege isolation. Compared to OctoPOS, the kernel is reduced in scope and size; functionality that covers the scheduling, dispatching and synchronisation of lightweight i-lets is relocated into a dedicated user-space runtime system named InvRT.

In the AtroPOS design, kernel-level control flows – called kernel jobs – are decoupled from application-level activities. Both are scheduled separately and independently on their respective level; a particular focus of this work lies on the interplay between them. The kernel execution model is hybrid: it is event-based by default but becomes process-based on demand when a kernel...
job blocks. Kernel jobs are triggered by interrupts or by system calls, of which there are two types: *simple* calls are synchronous and run to completion; *extended* calls potentially block during their execution and have asynchronous semantics. That is, when an extended system call blocks, control is immediately returned to user space, giving the application the opportunity to perform other concurrent activities in the meantime. When the operation finishes, the kernel notifies the user program by pushing a kernel event, a reference to an application-defined object, into an event queue. The event queue is implemented using shared memory and is synchronised in a non-blocking manner. It is used not only as an asynchronous notification mechanism but also as a foundation for arbitrary user-level synchronisation constructs such as futures, and as a generic means of communication between parts of a distributed application on a tiled multiprocessor. The system operates in a mostly interrupt-free manner to keep interference minimal.

One distinctive feature of AtroPOS is its configurable vertical isolation: the kernel memory can be cordoned off from user applications using the hardware mechanism available in the CPU, but that barrier can alternatively be established purely in software or it can be dispensed with entirely. If desired, this decision may be altered at run time through dynamic reconfiguration. Depending on the isolation status, system calls are executed either with a traditional privilege-mode switch or as ordinary function calls, with the latter having reduced direct and indirect invocation costs.

InvRT, the invasive runtime system, is a user-level implementation of the *i*-let execution model and the invasive programming interface. It schedules and dispatches *i*-lets onto the CPU cores exposed by the AtroPOS kernel, supporting both work-sharing and work-stealing strategies. Idle cores are put to sleep and awoken with the help of a specialised cross-layer mechanism. InvRT makes heavy use of kernel events to implement application constructs such as futures, completion *i*-lets and remote *i*-lets. In addition, the Cilk Plus runtime library was ported to AtroPOS as a proof of concept.

The *i*-let execution model was revisited and selective improvements were made to benefit the efficiency and predictability of synchronisation. On the one hand, context management was improved and context switches were made faster; on the other hand, new synchronisation data structures and primitives for common use cases were introduced: a wait-free binary signal – a kind of private binary semaphore – for synchronisation scenarios where only a single event is waited for, and a unified counting signal that supports both the blocking and the spawning mode of operation, allowing the application to decide at run time which mode to use. Last but not least, a handoff mechanism was implemented that allows the direct activation of an unblocked or spawned *i*-let by circumventing the user scheduler.

The implementation of the aforementioned concepts was evaluated with the aid of micro-benchmarks and of various application suites written in C, Fortran, X10 and Cilk Plus. The measurements confirm the effectiveness of the synchronisation improvements. Work-stealing scheduling is shown to be generally preferable over work sharing because the scheduler operates more efficiently and is capable of dynamic load redistribution, but the application's concurrency must be structured appropriately. The option to switch off privilege isolation has the potential to benefit certain user programs that are system-call-heavy, and the fact that vertical isolation can be reconfigured at run time and at core granularity opens up a wide range of possible application scenarios. Generally, application programs running on AtroPOS perform better across the board than on OctoPOS, bearing witness to the efficacy of both the superior design of AtroPOS and the optimisations carried out on the implementation level.
8.2 Outlook

While AtroPOS is a well-functioning, performant and largely mature operating-system prototype, there is ample opportunity for future work. This section presents a brief collection of possible improvements and enhancements, and outlines opportunities for further research activities.

Several system parts have been identified in this work where there is a potential for refinement. The scheduler infrastructure in InvRT could implement a sophisticated mechanism for dealing with run-queue exhaustion – for example, the infect operation could opt to dispense with concurrency if the queue is full, executing the i-let synchronously instead. This would benefit applications with highly unbalanced concurrency-creation patterns, such as those evaluated in Section 7.4.2. Also, a variant of the work-stealing scheduler could be built that uses continuation stealing instead of child stealing. Optionally, the memory-management library could eagerly preallocate larger chunks of memory pages for the heap instead of doing so on demand, and it could release unused pages far more lazily. This could reduce the frequency of system calls and TLB invalidations. Finally, KPTI – the optional near-complete masking of the kernel’s memory pages from the user address space, which is an essential defence technique against the Meltdown processor vulnerability – could benefit from operating-system support for TLB tagging (PCID by Intel or ASID by AMD) to keep the mode-switching costs at a minimum.

Currently, the primary hardware platform supported by AtroPOS is x86_64. The SPARC LEON3 variant needs additional work on the implementation of privilege-mode switches and system calls. In order to securely integrate the invasive hardware components into a privilege-isolated system, a number of modifications to these components are advisable. A list of pragmatic recommendations concerning this topic can be found in Appendix B. Since AtroPOS is equally portable in principle as OctoPOS, targeting further architectures such as AArch64 is easily conceivable.

One topic that has only been covered tangentially in this thesis is claim virtualisation, which constitutes the basis for selective time sharing of compute resources between applications. While a general outline of the optional preemptive scheduling of elementary claims has been sketched out in Chapter 5, a complete realisation of that concept is subject to further research.

The OctoPOS Ethernet stack has been adapted to work with AtroPOS, but no performance measurements with networked applications have made their way into this thesis. This is owed to the fact that an entirely new TCP/IP stack for OctoPOS – based on the lwIP project [Dun01] – was under development at the time of writing. It would be interesting to bring that new network-communication infrastructure to AtroPOS and to evaluate the efficiency of data-intensive concurrent programs that send and receive large amounts of data over Ethernet.

To support a wider range of concurrent applications, the runtime systems of additional concurrency platforms such as OpenMP, Go or Threading Building Blocks may be ported to the AtroPOS API. The difficulty of this undertaking will vary from system to system. For instance, it is expected that porting Go, where the dependency on the thread concept is rooted deeply in the runtime library, would be an ambitious endeavour that would require the reimplemention of considerable parts of the library – see the reasoning by the Akaroas authors on that issue [Klu15, Section 4.2]. By contrast, adapting an existing open-source OpenMP runtime library such as libgomp or libomp (from the GCC and Clang projects, respectively) is likely to be similarly feasible as porting libcilkrts, the Cilk Plus library.

In the long run, once AtroPOS and InvRT have been brought to the same level as OctoPOS in terms of platform support and features, their code base could be merged back into the mainline branch of OctoPOS, thus recombining the strengths of both systems.
8 Conclusion

8.3 Closing Remarks

The result that has emanated from this work is AtroPOS, the design and implementation of an operating system that is …

... **lightweight**. The operating-system kernel has a small memory footprint and is optimised for low overheads and interference.

... **scalable**. AtroPOS is built to support tiled manycore processor architectures with large numbers of CPU cores.

... **tailorable**. The system is able to adapt itself to the needs of the application, both statically and dynamically.

... **versatile**. AtroPOS supports multi-model parallel programming and is fit to serve as a generic foundation for arbitrary concurrency platforms.

Although the proverbial free lunch may be over for good, well-designed concurrency platforms whose requirements are considered on all levels of the system-software stack – including the operating-system kernel – will continue to provide application developers with all the tools necessary to write efficient software that is able to harness the raw parallel processing power of hardware generations to come.
Appendices
This is a list of all system calls currently implemented in the AtroPOS kernel.

By default, simple system calls have synchronous call semantics and extended system calls have asynchronous semantics – but there are a number of exceptions. Asynchronous simple functions have an asynchronous interface but block in the kernel using a continuation instead of switching stacks. Synchronous extended functions behave synchronously, but they may clobber the set of non-volatile registers, thus requiring the caller to save and restore these registers. A special case is claim_idle(), which has synchronous semantics per se but does block in the kernel in the sense that it brings the CPU core into a sleep state of indefinite duration.

<table>
<thead>
<tr>
<th>Name</th>
<th>Call semantics</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>claim_construct_with_cores()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_construct_with_core_ids()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_destroy()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_startup_subordinate_cores()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_add_cores()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_add_core_ids()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_prepare_destroy()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_prepare_remove_core_ids()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_remove_own_core()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_construct_on_tile_with_cores()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_construct_on_tile_with_core_ids()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_add_cores_on_tile()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_send_message()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_push_data()</td>
<td>asynchronous</td>
<td>extended</td>
</tr>
<tr>
<td>claim_idle()</td>
<td>synchronous</td>
<td>extended</td>
</tr>
<tr>
<td>claim_notify()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_notify_self()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>claim_wake_core()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>gc_stop_world()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>gc_start_world()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>gc_is_world_stopped()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>krn_eth_set_ip_address()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>krn_eth_open()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>krn_eth_set_transport()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>krn_eth_close()</td>
<td>asynchronous</td>
<td>simple</td>
</tr>
<tr>
<td>krn_eth_receive()</td>
<td>asynchronous</td>
<td>extended</td>
</tr>
<tr>
<td>krn_eth_send()</td>
<td>asynchronous</td>
<td>extended</td>
</tr>
<tr>
<td>mem_get_total_page_count()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>mem_map()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
</tbody>
</table>
## A List of System Calls

<table>
<thead>
<tr>
<th>Name</th>
<th>Call semantics</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>mem_map_aligned()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>mem_map_grow()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>mem_unmap()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>mem_protect()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>octo_app_get_info()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>octo_app_protected()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>octo_app_unprotect()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>octo_app_protect()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>octo_app_privileged()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>separation_enable()</td>
<td>synchronous</td>
<td>extended</td>
</tr>
<tr>
<td>separation_disable()</td>
<td>synchronous</td>
<td>extended</td>
</tr>
<tr>
<td>shutdown()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
<tr>
<td>stdio_send_data()</td>
<td>synchronous</td>
<td>simple</td>
</tr>
</tbody>
</table>
Security Recommendations for Invasive Hardware Components

At the time the CIC and the invasive NoC adapter were designed, making these components efficiently available for use by applications in the face of privilege isolation was an afterthought. In this appendix, a number of recommendations are collected that, if implemented, could maintain the efficiency of the CIC and NoC while providing full isolation guarantees. The fundamental approach is the following: have the kernel expose all safe functionality to the runtime system in the form of user-accessible memory-mapped registers while preventing direct user access to unsafe functionality; also, carry out permission checking in hardware where possible.

B.1 Suggested CIC Modifications

The CIC already gives every CPU core its own set of input and output registers for enqueuing and dequeuing i-lets. They should be laid out in the address space in such a way that each pair of input and output registers resides in a separate page, making it possible for the kernel to make only those pages accessible to an application that correspond to the cores staked by that application's elementary claims. The control registers that allow the CIC's lookup tables to be programmed should also be placed in a dedicated page, which would then not be exposed to user space, permitting only the kernel to change the CIC's configuration.

The destination claim of an i-let is specified by the claim-tag field in its control word. When the i-let is written into one of the CIC's input ports, the CIC's mapping logic evaluates that control word and determines a destination core based on the claim tag. To prevent an application from illegally injecting i-lets into an elementary claim that is owned by another application, a permission check should be added into the mapping logic. For example, the kernel could advise the CIC with the help of a bitmap for each input port which destination claim tags are allowed and which ones are forbidden. Attempts to schedule an i-let for a claim without permission should be rejected by the CIC.

B.2 Recommendations for the NoC Adapter

The situation is somewhat more complex with respect to the NoC. The network adapter of a tile has one input port per functional unit: one for remote i-lets, one for push data transfers and one for sys-i-lets. The former two operations are intended for application-level communication; the latter serves as a basis for RPCs between kernel instances and must not be exposed to user space, so it should be mapped into a separate page.
Since the network adapter accepts requests from all cores of the local tile on the same input port, it is harder to determine the identity of the sending claim – and with it the identity of the sending application, which is essential for assessing in hardware whether the operation is legal. The network adapter could pinpoint the sending core by examining the HMASTER signal, which indicates the bus master that is performing the transfer on the AMBA bus [ARM15]. If permission checking is to be done in hardware, this is best achieved through cooperation between the sending and the receiving side: the source network adapter could determine the sender’s application ID and attach it as metadata to the packets that travel over the NoC; the destination adapter could check whether that application is really allowed to infect the given elementary claim. This way, the information as to who is allowed to communicate with whom could be held locally and would not have to be distributed across all tiles. Note that hardware-based permission checking would require the network adapter to perform extensive lookups operating-system-managed data structures that map the relationship between cores, elementary claims and applications on the local tile. Secure push data transfers would additionally require the integration of a kind of I/O MMU or MPU to prevent writing to illegal addresses.

As a pragmatic alternative to the above suggestions, the network adapter can be left unmodified and the permission checks can be carried out in software by the operating-system kernel. This requires hiding the network adapter’s control registers from user space and exposing the NoC operations exclusively in the form of system calls. This way, the kernel remains in charge of performing all security checks. Based on the observations made in Chapter 7, it can be expected that such an approach is efficient enough for most use cases.
This appendix contains detailed aggregated time-measurement results that would have taken up too much space in Chapter 7. The results shown here for the execution times are the arithmetic means, rounded to four significant digits.

The following listing shows which of the subsequent result tables refer to which section of the evaluation chapter.

<table>
<thead>
<tr>
<th>Section</th>
<th>Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2 Memory Footprint</td>
<td>C.1</td>
</tr>
<tr>
<td>7.3 Micro-Benchmarks</td>
<td></td>
</tr>
<tr>
<td>7.3.1 Scheduling Operations on the User Level</td>
<td>C.2, C.3</td>
</tr>
<tr>
<td>7.3.2 Synchronisation Primitives</td>
<td>C.4, C.5, C.6, C.7</td>
</tr>
<tr>
<td>7.3.3 System Calls</td>
<td>C.8</td>
</tr>
<tr>
<td>7.3.4 Communication Operations</td>
<td>C.9</td>
</tr>
<tr>
<td>7.4 Application Benchmarks</td>
<td></td>
</tr>
<tr>
<td>7.4.1 NAS Parallel Benchmarks</td>
<td>C.10</td>
</tr>
<tr>
<td>7.4.2 IMSuite</td>
<td>C.11</td>
</tr>
<tr>
<td>7.4.3 Geometric Multigrid Solver</td>
<td>C.12</td>
</tr>
<tr>
<td>7.4.4 Cilk Plus Benchmarks</td>
<td>C.13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OctoPOS</th>
<th>AtroPOS + InvRT</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>388 494</td>
</tr>
<tr>
<td>syscall stubs</td>
<td>—</td>
</tr>
<tr>
<td>libc</td>
<td>137 015</td>
</tr>
<tr>
<td>malloc</td>
<td>59 538</td>
</tr>
<tr>
<td>runtime system</td>
<td>—</td>
</tr>
</tbody>
</table>

Table C.1: Text-section size comparison.
Results are given in bytes. Visualisation: Figure 7.1 on page 145.
<table>
<thead>
<tr>
<th></th>
<th>dispatch latency</th>
<th>self-infect overhead</th>
<th>self-infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>17.74</td>
<td>185.9</td>
<td>203.2</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>26.45</td>
<td>225.8</td>
<td>258.7</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>105.8</td>
<td>42.54</td>
<td>152.1</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>dispatch latency</th>
<th>self-infect overhead</th>
<th>self-infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>51.12</td>
<td>752.4</td>
<td>791.9</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>40.42</td>
<td>697.9</td>
<td>770.4</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>109.2</td>
<td>23.25</td>
<td>142.5</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>dispatch latency</th>
<th>self-infect overhead</th>
<th>self-infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>54.63</td>
<td>855.6</td>
<td>887.9</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>44.96</td>
<td>865.5</td>
<td>957.6</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>96.25</td>
<td>32.00</td>
<td>134.1</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

<table>
<thead>
<tr>
<th></th>
<th>dispatch latency</th>
<th>self-infect overhead</th>
<th>self-infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>58.10</td>
<td>322.3</td>
<td>352.4</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>48.61</td>
<td>342.5</td>
<td>379.8</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>97.23</td>
<td>39.66</td>
<td>132.9</td>
</tr>
</tbody>
</table>

(d) Results on Intel8.

Table C.2: Overheads and latencies of core-local scheduling and dispatching operations.
Results are given in clock cycles. Visualisation: Figure 7.2 on page 148.
<table>
<thead>
<tr>
<th></th>
<th>infect overhead</th>
<th>infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>1058</td>
<td>1751</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>1053</td>
<td>1947</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>1161</td>
<td>2270</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>infect overhead</th>
<th>infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>988.5</td>
<td>1472</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>995.1</td>
<td>1544</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>402.0</td>
<td>928.7</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>infect overhead</th>
<th>infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>1127</td>
<td>1694</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>1160</td>
<td>1798</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>526.3</td>
<td>1151</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

<table>
<thead>
<tr>
<th></th>
<th>infect overhead</th>
<th>infect latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>512.2</td>
<td>908.3</td>
</tr>
<tr>
<td>InvRT (sharing)</td>
<td>542.4</td>
<td>972.3</td>
</tr>
<tr>
<td>InvRT (stealing)</td>
<td>401.1</td>
<td>756.2</td>
</tr>
</tbody>
</table>

(d) Results on Intel8.

**Table C.3: Overheads and latencies of scheduling work to another core.**
Results are given in clock cycles. Visualisation: Figure 7.3 on page 150.

<table>
<thead>
<tr>
<th></th>
<th>AMD48n</th>
<th>Intel80n</th>
<th>Intel96n</th>
<th>Intel8</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>238.4</td>
<td>225.4</td>
<td>215.3</td>
<td>233.6</td>
</tr>
<tr>
<td>counting</td>
<td>219.0</td>
<td>226.8</td>
<td>226.2</td>
<td>220.4</td>
</tr>
<tr>
<td>binary</td>
<td>220.1</td>
<td>228.9</td>
<td>239.1</td>
<td>207.1</td>
</tr>
</tbody>
</table>

**Table C.4: Latencies of the wait() operation.**
Results are given in clock cycles. Visualisation: Figure 7.5 on page 153.
### Evaluation Results

#### Table C.5: Latencies of wakeup/spawn on the same core.

Results are given in clock cycles. Visualisation: Figure 7.6 on page 154.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>193.6</td>
<td>—</td>
<td>61.48</td>
<td>—</td>
</tr>
<tr>
<td>infect</td>
<td>—</td>
<td>167.6</td>
<td>—</td>
<td>42.83</td>
</tr>
<tr>
<td>counting</td>
<td>192.6</td>
<td>168.9</td>
<td>63.92</td>
<td>44.73</td>
</tr>
<tr>
<td>binary</td>
<td>166.8</td>
<td>142.1</td>
<td>32.80</td>
<td>18.48</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>171.3</td>
<td>—</td>
<td>54.01</td>
<td>—</td>
</tr>
<tr>
<td>infect</td>
<td>—</td>
<td>147.3</td>
<td>—</td>
<td>29.99</td>
</tr>
<tr>
<td>counting</td>
<td>168.1</td>
<td>148.1</td>
<td>57.40</td>
<td>29.00</td>
</tr>
<tr>
<td>binary</td>
<td>156.2</td>
<td>139.2</td>
<td>47.68</td>
<td>7.032</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>169.8</td>
<td>—</td>
<td>56.52</td>
<td>—</td>
</tr>
<tr>
<td>infect</td>
<td>—</td>
<td>135.6</td>
<td>—</td>
<td>33.90</td>
</tr>
<tr>
<td>counting</td>
<td>171.1</td>
<td>138.2</td>
<td>61.34</td>
<td>35.66</td>
</tr>
<tr>
<td>binary</td>
<td>159.9</td>
<td>127.5</td>
<td>52.49</td>
<td>12.00</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>170.5</td>
<td>—</td>
<td>62.41</td>
<td>—</td>
</tr>
<tr>
<td>infect</td>
<td>—</td>
<td>135.8</td>
<td>—</td>
<td>38.89</td>
</tr>
<tr>
<td>counting</td>
<td>169.7</td>
<td>138.1</td>
<td>61.08</td>
<td>38.21</td>
</tr>
<tr>
<td>binary</td>
<td>160.5</td>
<td>127.2</td>
<td>53.77</td>
<td>18.22</td>
</tr>
</tbody>
</table>

(d) Results on Intel8.
Table C.6: Latencies of wakeup/spawn on another core.
Results are given in clock cycles. Visualisation: Figure 7.7 on page 155.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>simple</strong></td>
<td>3042</td>
<td>—</td>
<td>3046</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>infect</strong></td>
<td>—</td>
<td>3003</td>
<td>—</td>
<td>2860</td>
</tr>
<tr>
<td><strong>counting</strong></td>
<td>3089</td>
<td>3081</td>
<td>3065</td>
<td>2901</td>
</tr>
<tr>
<td><strong>binary</strong></td>
<td>2736</td>
<td>2754</td>
<td>2740</td>
<td>2748</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>simple</strong></td>
<td>1436</td>
<td>—</td>
<td>1401</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>infect</strong></td>
<td>—</td>
<td>1270</td>
<td>—</td>
<td>1244</td>
</tr>
<tr>
<td><strong>counting</strong></td>
<td>1433</td>
<td>1383</td>
<td>1406</td>
<td>1395</td>
</tr>
<tr>
<td><strong>binary</strong></td>
<td>1433</td>
<td>1346</td>
<td>1447</td>
<td>1417</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>simple</strong></td>
<td>1479</td>
<td>—</td>
<td>1454</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>infect</strong></td>
<td>—</td>
<td>1360</td>
<td>—</td>
<td>1377</td>
</tr>
<tr>
<td><strong>counting</strong></td>
<td>1433</td>
<td>1443</td>
<td>1414</td>
<td>1500</td>
</tr>
<tr>
<td><strong>binary</strong></td>
<td>1417</td>
<td>1451</td>
<td>1462</td>
<td>1463</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

<table>
<thead>
<tr>
<th></th>
<th>wakeup</th>
<th>spawn</th>
<th>wakeup, handoff</th>
<th>spawn, handoff</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>simple</strong></td>
<td>1111</td>
<td>—</td>
<td>1124</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>infect</strong></td>
<td>—</td>
<td>974.6</td>
<td>—</td>
<td>976.7</td>
</tr>
<tr>
<td><strong>counting</strong></td>
<td>1113</td>
<td>1055</td>
<td>1112</td>
<td>1090</td>
</tr>
<tr>
<td><strong>binary</strong></td>
<td>1108</td>
<td>1055</td>
<td>1123</td>
<td>1094</td>
</tr>
</tbody>
</table>

(d) Results on Intel8.
## Evaluation Results

<table>
<thead>
<tr>
<th></th>
<th>$\nu &gt; 1$</th>
<th>$\nu = 1$, no wakeup/spawn</th>
<th>$\nu = 1$, wakeup/spawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>30.77</td>
<td>33.51</td>
<td>65.59</td>
</tr>
<tr>
<td>infect</td>
<td>29.34</td>
<td></td>
<td>64.35</td>
</tr>
<tr>
<td>counting</td>
<td>27.50</td>
<td>53.25</td>
<td>68.47</td>
</tr>
<tr>
<td>binary</td>
<td></td>
<td>27.96</td>
<td>39.33</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>$\nu &gt; 1$</th>
<th>$\nu = 1$, no wakeup/spawn</th>
<th>$\nu = 1$, wakeup/spawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>30.96</td>
<td>32.00</td>
<td>32.00</td>
</tr>
<tr>
<td>infect</td>
<td>28.00</td>
<td></td>
<td>29.56</td>
</tr>
<tr>
<td>counting</td>
<td>28.87</td>
<td>49.03</td>
<td>29.76</td>
</tr>
<tr>
<td>binary</td>
<td></td>
<td>28.97</td>
<td>18.98</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>$\nu &gt; 1$</th>
<th>$\nu = 1$, no wakeup/spawn</th>
<th>$\nu = 1$, wakeup/spawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>20.97</td>
<td>21.98</td>
<td>36.31</td>
</tr>
<tr>
<td>infect</td>
<td>18.96</td>
<td></td>
<td>39.00</td>
</tr>
<tr>
<td>counting</td>
<td>19.05</td>
<td>37.95</td>
<td>38.08</td>
</tr>
<tr>
<td>binary</td>
<td></td>
<td>20.00</td>
<td>28.96</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

<table>
<thead>
<tr>
<th></th>
<th>$\nu &gt; 1$</th>
<th>$\nu = 1$, no wakeup/spawn</th>
<th>$\nu = 1$, wakeup/spawn</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>26.48</td>
<td>27.12</td>
<td>42.10</td>
</tr>
<tr>
<td>infect</td>
<td>25.06</td>
<td></td>
<td>45.72</td>
</tr>
<tr>
<td>counting</td>
<td>24.22</td>
<td>42.19</td>
<td>43.36</td>
</tr>
<tr>
<td>binary</td>
<td></td>
<td>24.86</td>
<td>34.23</td>
</tr>
</tbody>
</table>

(d) Results on Intel8.

Table C.7: Overheads of the signal() operation.
Results are given in clock cycles. Visualisation: Figure 7.8 on page 156.
<table>
<thead>
<tr>
<th></th>
<th>Privilege isolation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>statically on</td>
<td>dynamically on</td>
</tr>
<tr>
<td>Entry latency</td>
<td>simple</td>
<td>53.16</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>55.86</td>
</tr>
<tr>
<td>Syscall overhead</td>
<td>simple</td>
<td>122.3</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>120.6</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>Privilege isolation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>statically on</td>
<td>dynamically on</td>
</tr>
<tr>
<td>Entry latency</td>
<td>simple</td>
<td>58.88</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>61.04</td>
</tr>
<tr>
<td>Syscall overhead</td>
<td>simple</td>
<td>101.8</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>104.0</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>Privilege isolation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>statically on</td>
<td>dynamically on</td>
</tr>
<tr>
<td>Entry latency</td>
<td>simple</td>
<td>57.02</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>59.04</td>
</tr>
<tr>
<td>Syscall overhead</td>
<td>simple</td>
<td>113.0</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>116.0</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

<table>
<thead>
<tr>
<th></th>
<th>Privilege isolation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>statically on</td>
<td>dynamically on</td>
</tr>
<tr>
<td>Entry latency</td>
<td>simple</td>
<td>60.09</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>61.81</td>
</tr>
<tr>
<td>Syscall overhead</td>
<td>simple</td>
<td>113.2</td>
</tr>
<tr>
<td></td>
<td>extended</td>
<td>116.0</td>
</tr>
</tbody>
</table>

(d) Results on Intel8.

**Table C.8: System-call overheads.**
Results are given in clock cycles. Visualisation: Figure 7.9 on page 159.
### Evaluation Results

<table>
<thead>
<tr>
<th></th>
<th>msg. overhead</th>
<th>msg. latency</th>
<th>push overhead</th>
<th>push latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>1577</td>
<td>3139</td>
<td>1614</td>
<td>3181</td>
</tr>
<tr>
<td>AtroPOS (bare)</td>
<td>1516</td>
<td>2854</td>
<td>2531</td>
<td>3154</td>
</tr>
<tr>
<td>AtroPOS (unisolated)</td>
<td>1525</td>
<td>2863</td>
<td>2738</td>
<td>3430</td>
</tr>
<tr>
<td>AtroPOS (isolated)</td>
<td>1890</td>
<td>3065</td>
<td>3487</td>
<td>4096</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>msg. overhead</th>
<th>msg. latency</th>
<th>push overhead</th>
<th>push latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>1405</td>
<td>3915</td>
<td>1462</td>
<td>4001</td>
</tr>
<tr>
<td>AtroPOS (bare)</td>
<td>1315</td>
<td>2101</td>
<td>1778</td>
<td>2199</td>
</tr>
<tr>
<td>AtroPOS (unisolated)</td>
<td>1373</td>
<td>2111</td>
<td>2543</td>
<td>3042</td>
</tr>
<tr>
<td>AtroPOS (isolated)</td>
<td>1779</td>
<td>2202</td>
<td>3505</td>
<td>3907</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>msg. overhead</th>
<th>msg. latency</th>
<th>push overhead</th>
<th>push latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>1347</td>
<td>4058</td>
<td>1396</td>
<td>4125</td>
</tr>
<tr>
<td>AtroPOS (bare)</td>
<td>909.0</td>
<td>1799</td>
<td>1282</td>
<td>2010</td>
</tr>
<tr>
<td>AtroPOS (unisolated)</td>
<td>894.8</td>
<td>1841</td>
<td>1942</td>
<td>2626</td>
</tr>
<tr>
<td>AtroPOS (isolated)</td>
<td>1236</td>
<td>1990</td>
<td>2749</td>
<td>3408</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

**Table C.9: Communication overheads and latencies.**

Results are given in clock cycles. Visualisation: Figure 7.10 on page 161.
<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>78.61 s</td>
<td>833.5 s</td>
<td>116.7 s</td>
<td>35.67 s</td>
<td>72.78 s</td>
<td>—</td>
<td>148.9 s</td>
<td>130.9 s</td>
</tr>
<tr>
<td>bare (work-sharing)</td>
<td>78.09 s</td>
<td>831.4 s</td>
<td>116.5 s</td>
<td>35.00 s</td>
<td>70.45 s</td>
<td>52.07 s</td>
<td>148.5 s</td>
<td>130.0 s</td>
</tr>
<tr>
<td>bare (work-stealing)</td>
<td>78.08 s</td>
<td>829.8 s</td>
<td>116.4 s</td>
<td>35.05 s</td>
<td>70.51 s</td>
<td>52.24 s</td>
<td>148.2 s</td>
<td>130.2 s</td>
</tr>
<tr>
<td>unisolated</td>
<td>78.12 s</td>
<td>831.4 s</td>
<td>116.6 s</td>
<td>35.39 s</td>
<td>70.48 s</td>
<td>52.47 s</td>
<td>149.0 s</td>
<td>130.2 s</td>
</tr>
<tr>
<td>isolated</td>
<td>78.19 s</td>
<td>827.8 s</td>
<td>116.6 s</td>
<td>35.82 s</td>
<td>70.66 s</td>
<td>52.96 s</td>
<td>155.5 s</td>
<td>130.4 s</td>
</tr>
</tbody>
</table>

(a) Results on AMD48n.

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>51.31 s</td>
<td>777.9 s</td>
<td>69.39 s</td>
<td>29.97 s</td>
<td>42.98 s</td>
<td>—</td>
<td>120.6 s</td>
<td>76.36 s</td>
</tr>
<tr>
<td>bare (work-sharing)</td>
<td>50.36 s</td>
<td>769.3 s</td>
<td>68.99 s</td>
<td>29.36 s</td>
<td>40.95 s</td>
<td>45.44 s</td>
<td>120.7 s</td>
<td>70.02 s</td>
</tr>
<tr>
<td>bare (work-stealing)</td>
<td>50.37 s</td>
<td>767.8 s</td>
<td>68.79 s</td>
<td>29.28 s</td>
<td>40.95 s</td>
<td>44.98 s</td>
<td>121.0 s</td>
<td>70.11 s</td>
</tr>
<tr>
<td>unisolated</td>
<td>50.39 s</td>
<td>768.3 s</td>
<td>68.86 s</td>
<td>29.42 s</td>
<td>41.05 s</td>
<td>45.18 s</td>
<td>121.0 s</td>
<td>70.12 s</td>
</tr>
<tr>
<td>isolated</td>
<td>50.37 s</td>
<td>779.0 s</td>
<td>68.86 s</td>
<td>29.31 s</td>
<td>41.10 s</td>
<td>45.40 s</td>
<td>120.0 s</td>
<td>69.99 s</td>
</tr>
</tbody>
</table>

(b) Results on Intel80n.

<table>
<thead>
<tr>
<th></th>
<th>BT</th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
<th>LU</th>
<th>MG</th>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>36.87 s</td>
<td>513.4 s</td>
<td>57.08 s</td>
<td>25.87 s</td>
<td>41.73 s</td>
<td>—</td>
<td>51.28 s</td>
<td>39.27 s</td>
</tr>
<tr>
<td>bare (work-sharing)</td>
<td>35.93 s</td>
<td>514.0 s</td>
<td>56.74 s</td>
<td>24.50 s</td>
<td>39.17 s</td>
<td>35.11 s</td>
<td>50.96 s</td>
<td>36.55 s</td>
</tr>
<tr>
<td>bare (work-stealing)</td>
<td>36.08 s</td>
<td>513.2 s</td>
<td>56.81 s</td>
<td>24.65 s</td>
<td>39.19 s</td>
<td>34.06 s</td>
<td>50.96 s</td>
<td>36.46 s</td>
</tr>
<tr>
<td>unisolated</td>
<td>36.06 s</td>
<td>513.3 s</td>
<td>56.49 s</td>
<td>24.68 s</td>
<td>39.36 s</td>
<td>34.31 s</td>
<td>50.99 s</td>
<td>36.62 s</td>
</tr>
<tr>
<td>isolated</td>
<td>36.13 s</td>
<td>501.0 s</td>
<td>56.48 s</td>
<td>24.98 s</td>
<td>39.44 s</td>
<td>34.72 s</td>
<td>51.10 s</td>
<td>36.68 s</td>
</tr>
</tbody>
</table>

(c) Results on Intel96n.

Table C.10: Mean execution times of the NAS Parallel Benchmarks.

Visualisation: Figure 7.11 on page 166.

<table>
<thead>
<tr>
<th></th>
<th>BF</th>
<th>DST</th>
<th>BY</th>
<th>DR</th>
<th>DS</th>
<th>KC</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>70.13 ms</td>
<td>1.009 s</td>
<td>95.55 s</td>
<td>8.223 s</td>
<td>5.555 s</td>
<td>735.9 ms</td>
<td>—</td>
</tr>
<tr>
<td>bare (work-sharing)</td>
<td>64.99 ms</td>
<td>0.9998 s</td>
<td>85.73 s</td>
<td>7.697 s</td>
<td>5.055 s</td>
<td>683.8 ms</td>
<td>1.272 s</td>
</tr>
<tr>
<td>bare (work-stealing)</td>
<td>64.57 ms</td>
<td>0.9986 s</td>
<td>87.13 s</td>
<td>7.634 s</td>
<td>5.123 s</td>
<td>679.5 ms</td>
<td>1.268 s</td>
</tr>
<tr>
<td>unisolated</td>
<td>64.68 ms</td>
<td>0.9866 s</td>
<td>87.29 s</td>
<td>7.647 s</td>
<td>5.134 s</td>
<td>679.6 ms</td>
<td>1.269 s</td>
</tr>
<tr>
<td>isolated</td>
<td>65.20 ms</td>
<td>0.9872 s</td>
<td>87.97 s</td>
<td>7.681 s</td>
<td>5.194 s</td>
<td>685.0 ms</td>
<td>1.276 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>HS</th>
<th>LCR</th>
<th>MIS</th>
<th>MST</th>
<th>VC</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>13.09 s</td>
<td>4.814 s</td>
<td>330.4 ms</td>
<td>25.55 s</td>
<td>51.18 ms</td>
</tr>
<tr>
<td>bare (work-sharing)</td>
<td>12.14 s</td>
<td>4.476 s</td>
<td>306.7 ms</td>
<td>25.06 s</td>
<td>47.64 ms</td>
</tr>
<tr>
<td>bare (work-stealing)</td>
<td>12.16 s</td>
<td>4.458 s</td>
<td>317.2 ms</td>
<td>25.02 s</td>
<td>47.38 ms</td>
</tr>
<tr>
<td>unisolated</td>
<td>12.16 s</td>
<td>4.416 s</td>
<td>319.7 ms</td>
<td>25.01 s</td>
<td>46.95 ms</td>
</tr>
<tr>
<td>isolated</td>
<td>12.24 s</td>
<td>4.451 s</td>
<td>319.4 ms</td>
<td>25.03 s</td>
<td>47.64 ms</td>
</tr>
</tbody>
</table>

Table C.11: Mean execution times of IMSuite on Intel8.

Visualisation: Figure 7.12 on page 170.
### Table C.12: Mean execution times of the geometric multigrid solver.

<table>
<thead>
<tr>
<th></th>
<th>AMD48n</th>
<th>Intel80n</th>
<th>Intel96n</th>
<th>Intel8</th>
</tr>
</thead>
<tbody>
<tr>
<td>OctoPOS</td>
<td>211.7 s</td>
<td>133.6 s</td>
<td>114.7 s</td>
<td>162.5 s</td>
</tr>
<tr>
<td>bare (work-sharing)</td>
<td>211.4 s</td>
<td>130.7 s</td>
<td>113.1 s</td>
<td>160.6 s</td>
</tr>
<tr>
<td>bare (work-stealing)</td>
<td>171.4 s</td>
<td>129.4 s</td>
<td>111.7 s</td>
<td>159.6 s</td>
</tr>
<tr>
<td>isolated</td>
<td>168.0 s</td>
<td>127.6 s</td>
<td>110.2 s</td>
<td>160.1 s</td>
</tr>
<tr>
<td>isolated</td>
<td>169.2 s</td>
<td>128.9 s</td>
<td>111.3 s</td>
<td>160.3 s</td>
</tr>
</tbody>
</table>

Visualisation: Figure 7.13 on page 171.

### Table C.13: Mean execution times of the Cilk Plus application benchmarks on Intel8.

<table>
<thead>
<tr>
<th></th>
<th>BL</th>
<th>BS</th>
<th>FI</th>
<th>KA</th>
<th>MA</th>
<th>MC</th>
<th>MS</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>261.8 ms</td>
<td>15.85 s</td>
<td>944.0 ms</td>
<td>2.893 s</td>
<td>3.579 s</td>
<td>619.0 ms</td>
<td>3.207 s</td>
<td>135.1 ms</td>
</tr>
<tr>
<td>AtroPOS</td>
<td>191.2 ms</td>
<td>16.51 s</td>
<td>937.0 ms</td>
<td>3.129 s</td>
<td>3.532 s</td>
<td>459.2 ms</td>
<td>3.298 s</td>
<td>135.6 ms</td>
</tr>
</tbody>
</table>

Visualisation: Figure 7.14 on page 173.
<table>
<thead>
<tr>
<th>Acronyms and Abbreviations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
</tr>
<tr>
<td>API</td>
</tr>
<tr>
<td>APIC</td>
</tr>
<tr>
<td>ASLR</td>
</tr>
<tr>
<td>AST</td>
</tr>
<tr>
<td>CAS</td>
</tr>
<tr>
<td>COTS</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>CSP</td>
</tr>
<tr>
<td>DAG</td>
</tr>
<tr>
<td>DDR</td>
</tr>
<tr>
<td>deque</td>
</tr>
<tr>
<td>DMA</td>
</tr>
<tr>
<td>ELF</td>
</tr>
<tr>
<td>FIFO</td>
</tr>
<tr>
<td>HAL</td>
</tr>
<tr>
<td>HPC</td>
</tr>
<tr>
<td>I/O</td>
</tr>
<tr>
<td>ID</td>
</tr>
<tr>
<td>IPC</td>
</tr>
<tr>
<td>IPI</td>
</tr>
<tr>
<td>IR</td>
</tr>
<tr>
<td>IRQ</td>
</tr>
<tr>
<td>ISA</td>
</tr>
<tr>
<td>JVM</td>
</tr>
<tr>
<td>KPTI</td>
</tr>
<tr>
<td>libc</td>
</tr>
<tr>
<td>LIFO</td>
</tr>
<tr>
<td>LL/SC</td>
</tr>
<tr>
<td>MCP</td>
</tr>
<tr>
<td>MIMD</td>
</tr>
<tr>
<td>MMU</td>
</tr>
<tr>
<td>MPI</td>
</tr>
<tr>
<td>MPMC</td>
</tr>
<tr>
<td>MPSC</td>
</tr>
<tr>
<td>MPU</td>
</tr>
<tr>
<td>NMI</td>
</tr>
<tr>
<td>NoC</td>
</tr>
<tr>
<td>NUMA</td>
</tr>
<tr>
<td>PGAS</td>
</tr>
<tr>
<td>RAIL</td>
</tr>
<tr>
<td>RAM</td>
</tr>
<tr>
<td>RPC</td>
</tr>
<tr>
<td>schedel</td>
</tr>
<tr>
<td>SIMD</td>
</tr>
<tr>
<td>SPSC</td>
</tr>
<tr>
<td>STL</td>
</tr>
<tr>
<td>TCPA</td>
</tr>
<tr>
<td>TLB</td>
</tr>
<tr>
<td>UCQ</td>
</tr>
<tr>
<td>vDSO</td>
</tr>
<tr>
<td>vtable</td>
</tr>
</tbody>
</table>


Bibliography


[Hib76] Peter Hibbard. ‘Parallel Processing Facilities’. In: *New Directions in Algorithmic Languages*. Institut de Recherche d’Informatique et d’Automatique (IRIA), Sept. 1976, pages 1–7 (cited on page 46).


Bibliography


Bibliography


Bibliography


Bibliography


