Adaptive Address-Space Management for Resource-Aware Applications

—

Anpassungsfähige Adressraumverwaltung für Resourcengewahre Anwendungen

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

Doktor der Ingenieurwissenschaften
(Dr.-Ing.)

vorgelegt von

Gabor Drescher
## Contents

Abstract 1

Zusammenfassung 3

Acknowledgments 5

1 Introduction and Motivation 7
   1.1 Problems Related to Address-Space Management . . . . . . . . . . . 8
      1.1.1 Memory-Page Allocation and Locality . . . . . . . . . . . . . 8
      1.1.2 Memory Protection Overheads and Predictability . . . . . . . 9
      1.1.3 Memory Allocation within Parallel Applications . . . . . . . 10
   1.2 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
   1.3 Publications . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2 Background 15
   2.1 Invasive Computing . . . . . . . . . . . . . . . . . . . . . . . . . . 15
   2.2 Abstract Hardware Model . . . . . . . . . . . . . . . . . . . . . . . 16
   2.3 Invasive Run-Time Support System . . . . . . . . . . . . . . . . . . 17
      2.3.1 Invasive Concepts and Operations in OctoPOS . . . . . . . . 17
      2.3.2 Invasive Concepts in the Agent System . . . . . . . . . . . . 19
   2.4 Software- and Hardware-Architecture Overview . . . . . . . . . . . . 19

3 Adaptive Address-Space Design 21
   3.1 Application Model . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
      3.1.1 Address-Space Layout . . . . . . . . . . . . . . . . . . . . . . 22
      3.1.2 iRTSS and System-Call Interface . . . . . . . . . . . . . . . . 23
   3.2 Memory Allocation . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
      3.2.1 Block-Allocation Requirements . . . . . . . . . . . . . . . . . 29
      3.2.2 Garbage-Collector Requirements . . . . . . . . . . . . . . . . 31
   3.3 Adaptive Memory Protection . . . . . . . . . . . . . . . . . . . . . 32
      3.3.1 Switching Protection Modes . . . . . . . . . . . . . . . . . . . 35
   3.4 Adaptive Inter-Tile Communication . . . . . . . . . . . . . . . . . . 35
   3.5 Strategies for Lifting Address-Space Isolation . . . . . . . . . . . . . 39

4 Adaptive Address-Space Implementation 41
   4.1 Invasive Hardware-Model Implementations . . . . . . . . . . . . . . . 41
      4.1.1 Page-Table Implementation and Abstraction . . . . . . . . . 42
      4.1.2 Invasive SPARC-Leon Architecture . . . . . . . . . . . . . . . 44
      4.1.3 x86-64 Support . . . . . . . . . . . . . . . . . . . . . . . . . . 45
      4.1.4 Linux Guest Support . . . . . . . . . . . . . . . . . . . . . . . 47
A Appendix

A.1 Source Code
  A.1.1 Tree Block Allocator
  A.1.2 Guarded Sections
  A.1.3 MPI Barrier

A.2 Evaluation Data
  A.2.1 Microbenchmarks for the Intel8 Variant
  A.2.2 Linux Performance
  A.2.3 Red-Black Tree vs Sorted List
  A.2.4 X10 Benchmarks for Intel8

A.3 Embedded Resources

Bibliography
This thesis presents adaptive memory protection, isolation and management techniques that were developed in the Collaborative Research Centre (CRC) Invasive Computing. These techniques were deployed in the operating system OctoPOS which is part of the CRC. The aim was to isolate concurrent applications on multi-core and multi-tile hardware architectures. Memory protection is an integral part of many operating systems to ensure safety and security. However, significant runtime overheads and resources may be involved in the enforcement of memory protection. This is especially the case for parallel applications that run on multiple cores and multiple tiles. Overheads include the management of page tables, TLB consistency and synchronisation.

For trusted applications or applications written in type-safe programming languages with a trusted runtime environment, these overheads and resources are completely in vain. This is in contrast to the intention of Invasive Computing, which is to optimise the usage of computing platforms by promoting resource awareness. Adaptive memory protection in OctoPOS achieves this resource awareness by being able to lift application isolation at runtime and to establish said isolation again as circumstances demand. Furthermore, resource-aware programs are promoted that often release and acquire resources from the operating system. As the evaluations show, unrestricted applications can benefit from less runtime jitter, faster system calls and improved predictability. The adaptivity itself seems not to harm the general performance of the tested operating system variants.

Subproblems were tackled that arise from multi-core architectures and shared memory applications. Inter-processor interrupts must be sent and synchronously acknowledged to manage TLB entries. For this problem an innovative acknowledged-multicast algorithm was developed that outperforms existing solutions. Management of physical memory is also deeply coupled with memory protection. Novel solutions to manage physical memory from page frames up to user-space memory allocation are presented, yielding low fragmentation, high performance and good predictability properties. Synchronisation of interrelated system calls that deal with resource management are handled using Guarded Sections. These allow asynchronous progression of activities while a critical section is blocked. Adaptive memory protection is also applied to an asynchronous message-passing mechanism utilising hardware DMA. In the prototype systems the DMA engine is not restricted by any IO-MMU, yet it is controlled by applications for communication. An adaptive protection mechanism was established in OctoPOS to restrict DMA operations to be contained within an application that can spread over multiple tiles dynamically. This isolation mechanism is also able to lift and establish the protection at runtime, all while concurrently serving DMA requests.
Zusammenfassung


Acknowledgments

In my time at the chair, I learned a lot about scientific research, programming, debugging, hardware, business-trip accounting, large projects, general work life, the universe and I acquired a lot of other worldly wisdom. I am very grateful for that time of my life. I have also come to know my colleagues, many of whom I call friends, now. Without these friends I would lack the endurance, knowledge and skill to write this thesis. I cannot go into details about everybody as this would exceed my page limit by far. For this reason I list their names in alphabetical order and hereby invite them to have a drink with me once we should meet after completion of my thesis and once the contemporary pandemic is over.

I thank:


Special thanks to Wolfgang Schröder-Preikschat (wosch), my doctorate supervisor, who allowed me to work and study at his chair and who helped me with his advice and support in scientific and non-scientific issues. Also, special thanks to Jürgen Kleinöder for his administrative ambition and transparent resource scheduling.

Last but not least, I thank my wife for her enduring support and encouragement.

Erlangen, 2021
Introduction and Motivation

The appearance of multi-core shared-memory computers was promoted by the hope that parallel computations could be accelerated with less overhead than that of separated single-core machines connected over a network. Unfortunately, this acceleration came not for free but existing software had to be rewritten and rethought. This is aggravated by the fact that the number of processing units still grows and machines with thousands of cores are within sight. Today, the computer science community is still in the process of reworking existing software and algorithms to utilise the parallel hardware efficiently.

This thesis focuses on certain aspects of address-space management within an operating system. In every computing system the operating system is a central component that orchestrates computing resources and dispatches them to the applications. In return, requests to the operating system also happen in parallel. For this reason it is especially important that the operating system is not only aware of multi-core architectures but is also itself subject to optimisations regarding parallel scalability.

A prominent example of this kind of optimisation is the “Big Kernel Lock” that was introduced in the Linux kernel in 1996 to support multi-core computers. Every subsystem of the kernel was protected by the same lock. Performance drops were noticeable even on dual-core computers. It took 15 years until 2011 the last parts of the lock were replaced by fine-grained locks or other means of synchronisation [1]. This shows the amount of effort and time scale that is sometimes needed to restructure existing software to benefit from shared memory parallelism.

Address-space management is also subject to scalability problems when the number of cores and application parallelism increases. There are two kinds of parallelism that are in the focus of this work. The first kind is that of multiple processes executing in their own isolated address space. Although, these processes are isolated with regards to their logical memory regions and may be single-threaded, they still access a global operating system and may cause concurrent execution within it.
Chapter 1 Introduction and Motivation

The second type of parallelism is within such a process and its address space. Commonly this is called shared memory parallelism or thread parallelism. As with multiple processes these threads may enter the operating system concurrently. Because they share parts of their address space, matters become worse in the operating system when some part of this address space should be modified.

Section 1.1 discusses these problems, its partial aspects and key concepts of the approaches taken to overcome the presented issues. An overview of the whole thesis is given in Section 1.2. Section 1.3 lists the author’s papers and highlights the contributions to them.

1.1 Problems Related to Address-Space Management

Address-space management in operating systems is subject to multiple problems when dealing with parallel applications and multi-core machines. Unfortunately, the obvious synchronisation problems are just one facet of this problem set. The following sections shed light on three aspects of address-space management.

- Physical-page allocation on NUMA architectures and related scheduling problems.
- Memory protection for multithreaded applications.
- Memory allocation, fragmentation and caching within an application.

All of these problems are tackled in some way by this thesis. To describe the state of the art problems related to address-space management, terms from the current POSIX specification [2] are used and Linux as a mostly POSIX compliant system is taken as an example.

1.1.1 Memory-Page Allocation and Locality

Memory access is often the bottleneck for computations, especially memory-intensive computations. The reason is that memory chips have high latency compared to CPU cores and constrained throughput, see [3]. Throughput becomes a problem when multiple cores need to access a lot of memory. This can be already the case with under ten cores, not to mention hundreds or thousands of cores. For this reason modern computing systems employ a non-uniform memory architecture (NUMA). Figure 1.1 illustrates an example NUMA architecture. Multiple physical memory modules exist in the system, each of them form a NUMA domain or NUMA node. CPU cores are local to one of these domains and respectively to the memory. Memory access to the local memory has low latency and maximum throughput, whereas access to remote memory must pass an additional interconnect which causes higher latency and sharing of memory throughput with the remote cores. Shared memory properties of cache coherence and memory consistency are generally unaffected by the NUMA architecture.
Software may therefore ignore the underlying hardware architecture and still work correctly. However, to work efficiently, parallel applications and the operating system must be aware of the NUMA architecture and distribute their memory accordingly.

![Figure 1.1: Example for the relationship of NUMA domains, memories and CPU cores.](image)

Typically, automatic strategies are carried out by operating systems like Linux to decide where to allocate physical memory for an application. Scheduling must be aware of these allocation strategies and assign appropriate affinities to the threads. To accommodate for more threads than cores in a NUMA domain, Linux employs a first-touch strategy [4]. Memory allocated from the operating system is mapped read-only into the address space of the process. When the first thread writes to such a memory page, the physical memory is allocated from the current NUMA domain and the thread’s affinity is associated with that domain. Such an automatic and heuristic memory placement is of course not optimal and the application can be given fine-grained control over memory placement. This, of course, places another burden on programmers seeking to improve the performance or predictability of their programs.

NUMA architectures increase the complexity of memory allocation in the operating system but also in applications when programming for performance or predictability. This thesis explores a different approach that tackles these problems introduced by NUMA architectures in a way, that simplifies memory management for both, operating system and user-space applications.

### 1.1.2 Memory Protection Overheads and Predictability

Memory protection based on memory management units (MMU) causes a range of overheads for operating systems and user-space programs. Context switches are more expensive when switching from one address space to another.
Chapter 1 Introduction and Motivation

Depending on the hardware architecture, the translation caches, so called translation lookahead buffers (TLB), or other caches must be flushed, new entries have to be fetched and the MMU must be reprogrammed.

Memory allocation from the operating system is slower as page tables must be adjusted. This may even cause further allocations on hierarchical page tables. When the memory allocation to the user is lazy and copy-on-write mechanisms are utilised, further unexpected overheads may be caused in the application on memory access.

Finally, reducing permissions on memory ranges by using one of the POSIX functions `munmap` or `mprotect` is defined to be synchronous. This implies that the page-table entries must be adjusted and TLB entries be invalidated immediately, before returning from these functions. For multithreaded programs running on multiple cores the operating system must visit (via inter-processor interrupts) each core and invalidate the respective memory range from the TLB of the core. This process is commonly referred to as “TLB shootdown” and may greatly hinder scalability on many cores [5]. Even on architectures that feature TLB shootdowns in hardware, like ARM 64 bit [6], it is still a costly operation.

All these overheads influence the timing predicability of applications in a negative way. Context switches may take longer depending on the actual source or destination context. Memory allocation may be slower or faster depending on whether a sub hierarchy of page-tables must be allocated and initialised in the operating system. Even memory access may have different timings if copy-on-write was used. TLB shootdowns in particular cause unpredictable delays in uninvolved threads (of the same application) at random points in time.

Adaptive address-space management is able to avoid these overheads and unpredictabilities by deploying invasive strategies and lifting memory protection for certain applications.

1.1.3 Memory Allocation within Parallel Applications

Typically the C library is responsible for the user-space memory allocation strategy that is implemented in `malloc` and `free` functions. Significant performance improvements were made in the last decades with special focus on parallel scalability and efficiency. These improvements partially originate from two optimisations. First, caching memory in thread-local storage gives each thread a local memory pool that reduces the need for synchronisation and on average increases the performance for the allocation functions. However, memory fragmentation may increase and adjacent memory regions are harder to find and merge. Second, reducing the interaction with the operating system to avoid the overheads mentioned in the previous Section. This is especially true for `munmap` that serves the purpose of returning memory to the operating system. While these optimisations are great for fast execution, they are wasteful in terms of memory resources.
Some allocation libraries like TCMalloc from Google even state “TCMalloc currently does not return any memory to the system” [7]. Which of course simplifies the implementation, but only really works if there is only one application per machine or unpredictable large overheads may be incurred by memory virtualisation. Invasive Computing promotes resource awareness to efficiently utilise a system from a global perspective. This means that it might cause overhead from an application’s point of view to return memory to the system, but it increases overall performance when the system in return does not have to swap out memory into secondary storage. Resource awareness is also important in embedded or constrained environments that have little memory. In such an environment wasteful allocators are unusable but performance may still be important. This thesis presents memory allocation algorithms for the operating system as well as user space that are fast, predictable, have low fragmentation and can be used in a resource aware way.

1.2 Overview

An overview of the upcoming chapters follows:

Chapter 2 gives background information on Invasive Computing and how this work is integrated into it. An overview of the abstract hardware and software architecture is given with special focus on iRTSS and the operating system OctoPOS. Relevant invasive concepts and operations are introduced that play an important role in the design and implementation of adaptive address-space management.

Chapter 3 presents the design of the adaptive address-space management system for OctoPOS. The abstract hardware structure and concepts from OctoPOS form the framework of the design. From this framework the design of an adaptive memory protection system and inter-tile communication system is presented. The concept of applications is defined that are subject to optional memory isolation.

Chapter 4 summarises the concrete implementation details of the invasive hardware model on contemporary architectures. In the course of this thesis, OctoPOS and its specific memory layout was additionally ported to x86-64 hardware and x86 32 bit software interfaces. The specifics of these ports are outlined in this Chapter. The tile-local implementation details include concrete hardware interfaces, data structures and algorithms for memory allocation, multicast communication and synchronisation. The inter-tile communication part of the implementation is also addressed. It consists of an adaptive access and caching protocol that enhances the operations of the raw DMA unit. Based on this enhanced DMA operation, the inner workings of a standard compliant Message Passing Interface (MPI) implementation on OctoPOS are described.

Chapter 5, the evaluation, consists of microbenchmarks for certain system calls as well as application benchmarks written in X10 and C/C++ languages and for OctoPOS and MPI interfaces. Special focus is on resource awareness and how fine-grained interaction with the operating system influences performance and predictability.
Chapter 1 Introduction and Motivation

Chapter 6 discusses the evaluation results, concludes with the contribution of this thesis and an outlook is given for future work particularly in the context of Invasive Computing. This Chapter includes references to related work and delimits this thesis from other work.

1.3 Publications

Ideas and results discussed in this thesis have been partly published beforehand:


The remainder of this section uses personal pronouns to clearly reveal the author’s contribution on scientific works.

I was the primary author for the section on adaptive memory protection within and between tiles for the article on providing security on demand using Invasive Computing in [8]. Main concepts and technical details for adaptive memory protection and evaluation results were presented in the Dagstuhl seminar [9]. I created the contents for this presentation, but it was delivered by Schröder-Preikschat.

Another aspect of physical memory protection is memory encryption. In [10] a standard Linux kernel was modified to encrypt applications’ memory transparently. The goal was to protect against physical memory-disclosure attacks. I contributed the evaluation implementation and the paper’s section on the evaluation.

The workshop paper [11] introduces a novel algorithm and implementation for synchronous multicast. This algorithm is one possible variant for IPI multicast in the implementation for this thesis. I contributed the section on related work and was responsible for the execution of evaluation scenarios.

Guarded Sections play an important role in synchronisation for this thesis. The conference paper [12] establishes Guarded Sections as a novel algorithm for shared-memory synchronisation. The ideas for Guarded Sections were further elaborated in the presentation [13] and in the technical report [14] together with parts of a wait-free operating system. I contributed ideas and algorithms for completely wait-free Guarded Sections and implemented multiple variants for the evaluations in the cited works.

In [15] we present concepts for energy and performance efficient operating systems on many-core computers. As a co-author, I contributed the parts on LAKE as well as the kernel implementation and evaluation.

The paper [16] sheds light on distributed shared-memory (DSM) systems in the many-core era and discusses possibilities for elementary operations to support DSM in an efficient and portable way. I contributed, as a co-author, to the parts on Invasive Computing and related work.

I made editorial contributions to various sections on all of these papers.
2 Background

2.1 Invasive Computing

This work is integrated into a larger research programme and the following sections
give an understanding about all relevant projects and key concepts that form the
basis for this thesis.

Invasive Computing (InvasiC) is a trans-regional Collaborative Research Centre
supported by the German Research Foundation (Deutsche Forschungsgemeinschaft)
under grant number SFB/TR 89. InvasiC pursues many goals that were also subject
to refinements and enhancements over the years. A concise overview of these goals is
presented in [17] and [18]. In this work, only a subset of the objectives are outlined,
that have relevance to the presented problems and proposed solutions. Also, only a
subset of the projects will be addressed, that have direct influence on this work.

The general idea of InvasiC is to promote resource awareness in all layers of a
computing system, especially applications, the operating system and hardware. The
intention is that this will lead to beneficial properties of the overall system. Examples
of such beneficial properties might be higher runtime performance, predicability or
power savings. Resource awareness in applications demands the exact knowledge
and exclusive use of the available resources. This can be seen in diametric opposition
to the strategy of resource virtualisation, where applications see nearly unlimited
virtual resources that are transparently scheduled between competing applications.
Therefore, to realise the invasive concept, a computing system must refrain from
virtualisation and instead cooperatively manage resources.

To support resource awareness, InvasiC proposes three operations: invade, infect
and retreat. Invade allocates computing resources to an invasive program based on
constraints. Infect starts parallel execution of parts of the program, once all required
computing resources are reserved. Retreat releases the resources that were acquired
using invade, when parallel execution finishes.
Chapter 2 Background

InvasiC tackles all aspects of computing to consequently implement an invasive system from ground up. These aspects are divided into the following projects:

- **Algorithms and Language:** Projects \( A1 \), \( A3 \) and \( A4 \) study the theory and basics of InvasiC, algorithmic patterns of invasive programs and address the theory of scheduling and load balancing.
- **Applications** stem from the fields of robotics and high performance computing, projects \( D1 \) and \( D3 \).
- **Compiler and Runtime:** Projects \( C2 \) and \( C3 \) develop compilers, runtime systems, evaluation and simulation frameworks.
- **Operating System:** The invasive run-time support system (iRTSS) developed by \( C1 \) provides abstractions and resource-management strategies to the applications and runtimes on top of it.
- **Hardware:** The hardware projects \( B1 \), \( B2 \), \( B3 \), \( B4 \), \( B5 \) design units that support invasive concepts at micro-architectural scale in processors, on-chip networks and the memory hierarchy.

The following sections describe concepts of InvasiC in a bottom-up manner, beginning with the hardware model, continuing with iRTSS and finishing with a general software overview.

2.2 Abstract Hardware Model

Invasive Computing anticipates future many-core computing systems as a grid of tiles, Figure 2.1 exemplifies such a grid. Each tile may contain either loosely-coupled processors, tightly-coupled co-processor arrays (TCPA) or input-output (I/O) devices. TCPA tiles are not considered in this thesis. Tiles containing loosely-coupled processors have random-access memory (RAM) and may have multiple CPU cores. I/O tiles may hold network devices and external RAM modules.

Memory local to a tile is called tile-local memory (TLM) and memory residing on the I/O tile is denoted as shared memory (SHM), as it is common to all tiles.

All tiles are connected via a network-on-chip (NoC) that permits native memory access from the CPU cores to remote TLM, SHM and memory-mapped devices on the I/O tile. The hardware model defines no cache coherence between tiles, therefore a concrete implementation of a NoC may not support it. The intention of this is that cache coherence may not scale well to hundreds of tiles and thousands of cores. In contrast, cache coherence is guaranteed within a tile.

The specified hardware model covers a lot of contemporary hardware architectures, that can be seen as specialisations. This is proved in Section 4.1 where the concrete implementations for specific architectures are summarised.
2.3 Invasive Run-Time Support System

The subject of this thesis originates from project C1 and the invasive run-time support system (iRTSS). As the implementation for this thesis is integrated into iRTSS, it is useful to introduce some of its characteristics. It can be divided into three parts: OctoPOS, the agent system and certain user-space libraries. OctoPOS is the operating system of InvasiC and the agent system is a resource management system that encapsulates decision strategies for resource assignment. User-space libraries such as the C-language standard library are implemented and modified such, that they make special use of the invasive concepts and OctoPOS. Many of these user-space libraries were designed and implemented for this thesis and will be discussed in Chapter 4.

2.3.1 Invasive Concepts and Operations in OctoPOS

Large parts of OctoPOS were developed by Oechslein, Schedel and Erhardt, see [19]–[22]. It is highly configurable for the needs of Invasive Computing and represents a family of operating systems. The configuration space is rather large and continuously changing, so only a subset of the possible behaviours and features are presented here, to give a comprehensible overview. Generally, OctoPOS started as a library operating system and therefore, did not provide isolation or privilege separation. This changed in certain configurations, as [22] introduced privilege separation and this work brings memory protection.
Chapter 2 Background

The functional interface presented here is uninvolved in these changes. OctoPOS’
arccxhitecture is that of a multi-kernel operating system. This means, that on each
tile of an invasive hardware platform, a discrete instance of OctoPOS is started.
These instances may communicate via a remote procedure call (RPC) mechanism to
interact with each other.

OctoPOS implements and extends the basic invasive operations *invaıde, infect* and
*retreat* at a certain abstraction level. A set of tile-local cores is represented as
an elementary claim called the *DispatchClaim* type. An empty DispatchClaim
can be constructed using *claim_construct* and one containing a single core is
automatically created on application startup. Invade is restricted to add tile-local
cores to a DispatchClaim and either succeeds for the requested amount or fails
without obtaining any cores. Notably, no constraints other than the exact number of
cores are taken into consideration. More complex constraints are handled by the agent
system and are indicated in Section 2.3.2. Conversely, the retreat operation removes
cores from a DispatchClaim. To start concurrent execution on a DispatchClaim,
OctoPOS defines the iLet type. It is a descriptor for a function and its arguments,
resembling a function call that may be executed concurrently. The infect operation
takes iLets and schedules them to run on the cores of the specified DispatchClaim.

All these operations are also available in a remote variant. DispatchClaims may
be constructed and used on remote tiles, where OctoPOS uses its RPC mechanism
to communicate these operations to the distant tile. For user communication to
da distant tile, an asynchronous DMA mechanism is provided through the function
*dispatch_claim_push_dma*. This function copies a user-specified buffer from the
source tile to a destination tile and schedules two iLets, respectively on source and
destination, when the DMA operation finished.

Further, OctoPOS introduces tile-local synchronisation primitives for iLets. These
include counting signals called *SimpleSignal* and binary signals called *BinarySignal*,
these are simplifications of traditional Semaphores as described in [23]. Both signals
define a *signal* and *wait* operation. The simplifications rest upon the decision to only
support one waiting iLet per signal. This allows for an efficient and wait-free imple-
mentation of these primitives. Signals support consumer-producer synchronisation
between iLets, effectively allowing iLets to wait for the completion of other iLets or
asynchronous operations.

OctoPOS, by default, does not virtualise CPU cores. Once an application obtains
a DispatchClaim it can be sure that these resources are exclusive for its iLets. It
is also its responsibility to retreat these resources as soon as it does not need them.
This behaviour is also consistent with the idea of resource awareness in Invasive
Computing, as presented earlier.
2.3.2 Invasive Concepts in the Agent System

The agent system encapsulates decision strategies for resource assignment on local and remote tiles, see [24] and [25]. Concretely, the agent system provides a more powerful invade operation that may take complex constraints into account. Such constraints are parameters of Downey’s model of application speedup [26] and the average required NoC throughput.

After the decision for the concrete computing resources is made, the agent system creates DispatchClaims and invades cores from OctoPOS for each tile. Additionally, the agent-system claim is defined to represent multiple DispatchClaims that are potentially distributed over multiple tiles. The agent system exists to relieve application programmers from implementing complex resource-managing strategies.

After the initial resource assignment agents are created for each application that bargain, in a distributed manner, about further adjustments to the acquired resources. This distributed system may then decide to shrink or grow the amount of resources that are allocated to an application. The size and amount of DispatchClaims may then be adjusted by the agent system. In this regard, applications must be malleable to deal with the new resource situation. If an application is not malleable, the initial resource assignment may not be changed until the application finishes or explicitly de- and reallocates resources.

The role of the agent system for this thesis will be pointed out in Section 3.3.

2.4 Software- and Hardware-Architecture Overview

An overview of the available software platforms on top of iRTSS and supported hardware architectures is depicted in Figure 2.2. This overview shows the state of the current development and includes results of this thesis. The hardware platforms include 32 and 64 bit architectures as well as big- and little-endian cores. The SPARC Leon architecture is the target for many of the invasive hardware projects. The VHDL design is open source and therefore, it is easy to include custom hardware for research purposes. Support for x86 64 bit was added as part of this thesis. This architecture is dominant in many fields of computing, from high performance to desktop systems. For this reason it was important to evaluate iRTSS and the results of this thesis on such commodity of-the-shelf (COTS) hardware. The x86 guest architecture is the interface of the x86 32 bit hardware and Linux system call interface. In this sense, iRTSS runs as an unprivileged application on top of Linux. The use for this architecture lies in the ability to quickly evaluate architecture-independent changes to either iRTSS or applications, as a “reboot” is just a restart of the Linux application. To support diverse architectures, OctoPOS defines a hardware abstraction layer (HAL) that must be implemented for the respective hardware. On top of this HAL, most of OctoPOS, the agent system and user-space libraries are platform independent. Implementations of the HAL for all of the mentioned architectures are detailed in Chapter 4.1.
From an application’s point of view, iRTSS provides interfaces at different abstraction levels. The system-call interface may be used directly or indirectly through many user-space libraries, foremost the C library and iRTSS user-space libraries, that provide the invasive operations mentioned earlier.

The main programming languages for user-space applications are C/C++ and X10. X10 is a type-safe partitioned global address-space language, originally invented by IBM, see [27]. Project C3 developed a compiler framework and runtime system to compile X10 to all invasive hardware platforms and to the iRTSS interface. The X10 runtime system uses the low-level DMA, claim, iLet and signal primitives to provide a type-safe and managed environment for application development.

Support for Fortran and MPI was added for this thesis to evaluate the NASA Advanced Supercomputing Division benchmarks [28]. These are comparable benchmarks, that utilise various communication patterns through MPI messages. Most of these benchmarks are implemented in Fortran and a subset of the Fortran runtime library for the GNU Fortran compiler was ported to iRTSS.
Adaptive Address-Space Design

This Chapter presents the design of all components of the adaptive address-space management system for OctoPOS. In Section 3.1 the scaffolding of the application model is introduced, supplemented with the design of the flat address-space layout. Three iRTSS libraries define the main interface for applications to the memory subsystem and OctoPOS. These libraries are themselves split into multiple areas of responsibility, that are: manual memory allocation, garbage collection, page-based allocation from OctoPOS, application protection and inter-tile communication. Subsequent Sections in this Chapter define these interfaces in more detail.

Memory allocation, in Section 3.2 is structured as a hierarchical model of coarse- and fine-grained allocators, from the C library down to physical memory. Requirements for the allocators and the garbage collector are made explicit. Adaptive memory protection acts on the coarser grained levels of this hierarchy. The design of adaptive memory protection is elucidated in Section 3.3 with the strategies of dynamic application protection and the protected and unprotected views on memory. Restricted memory access also interferes with communication between tiles. Inter-tile communication must also be controlled for isolated applications and Section 3.4 gives an overview for a design that covers shared memory access, as well as asynchronous message passing.

The fundamental principles of adaptive memory protection do not depend on a particular type of memory-protection mechanism. The concepts presented here may be realisable, for example, with MMU, MPU and segmentation-based protection. However, MMU-based protection is very popular, especially in systems with large main memory. It provides flexible support for many address-space models and is able to manage a lot of distinct regions with different access rights. Finally, the address-space layout, as defined in Section 3.1.1 may not be feasible with other techniques. For these reasons, the design assumes paged memory and MMU-based protection.
3.1 Application Model

The address-space management system establishes memory protection for applications and implements an application model. The definition of this model in the context of Invasive Computing shall be given in this Section. There are two points of view to describe an application, one is in terms of OctoPOS and the constructs described in Section 2.3 and one is in terms of memory isolation.

An application starts out on a single CPU core and a single DispatchClaim. It has a unique identifier and whenever it invades new resources, this identifier is associated with the respective resource. Therefore, a single application may consist of multiple iLets and multiple DispatchClaims on multiple tiles. To keep track of this information, OctoPOS was extended with an Application type and an instance of this type is created for each application. To keep this application model scalable, an instance of Application is also created per OctoPOS instance on each tile. OctoPOS is able to associate these instances, as they share the same application identifier. Multiple applications may be started, their concrete placing on the invasive hardware is implementation specific, but at runtime it is expected that they may share tiles and compete for resources.

From the point of view of memory protection, an application instance is a tile-local unit of isolation. Memory protection is either enabled for the whole unit, or not at all. Communication is supported between application instances that share the same identifier on different tiles. This is possible, either via shared memory access or OctoPOS’ asynchronous message-passing interface. It is expected, that applications bring their own copy of all their user-space dependencies. Sharing of libraries was not considered in this thesis, although there is no technical or conceptual reason why this should not work.

3.1.1 Address-Space Layout

Applications and OctoPOS reside in a flat logical address-space. This means, that OctoPOS establishes a surjective mapping $L_i \rightarrow P_k$, where a particular logical address $L_i$ always references the same physical address $P_k$ on a tile. In particular, a physical address may be referenced from multiple logical addresses and all physical addresses are reachable via logical addresses. The relationship of logical and physical addresses does not change after system initialisation. Nevertheless, access rights and ownership may be altered when switching between protected and unprotected modes or allocating memory for the application. This is in contrast to a non-flat or virtual address space, where the same logical address (in several address spaces) may be mapped multiple times to different or changing physical addresses. The reason for a flat address-space stems from OctoPOS’ history as a library operating system. No protection or abstraction based on physical memory was made before this thesis in OctoPOS. Consequently, the simplest expansion towards memory protection is a flat address space.
Furthermore, hardware devices, like DMA units or network devices, benefit from physically continuous memory ranges. This can be easily achieved in a flat address-space but requires attention and specialised allocation functionality otherwise. Section 6.3.1 discusses future work for a virtualised address space.

Figure 3.1 illustrates the conceptual design and logical layout of the flat address space from the point of view of a single tile. The logical addresses are the same on different tiles but reference different physical addresses for the local TLM and local devices. At large scale, TLM and SHM, introduced in Section 2.2, as well as devices are mapped side by side. A peculiarity is the mapping of TLM. There is a region \( \{g_0 - g_n\} \) where each TLM is mapped into the logical address space. This allows direct memory access into remote tiles and also allows the notion of local and global addresses. An address \( g_i \) into said region may be passed to other tiles and is still valid on that tile. Whereas a pointer into local TLM \( \{l_0 - l_k\} \) may have a different meaning on a remote tile, as it references different physical memory. OctoPOS provides means to compute a unique global address given the local tile identifier and a local address and, conversely, the local address from a global pointer. Pointers into SHM are always global and all tiles reference the same physical SHM through the same logical addresses.

Tile-local memory contains a copy of OctoPOS and the static memory sections of applications. Since the address-space is flat, applications start at disjoint logical addresses and do not overlap. The remaining free space in the TLM is used for dynamic memory allocations, this is detailed in Section 3.2. SHM is statically split into equally sized chunks between tiles and may also be used for dynamic allocations.

### 3.1.2 iRTSS and System-Call Interface

Applications interact with OctoPOS through iRTSS libraries. Therefore, the design, dependencies and interface of these libraries constitute an important share of the application model and are part of this section.

These libraries may either implement certain functionality in user space, perform system calls or may be part of the OctoPOS kernel. Additionally, OctoPOS may be set up as a library operating system and system calls may not perform a privilege switch. The concrete behaviour and border between system call or library function depends on the configuration of iRTSS. Nonetheless, the functional properties remain the same and the description here continues to be valid.

The application interface, concerning memory management, is divided into three parts, see Figure 3.2. Arrows in this figure denote a use relationship. The detailed interface functions are specified in terms of C functions in Listings 3.1, 3.2 and 3.3. The C library on top provides standard allocation functions for the C and C++ languages, that includes without limitation: malloc, free, realloc and memalign.
Chapter 3 Adaptive Address-Space Design

Figure 3.1: Conceptual view of the flat address space for a single tile.

These functions acquire and release memory of the same category and can therefore be intermixed as the language standards demand. The C library depends on the correct function of libOctoMem and uses it to provide its functionality.

Listing 3.1 Allocation interface of the C library, only one memory type is provided.

```c
1 void* malloc(size_t size);
2 void free(void *ptr);
3 void* calloc(size_t nmemb, size_t size);
4 void* realloc(void *ptr, size_t size);
5 void* memalign(size_t alignment, size_t size);
6 /* and further memalign-like functions */
```

The scope of libOctoMem is to manage large numbers of small memory chunks, usually smaller than the system page size. This functionality is provided as a manual memory-management interface and through a garbage collector. To obtain a specific type of memory libOctoMem provides allocation functions for each kind of memory in the invasive system. Memory of one type shall not be passed to allocation functions that handle a different kind.
3.1 Application Model

• Manual memory allocation: TLM, SHM, ...
• Page-based allocation
• Application protection
• Inter-tile memory access

Garbage Collector:
• Allocation
• Collection
• Pinning
• Special non-GC memory
• Statistics

Figure 3.2: Overview of the libraries that build the application interface concerning memory management.
The interface defines the same amount of allocation functions as the C library but adds a type identifier: `mem_allocate(type, size), mem_reallocate(type, pointer, size)` and so on. Freeing functions automatically determine the type of memory, based on address. These interfaces are OctoPOS specific and are typically not found on a non-invasive system. The garbage collector (GC) provides means to allocate memory, control the actual garbage collection, to pin/unpin objects, handle the interaction between GC-managed and unmanaged memory and provides memory-related statistics. Details of the requirements for the GC are lined out in Section 3.2.2.

**Listing 3.2 Interface provided by libOctoMem.**

```c
/* manual memory allocation */
void* mem_allocate(int type, intptr_t size);
void* mem_allocate_aligned(int type, intptr_t align, intptr_t size);
void* mem_reallocate(int type, void* p, intptr_t newsize);
void mem_free(void* p);

/* garbage-collector interface */
void* octo_gc_xmalloc(size_t size);
void* octo_gc_xrealloc(void* p, size_t newsize);
void octo_gc_collect(void);
void octo_gc_pin(void* ptr);
void octo_gc_unpin(void* ptr);

/* free non-GC memory containing pointers to GC memory */
void octo_gc_raw_free(void* ptr);
```

The library `libOctoPOS` provides resource management for coarser-grained objects and is typically part of the system-call and kernel layer of OctoPOS. Its interface consists of functions for page-based memory allocation, memory protection for applications and inter-tile memory-access control. Page-based allocation is again provided for all memory types. The `mem_map` functions acquire physically-backed memory and in contrast to POSIX `mmap` the concrete location in the address space may not be chosen. To support efficient implementations of `realloc`-like functionality, `mem_map_grow` may be used to grow a memory region if there is enough room in the address space behind the specified address. Shrinking may be implemented by unmapping unneeded pages. In the protected case, applications are not allowed to unmap memory that they do not own, respectively, that they did not allocate themselves. The protection API is meant to be used by the privileged agent system on behalf of the current application to enable or disable memory protection using `octo_app_protect` and `octo_app_unprotect`. Inter-tile memory access may be granted by `octo_reserve_access` to the specified remote tile and address range. To successfully grant access, the remote region must be owned by the same application on the remote tile, recognisable by the application identifier. To release access the application must call `octo_release_access` for the same region.
3.2 Memory Allocation

Calls to \texttt{mem\_unmap} on the remote region by the remote application will block the caller and only return once \texttt{octo\_release\_access} was called.

\begin{verbatim}
Listing 3.3 Memory-specific interface provided by libOctoPOS.

1 /* page-based allocation */
2 void* mem_map(int type, uintptr_t size);
3 void* mem_map_aligned(int type, uintptr_t alignment, uintptr_t size);
4 int mem_map_grow(void* ptr, uintptr_t size, uintptr_t newsize);
5 int mem_unmap(void* ptr, uintptr_t size);

7 /* application protection */
8 void octo_app_protect(void);
9 void octo_app_unprotect(void);

11 /* inter-tile memory access */
12 int octo_reserve_access(uintptr_t tileID, void* start, uintptr_t size);
13 int octo_release_access(uintptr_t tileID, void* start, uintptr_t size);
\end{verbatim}

Calls to \texttt{octo\_app\_protect} and \texttt{unprotect} should always be possible while other functionality is executed in parallel. This causes intricate synchronisation requirements between the functions in \texttt{libOctoPOS}. The concrete synchronisation strategies of the implementation are described in Section 4.5.

3.2 Memory Allocation

The design of the memory-allocation functionality is based on two allocator types. After introducing the two types, a global overview of the memory allocation design is given. Interfaces of these allocators are described in terms of C++ classes but the concepts are of course not limited to a specific language.

The \textit{BlockAllocator} is responsible for managing memory blocks of the same size and multiples of them. The purpose of the BlockAllocator is to manage raw memory in a compact way. Its interface allows implementations to omit headers for allocated blocks, because all relevant information about memory size and type must be managed outside of the allocator. Therefore, instances of the BlockAllocator class do not have to store additional information regarding returned memory. This in turn means, that any memory may be freed to such an allocator, especially memory that was not allocated from it in the first place. Listing 3.4 gives an overview of relevant parts of the BlockAllocator interface. A BlockAllocator is initialised with a certain block size that is computed as $2^{\text{blockBits}}$, that may not be changed after initialisation. Allocation via \texttt{alloc} is only possible in entire blocks and when returning memory to the allocator, via \texttt{free}, the amount of blocks must be specified. It is perfectly legal to free only a subset of previously allocated blocks. Memory given to the allocator, as well as all memory returned from the allocator must be aligned to be a multiple of the block size. Instances of the BlockAllocator class start empty and receive memory via \texttt{free}.
Instances of the BlockAllocator type may be stacked, to allow allocation of finer-grained memory blocks based on a coarser BlockAllocator, see Figure 3.3. A fundamental aspect of allocator stacking is that the finer grained allocator must, at some point in time, return memory to the coarse grained allocator. This is of course only possible with memory of the right alignment and size. For this purpose, the BlockAllocator interface defines \texttt{allocLargest}. This function allocates the largest continuous chunk of memory with the given minimum alignment \texttt{minAlign} and minimum number of blocks \texttt{minBlocks}. It returns a pointer to this chunk and writes the exact number of blocks into \texttt{minBlocks}.

![Figure 3.3: Stacking a fine-grained BlockAllocator on a coarse-grained BlockAllocator.](image)

The second allocator type is the \textit{FlexibleAllocator}, see Listing 3.5. This interface resembles the traditional C library interface. Memory as low as one byte may be allocated, freeing memory does not require the user to specify the amount of memory and only pointers that were previously returned by \texttt{alloc} may be passed to \texttt{free} of the same FlexibleAllocator instance. The purpose of the FlexibleAllocator is to be stacked on top of a BlockAllocator, that serves as the source of memory for the FlexibleAllocator. FlexibleAllocator instances need to store additional information into the managed memory, for example the real size of allocated chunks, which the user of the allocator must not know. This convenience comes with the drawback of internal fragmentation.
The FlexibleAllocator provides additional functionality that is explained in Section 4.4 but which is not relevant to the application interface or the understanding of the design of memory management in iRTSS.

**Listing 3.5** FlexibleAllocator interface.

```c
1 class FlexibleAllocator {
2     FlexibleAllocator(BlockAllocator *ba);
3     void* alloc(uintptr_t size);
4     void* allocAligned(uintptr_t alignment, uintptr_t size);
5     void* realloc(void *ptr, uintptr_t size);
6     void free(void *ptr);
7 }
```

As already mentioned, these interfaces are not bound to C++. Actually, the allocation interface of libOctoMem shall be defined as an instance of FlexibleAllocator. Respectively, the page-allocation interface of libOctoPOS shall be defined as an instance of a BlockAllocator.

With this in mind, the proposed global design of the iRTSS memory management is depicted in Figure 3.4, where arrows denote a use relationship. For example, the C library function `malloc` uses `mem_allocate` with TLM memory, which may use a fine-grained allocator to serve the request. Depending on the strategy of the fine-grained BlockAllocator, it may request coarse chunks of memory from `mem_map`, which itself allocates raw memory pages from a coarse BlockAllocator in OctoPOS, that also serves allocation requests from page-table allocation and other OctoPOS-internal memory allocation. For this internal allocation, a FlexibleAllocator is provided through a `kmalloc, kfree, ...` interface internally in OctoPOS.

### 3.2.1 Block-Allocation Requirements

As described in Section 3.1.1, the address space is flat. This implies certain non-functional requirements for BlockAllocator implementations. External fragmentation should be as low as possible, especially for the lowest-level BlockAllocator, that manages block sizes in the scale of memory pages. The reason for this comes from the fact that pages may not be re-mapped by OctoPOS to form continuous logical ranges, as this would violate the properties of a flat address space.

In other words, OctoPOS cannot correct external fragmentation in page granularity by using the MMU and simultaneously preserve the conditions of a flat address space as defined in Section 3.1.1. Fragmentation on this level may cause “lost” memory that may never be used by applications, when `mem_map` calls always request larger amounts of memory.

Resource awareness, that stems form Invasive Computing, is another requirement on the whole allocator hierarchy.
Chapter 3 Adaptive Address-Space Design

Figure 3.4: Overview of the global allocator hierarchy in iRTSS.
3.2 Memory Allocation

No single instance of BlockAllocator or FlexibleAllocator shall buffer large amounts of memory. Memory shall always be returned to the coarser allocator in reasonable sizes. Ultimately, making the physical pages available to other applications or OctoPOS. This shall prevent wasting of memory resources as pointed out in Section 1.1.3. The concrete “reasonable” size depends on the implementation and architecture, as this eventually balances resource awareness versus performance.

3.2.2 Garbage-Collector Requirements

At least the invasive X10 runtime-system depends on the correct functioning of a garbage collector (GC). This runtime system assumes certain properties of the GC and these properties are made explicit in this section.

The GC provides means to allocate objects in memory and it must be able to collect these objects automatically when they are not referenced any more. This shall be possible without the knowledge of the application object layout. Therefore, the GC may assume, that pointers are not mangled or otherwise altered and always point to the beginning of the memory region that defines the object.

The GC shall serve one type of memory, for example SHM. It should be possible to store references to this type of memory in other memory types, that are not allocated through the GC, like TLM. The GC must, nevertheless, know this reference and not deallocate the memory, even if there are no references in SHM any more.

It must be possible to dynamically pin objects returned by the GC. This means, that even if there is no reference to that object in the future, the GC shall treat it as if it were referenced. Similarly, it shall be possible to dynamically unpin a pinned object. This requirement originates from other hardware-related research projects and details are mentioned in Section 4.4.3.

The GC shall be able to collect a lot of small objects in reasonable time. This non-functional property is hard to define precisely, as it also depends on the application behaviour. In general, the runtime of the application should be much larger than the collection time of the GC, for example 100:1.

Another non-functional property is, again, that of resource awareness. The GC shall, like the BlockAllocator implementations, return memory to OctoPOS regularly in adequate sizes.
3.3 Adaptive Memory Protection

Adaptive memory protection operates on whole memory pages and therefore concerns the design of libOctoPOS. At runtime, OctoPOS distinguishes three modes of operation.

**Statically unprotected system:** OctoPOS and the interface in libOctoPOS form an unprotected library operating system. System calls do not perform a privilege switch and all applications run in privileged mode. It is not possible for OctoPOS to activate correct memory protection at a later time, because the used memory regions of applications may be unknown. This mode of operation may be the most performant and predictable with respect to memory management, but also the most unsecure.

**Dynamically unprotected application:** Similarly to the previous mode, certain applications may run in privileged mode, but OctoPOS retains knowledge about the allocated and freed pages through the page-based interface. Access to other memory regions, that were not previously allocated may be possible, but this shall be defined as faulty behaviour. As such erroneous accesses may not be detected, this mode is also unsecure. However, with the information of the exact memory regions that belong to the application, OctoPOS is able to activate memory protection for a previously unprotected application, see below. OctoPOS must do additional bookkeeping for the allocated memory regions, plus take measures for the possibility of dynamically switching into protected mode. Hence, this mode may be considered less performant than a statically unprotected system but with similar predictability properties.

**Dynamically protected application:** When memory protection is active for an application, it may only access previously allocated regions of pages. Accesses to other regions are blocked and generate a trap. This has the effect of isolating the application from unallocated memory, OctoPOS itself and other applications’ memory. Therefore, this mode may be considered the most secure with regards to memory-access isolation. Albeit, OctoPOS must additionally manage MMU data structures and care for TLB consistency. As a consequence, this mode may be the least performant in terms of memory allocation from OctoPOS. Furthermore, this mode may be considered as the least suitable for real-time systems. The multicore TLB consistency causes jitter that may be detrimental for real-time use.

The goal of the design is to leave unprotected modes as unaffected as possible in order to enable maximally performant implementations and to reduce the background noise to a minimum. Whereas switching between modes and the protected mode itself may incur more overhead and noise caused by the operating system. A mixed mode of operation, where some applications are dynamically isolated and some are not is possible by design. Section 3.5 elaborates on the strategies and requirements for lifting isolation and applying it again.
3.3 Adaptive Memory Protection

To switch between protected and unprotected modes, OctoPOS maintains at least two mappings of the address space. One protected view on the flat address space that is application specific and a single tile-local mapping for unprotected mode.

Figure 3.5 shows an example using two applications. The unprotected mapping allows access to all memory by privileged (kernel-mode) and unprivileged (application) code, this is denoted in the top row by “K” and “U” letters. The protected mapping for the application consists of a kernel-mode region where OctoPOS resides and a region for the static application data that is user accessible, denoted by a “U” letter. In this mapping the static data of other applications is not accessible, denoted by a “0” letter. Beyond that, the region of dynamic allocation is split into a user-accessible and inaccessible parts, depending on the allocations and deallocations that the application performs. This dynamic region conceptually includes TLM and SHM. All unallocated parts of the dynamic region remain kernel accessible by design, even though they may belong to a different application. This avoids the cost of keeping the mapping in dynamic regions synchronous in all protected-mode page tables.

The high level overview of the page-based allocation and deallocation in the face of the specified configurability is depicted in Figure 3.6. The page-based allocation interface (1) performs requests to the correct BlockAllocator (2), depending on the requested memory type and operation. When running in a statically unprotected system, the resulting memory and success status is immediately returned back to the application (6). If dynamic memory protection is possible in the system, the application’s memory regions are updated (3) to reflect the exact memory pieces that belong to the application.
Figure 3.6: Activity diagram of the page-base allocation design in the face of static and dynamic configurability.
This may involve removing parts of a memory region or splitting a region. In the case that the application is isolated, the application’s page table must be adjusted to also reflect accessible and inaccessible regions (4). Afterwards, the local TLB entries must be adjusted and if the application spawns multiple CPU cores, the remote TLB entries must also be kept synchronous (5). As this overview reveals, the unprotected cases do not modify page tables. In particular, the tile-local unprotected page table is never modified, regardless if it is active or not.

3.3.1 Switching Protection Modes

With the page-based allocation, as designed for 
\texttt{libOctoPOS}, switching protection modes essentially reduces to switching the active mapping from the protected application mapping versus the tile-local unprotected mapping.

To switch from dynamically protected to unprotected mode, OctoPOS performs a multicast operation to all CPU cores that belong to the application. This multicast instructs all cores of the application to switch to the tile-local unprotected mapping. Afterwards, the successful page-table change is synchronously signalled back to the issuing core. It is only guaranteed via a synchronous feedback, that the application is completely in unprotected mode on all its cores. Once the operation completes, OctoPOS marks the application as unisolated. In the course of the application’s phase in unprotected mode, the application-specific protected-mode page table is not modified, as outlined in Figure 3.6.

To turn protection on again, the protected page-table must be updated to contain all memory regions that belong to the application. In particular, memory that was unmapped while in unprotected mode must be removed from the application page table and newly allocated memory must be added. Similar to the above, a synchronous multicast to all of the application’s CPU cores signals the switch to the protected mapping.

The implementation must cope with concurrent changes in the set of the application’s CPU cores and concurrent \texttt{map} and \texttt{unmap} operations, changing the set of memory pages while protection is activated or deactivated. Details of the synchronisation strategies are clarified in Chapter 4.

3.4 Adaptive Inter-Tile Communication

Inter-tile communication may happen between the instances of the same application running on different tiles. OctoPOS offers means to send iLets to remote claims and execute them in application context. iLets may carry data in form of function arguments. Although this may be enough to establish communication between tiles, it is less efficient when large amounts of application data must be copied. For this reason, OctoPOS provides an asynchronous message-passing mechanism through the \texttt{dispatch_claim_push_dma} interface that, depending on the architecture and configuration, may copy large amounts of data asynchronously.
Chapter 3 Adaptive Address-Space Design

Another way to access large amounts of memory on different tiles is to use shared memory, either in form of shared SHM or distributed TLM.

Without memory protection, these inter-tile accesses may happen wildly and erroneously and may corrupt foreign application data by accident or on purpose. Therefore, it is in the scope of the adaptive memory-protection system to restrict inter-tile accesses to the allowed regions of an application. The complexity of this problem arises from dynamic changes of the sender and receiver address space. These address spaces are separated and changes may occur in parallel. The receiver, for example, may unmap pages while the sender just started a DMA operation to this region. Checking accesses in OctoPOS before the operation lead to time of check, time of use (TOCTOU) problems. To resolve these TOCTOU problems, OctoPOS must explicitly grant accesses (on the sender side) to remote regions and block any attempts to unmap these regions (on the receiver side) until the remote access is not needed any more.

The following access-control mechanism is designed for both, asynchronous message-passing and shared memory access. With `octo_reserve_access` the sender side may reserve a remote memory region. OctoPOS is responsible for checking and granting the permission. Once the reservation is not needed any more, the application may enable access with `octo_release_access` again. Immediately after granting access, the remote tile will be unable to unmap the reserved region until the access is released. Each application representation in OctoPOS possesses a cache and a registry that store permissions per tile. Permissions for remote access must be acknowledged by the remote tile, which requires inter-tile operations using OctoPOS‘ RPC mechanism. The cache improves on this by storing already granted permissions. On the next access reservation, the local OctoPOS instance may find the entry in the cache and no RPC is needed. As address-spaces change, entries in the cache must be invalidated. For this reason, each OctoPOS instance stores information about all granted accesses in a data structure called “registry.” With this registry, OctoPOS is able to invalidate granted accesses on other tiles using the same RPC mechanism.

Figure 3.7 guides through the participating objects and activities. If the application wants to reserve access, OctoPOS searches the cache (1). If no cached permission was found, the OctoPOS instance on the remote tile is asked (2) to check the permissions (3). If the requested memory range belongs to the same application, an entry is created in the registry that stores the fact that the requesting tile has a permission for that range. Success is signalled back (5) to the requesting tile, that in turn creates a cache entry, containing the information about the valid range (6). Additionally the cache entry is marked as “in use.” As soon as the access is released, the cache entry is marked as “not in use” and stays in the cache. If, in between, the receiver side unmaps memory (7), OctoPOS checks the registry for overlapping regions. If a region is found, all tiles that store a cache entry must be invalidated (8), eventually removing the respective cache entry.
Figure 3.7: Reserving memory access for inter-tile communication using a per-application cache and registry.
When a cache entry is found to be “in use” the implementation may choose to abort the unmap operation with failure or wait for a release to happen. In any case the remote unmap cannot successfully return while the region is reserved. Checking the registry is performed in step (3) of Figure 3.6 where the application’s regions must be updated. The registry extends this set by those regions that were granted special access for other tiles.

This inter-tile permission mechanism is adaptive in the sense that less checking and inter tile operations are performed if the application is in unprotected mode. When switching into unprotected mode, the local registry is emptied and all remote permissions are invalidated. Reserving and releasing accesses are always successful and only operate on the local cache. The `mem_unmap` operation still checks the local registry, which might be empty. When memory protection is turned on, all “used” permissions in the cache must be validated on the respective remote tiles, all other cached permissions are deleted.

Using this strategy, there are three combinations of protected or unprotected application instances:

1. Local and remote application instances are all unprotected: Every instance will be trusted and permissions are only updated on the local cache. Unmap always finds an empty registry and the range check will be very cheap. No inter-tile operations happen to grant application communication.

2. All application instances are in protected mode: The system closely follows the procedure explained before. Permission must be validated on the remote tile, registered in the remote registry and locally cached. The unmap operation queries the local registry and may perform remote invalidations.

3. Some application instances are protected, some are not: The unprotected application instances are trusted and therefore still operate locally, while protected applications may place registry entries in the registry of an unprotected instance. Unmap may therefore find said registry entry and will invalidate it in the protected remote application instance.

The last mode is peculiar, because as soon as one application instance is unprotected, all application instances may instruct it to do arbitrary memory accesses and must therefore also be considered unprotected. From the point of view of OctoPOS the last mode is not useful as a permanent state, but may make sense when memory protection is lifted sequentially tile after tile, or memory protection is only temporarily activated. Nonetheless, it is automatically supported by the afore-mentioned strategy.
3.5 Strategies for Lifting Address-Space Isolation

Lifting address-space isolation for an application is dangerous. However, some applications may be trustworthy because they are written in a type-safe programming language and the compiler and runtime system is also considered trustworthy. This is the case with the invasive X10 runtime system and other language implementations that were ported in the course of this work. For these cases, the overhead of memory protection is in vain. Additionally, security may be traded for predictability and performance, as it is the case for many embedded systems or HPC systems. The evaluations in Chapter 5 will show these performance and predictability advantages.

The adaptive properties of the presented design allow dynamic protection strategies. For example, a type-safe application may run in unprotected mode and when calling a native library function the respective runtime system may enable memory protection for that single function call. This combines the advantages of high performance when executing trusted code and security when native code is involved.

Furthermore, the presented design allows a flexible combination with hardware-based protection mechanisms. This is very important in the context of different architectures and research projects. If, for example, applications cannot access memory outside of their tile, then memory protection may be lifted as long as there is only one application on that tile or on the system. As soon as more applications join, memory protection can, selectively, be activated.

Finally, the ability is made available to the agent system in iRTSS to dynamically bargain the state of application protection and to fathom the risks and benefits, using all available constraints and information. OctoPOS itself only provides the functionality and interfaces for adaptive memory protection, but it will not initiate a protection change by itself.
Adaptive Address-Space Implementation

This chapter starts with a summary of the concrete implementation details of the invasive hardware model on contemporary architectures. In the course of this thesis, OctoPOS and its specific memory layout was ported to x86-64 hardware and x86 32 bit software interfaces. The specifics of these ports are outlined in Section 4.1.

Section 4.3 presents the isolatable application model and its development from OctoPOS’ library system point of view. Sections 4.4 and 4.5 document tile-local algorithms and implementation details. These details include concrete data structures and algorithms for memory allocation, adaptive isolation and synchronisation.

Finally, inter-tile communication aspects are illuminated with emphasis on the implementation of a standards compliant MPI implementation.

4.1 Invasive Hardware-Model Implementations

For this thesis, OctoPOS was ported to x86 64 bit systems and support for x86 32 bit on top of Linux was enhanced. Adaptive memory protection was implemented for the Sparc-Leon and x86-64 architectures. The implementations of the invasive hardware model as introduced in Section 2.2 as well as the implementations of the address-space layout described in Section 3.1.1 are presented in this Section.

OctoPOS has a specific hardware abstraction layer (HAL) for each of its architectures. The responsibility of a HAL is to implement a mapping from the concrete hardware to a common model of invasive hardware. Every other part of OctoPOS is implemented against this common model, that corresponds to the description in Section 2.2. Challenges for the concrete implementation of adaptive address-space management on each architecture are outlined here. The concrete implementation of the invasive hardware model has significant influence on multiple non-functional properties of applications and OctoPOS. Parts of this are runtime performance of the system and ease of programmability for applications. Ultimately, adaptive memory protection depends on the HAL and benefits from a common abstraction.
Chapter 4 Adaptive Address-Space Implementation

To implement OctoPOS’ full API the HAL must provide, among others, the following abstractions:

- One or more tiles containing an copy of OctoPOS that is executed on all cores at boot time. Tiles have a logical identifier from 0 to $N - 1$
- Double mapping (or aliasing) of TLM to give the notion of local and global addresses.
- Inter processor interrupts from and to logical CPU cores. These are logically numbered from 0 to $M - 1$ on each tile. The HAL provides means to send interrupts to the logical pair <tile ID, core ID>.
- The implementation of static core-local memory.
- Mapping interrupts to logical handlers in OctoPOS.
- Control-flow contexts with the ability to save and restore context state.
- General hardware information in terms of available TLM, SHM, cores, tiles and availability of special hardware components.
- Finally, functions to access the MMU, potentially hierarchical page tables and their abstracted descriptors.

The following sections shed light on some details of the implementation of these abstractions on the mentioned architectures. Particularly, Section 4.1.1 highlights the last point of the HAL which was supplemented by this thesis.

4.1.1 Page-Table Implementation and Abstraction

The platform independent implementation for page-table manipulation assumes that the MMU may require multi-level page tables. The first level is denoted as level 0 down to level N and each level consists of an array of descriptors. The concrete layout of the descriptors is architecture dependent and the number of descriptors per level may be differing, but it is assumed to be a power of two. It is also assumed that consecutive bits in a logical address build an index into a level. It turns out, that these assumptions are sufficient for many MMU implementations, especially x86-64 and Leon that are the OctoPOS supported architectures with a hardware MMU.

The architecture independent implementation consists of the classes GenericPagemap and MapFlags. Architecture dependent parts are found in the PageHelper class and further architecture dependent constants. MapFlags encapsulates access rights and other descriptor flags using platform independent access methods. The architecture-specific implementation must be able to decode and encode descriptors from and to MapFlags objects.
4.1 Invasive Hardware-Model Implementations

Listing 4.1 GenericPagemap interface.

```cpp
class GenericPagemap {
    // invalidations
    static void invalidate(uintptr_t logicalAddress);
    static void invalidateRange(uintptr_t start, uintptr_t pages);
    static void invalidateAll();

    // change descriptor entries
    void map(uintptr_t logicalAddr, uintptr_t physicalAddr, MapFlags flags, uintptr_t pages);
    void mapFlags(uintptr_t logicalAddr, MapFlags flags, uintptr_t pages);

    // retrieve an entry
    void getEntry(uintptr_t logicalAddr, uintptr_t *physicalAddr, MapFlags *flags);

    // get and set the active CPU-local page table
    static GenericPagemap* getCurrent();
    void apply();

    // creation, destruction and copy
    static GenericPagemap* makeNew();
    static void free(GenericPagemap* pagemap);
    GenericPagemap* deepCopy();

    // iterate over page-table entries, iterator defines page-table levels,
    // and callbacks for "holes" and "entries"
    template<typename Iterator>
    void pageWalk(uintptr_t startAddr, uintptr_t endAddr, Iterator *iter);
};
```

The interface of GenericPagemap, see Listing 4.1, provides local TLB invalidation functions for single addresses, address ranges and the complete TLB. Page-table manipulation functions consist of map which can universally be used to map any logical address to any physical address. This is only relevant for system initialisation, as the implementation does not change the mapping at runtime. To modify ownership and other flags the mapFlags function applies these changes to a mapping while preserving the logical to physical correlation. Further functions serve for getting and setting the active CPU-local page table and creating, destroying and copying hierarchies.

The implementation of GenericPagemap is formed around a flexible iterator over hierarchical page-table data structures, this is also commonly referred to as page walk. It allows on-the-fly manipulation of the page-table hierarchy and is used internally for all page-table manipulation functions. The page-walk mechanism iterates through multilevel page-tables in a depth-first manner, see Figure 4.1. It is assumed to be more efficient than single accesses that traverse the whole hierarchy each time. By encapsulating this iteration mechanism, all page-table manipulation functions profit from locality when modifying the mapping or flags for a range of pages.
The architecture dependent part consists of constants that define the number of page-table levels, the number of bits for each level that are used as index into each level and the amount of bit-shift required to access the index bits in the descriptor. Additionally, the `PageHelper` class encapsulates the instructions for TLB invalidation, efficient descriptor handling and it provides two allocators for building the page-table hierarchy. The boot-time allocator is used early when initialising the system and the regular page allocator is used afterwards. The allocators allow different allocation strategies that may be architecture dependent. Details of the implementation of these allocators are explained in Section 4.4.

### 4.1.2 Invasive SPARC-Leon Architecture

SPARC-Leon is based on the SPARC-V8 32 bit RISC core with additions. It was chosen as the main architecture for Invasive Computing and the hardware projects introduced in Section 2.1 implement their research projects into it. The Leon CPU core as well as other essential components are available as VHDL and Verilog source code and run on FPGA boards. The invasive hardware design presented in Section 2.2 was closely followed and therefore, the HAL is very close to the bare hardware implementation. Tiles form self-contained computing systems with CPU cores, memory, caches and local serial I/O. The debugger and analysing tool `grmon` plays an important role in the boot process, as it also functions as a bootloader and takes care of copying OctoPOS into the TLM of all tiles. The tiles are connected with the invasive network-on-chip (iNOC) that allows direct memory access to other tiles and provides other hardware accelerated operations. All SHM is located at one tile, which is termed IO-tile. SHM is logically split into equal and continuous parts for each tile. Cache coherency is not available over the iNOC.
Therefore, memory access between tiles and between the tiles’ parts of SHM, must be done with special caution and manual cache invalidations. Accesses to TLM may be performed in parallel on different tiles, whereas access to SHM may be limited by the iNOC throughput and the fact that it is connected to a single tile. Currently, the largest system consists of 16 tiles with 5 CPU cores per tile, 8 MiB of TLM (per tile) and 1 GiB of SHM. This results in 64 MiB SHM + 8 MiB TLM per tile, which is rather lean for many applications and requires careful memory management.

The memory allocators must handle memory in a compact way and avoid internal and external fragmentation as far as possible. The address-space layout defined in Section 3.1.1 is implemented completely in hardware without MMU or OctoPOS intervention. The HAL and further infrastructure to boot OctoPOS on Leon was implemented by Oechslein, Schedel, Erhardt and others, see [19]–[22].

The MMU was utilised for this thesis to implement adaptive memory protection on the invasive Leon architecture. As the memory controllers and iNOC already implement the double mapping of TLM, only an identity mapping is applied. This means, that logical addresses match the physical addresses except for physical addresses without attached hardware, these are mapped as invalid and generate a trap on access. SPARC-Leon defines a three-level page table with support for 32 bit logical and 36 bit physical addresses. The TLB is automatically filled on miss and provides hierarchical and range-based control over local invalidations. TLB units are not synchronised between cores in hardware.

4.1.3 x86-64 Support

Support for x86-64 for OctoPOS and adaptive memory protection on this architecture was implemented in the course of this thesis. This architecture is also called x64 in iRTSS terms. The idea for x64 is to implement tiles as NUMA domains. CPU cores that belong to a domain form the cores that are available on that tile and the domain-local memory is made available as TLM. Invasive applications automatically profit from NUMA-local memory accesses, as they are already aware of local and remote memory. This addresses the problem mentioned in Section 1.1.1 in a way, that simplifies memory management for OctoPOS and applications. Cache coherence is available between NUMA domains and the responsibility for cache maintenance is shifted from software to hardware.

SHM is not modelled for x64 architectures and the respective allocators are empty. However, TLM storage is far larger than on the prototype Leon systems. The largest available evaluation system consists of four NUMA domains with 128 GiB RAM each and 24 cores per domain. On these systems, the challenge for the memory-allocation system lies in efficient running time despite the large amount of memory. To enable 64 bit mode the MMU must be used on x86-64. The implementation supports four-level page tables for 48 bit logical and 48 bit physical addresses.

The x64 startup system initialises the specific HAL for this architecture before calling into the OctoPOS main initialisation function.
Its duties include:

- Detecting the amount and addresses of the available physical memory.
- Determine NUMA domains and the associated physical memory and CPU cores.
- Assigning logical OctoPOS tile and core identifiers to the physical CPU core identifiers.
- Finding enough continuous memory for the OctoPOS binary image and physically replicating OctoPOS into all NUMA domains.
- Creating page tables for each NUMA domain that include the local memory as TLM and remote TLM via global addresses.
- Booting all CPU cores into 64 bit mode and applying the correct page-table hierarchy.
- Filling the tile-local allocators with the available memory.

![Page Table: Level 0](image)

Figure 4.2: Page-table level 0, implementing the TLM double mapping, tiles are identifiable by level 0 index bits of the logical address.

The concrete mapping of TLM is depicted in Figure 4.2. Each page-table hierarchy is 4 KiB in size and contains 512 descriptors, addressable with 9 index bits of the respective part of a pointer to logical memory. The local TLM is mapped into the first logical 512 GiB region, that is defined by the first sub-hierarchy entry in the lowest-level page table. Of course not all of the 512 GiB are available, but a constant offset to other tiles’ TLM is practical for address calculations. Accordingly, the global TLM mapping of tile $i$ begins at $512 \times (i + 1)$ GiB. This scheme is limited to 510 tiles, as one entry is responsible for the local TLM and the last entry is reserved for global device memory.
4.1 Invasive Hardware-Model Implementations

The x64 TLB is automatically filled on miss and provides invalidations for single entries or the whole TLB, range-based invalidations are not provided in hardware. TLB coherency between cores must be managed by the operating system.

4.1.4 Linux Guest Support

The x86-guest HAL provides an invasive hardware view on top of the 32 bit Linux system-call API in user land. Adaptive memory protection is not supported on x86 guest but the HAL was enhanced to support the double mapping of TLM. The goal for this architecture implementation is to provide means for faster prototyping of hardware-independent operating system and application software, as “rebooting” the system becomes a quick restart of an application. Debugging profits from all available tools on Linux and additionally, options such as SHM and TLM sizes or number of tiles and cores may be chosen by the user when starting the system. For this reason, the HAL should provide abstractions that implement the model of invasive computing as close as possible. Before the work of this thesis, local TLM pointers were non-existent and the conversion from global to local pointers returned the global pointers again. Combined with cache coherency on x86, this raised bugs when the local-to-global conversion operations were omitted by mistake. This gave rise to programs working well on x86 guest but only occasionally on Leon, which counteracts the promised goal of rapid prototyping.

Tiles are implemented as Linux processes and CPU cores are threads within those processes. Copying of OctoPOS and applications to the other “tiles” happens automatically, as the `fork` system call virtually copies all private memory of a process. This includes the static data sections and an additional private mapping that represents additional free TLM memory. Afterwards shared memory is mapped to create SHM and a global view on TLM. This is achieved by establishing a shared and named (file based) mapping between all processes (tiles) and copying the local TLM into the global mapping, see Figure 4.3. Thereafter, the global mapping is mapped again onto the original local TLM region of each process which finalises the double mapping of TLM.
Figure 4.3: TLM double mapping on x86 guest.
4.2 Red-Black Trees

Important parts of the implementation use red-black trees as a core data structure, these are self-balancing binary search-trees. A short investigation into red-black trees follows, to understand the performance and runtime implications of the implementation. This data structure was invented by Rudolf Bayer [29] and a tangible implementation and analysis may be found in the book “Introduction to Algorithms” [30].

Red-black trees store key-value pairs in sorted order and build a binary tree of nodes, connected with left, right and parent pointers. Additionally, a “colour” of red or black is associated with each node, which is used to rebalance the tree. This colour is encoded as least significant bit of the parent pointer, therefore it takes no additional storage space. Red-black trees support search, insert and delete operations in $O(\log N)$ worst-case bounded time, where $N$ is the number of nodes stored in the tree. This includes ceiling and floor search operations, that find a node which is “greater or equal” or “smaller or equal” to a given key. Similarly, the previous or next node for a given node can be found in $O(\log N)$ bounded time. Generally, insert and remove are considered more costly than other operations, because the tree has to be rebalanced.

The implementation developed for this work is based on the Linux kernel red-black tree but ported to an intrusive C++ template class. In an intrusive data structure the nodes in a red-black tree are not allocated internally but must be provided as storage within objects of the user of the tree. Thereby, the tree functions do not depend on dynamic memory allocation. As a consequence, this implementation of red-black trees may also be used in the memory-allocation algorithms itself.

4.3 Application Model Implementation

OctoPOS did not have an efficiently isolatable application model, as it was designed as a library operating system without memory protection. Applications were linked to the OctoPOS library forming a single binary with intermixed user and kernel code and data. Library state was shared between applications, this especially affects memory allocation and prohibits application isolation. To perform memory protection, application and operating-system data and code must be separated and aggregated into continuous sections. Furthermore, if applications from different origins are incorporated, symbol names may clash and creating a single binary is not possible by simple linking. A further challenge stems from the fact that user-space libraries may be stateful. To separate this state between multiple applications, the library state must also be multiplied. Loading application binaries in the Executable and Linking Format [31] (ELF) was considered but rejected because it requires a large implementation effort, especially when considering multiple target architectures. Instead, the OctoPOS and application build system was extended with the MultiApp program and further build steps, see Figure 4.4.
Figure 4.4: Modified OctoPOS build system, integrating multiple isolatable applications.

For each application a metadata file contains information about the object files that belong to the application. Further information may be provided such as the application name and other options. Additionally, all required user-space libraries must be listed. Building the application’s object files and libraries is performed by a previous build step (not shown in the figure). The MultiApp program then takes multiple metadata files, their associated application object files and libraries, a linker-script template from an OctoPOS release and generates three groups of output files.

The first group are all libraries and object files with renamed symbols. Each defined symbol is prefixed with an application-specific name. All uses of these symbols are also expanded with the specific prefix. This avoids name collisions, allows multiple applications to use the same library and even allows the inclusion of the same application multiple times into the final binary.

The linker-script template contains sections for OctoPOS and is filled with all applications’ sections separated and aligned to page boundaries. Each section is tagged with an start and end marker, that can be used as symbols from within OctoPOS and especially AppInfo.cc.

AppInfo.cc comprises of a generated C++ class containing all information that OctoPOS needs to enable memory protection for the applications. This includes for each application: the address ranges of all sections, each entry point and other information provided by the metadata file.
4.3.1 Application Objects

At system initialisation all information in AppInfo is transferred into dynamic data structures. For each application a tile-local object of type Application is created. It contains a set of the application’s memory ranges, both, static and dynamic. Each range stores type and access information, as well as start and end addresses. A red-black tree is used as an implementation for the set of memory ranges, sorted by the starting address of the respective range. All of the following operations on the set complete in $O(\log N)$ bounded time, where $N$ is the number of discrete ranges in the set:

- Insert a range and merge it with neighbouring ranges if type and access types match.
- Remove a (sub-)range, potentially splitting an existing range into a maximum of two new ranges.
- Check whether a user-provided pointer or memory range in fact belongs to that application.

Applications also contain a map of tile-local cores which belong to the application and thus, are subject to the same isolation restrictions. This set contains the union of all cores in all local DispatchClaim objects that belong to the application.

4.4 Memory Allocation

In Section 4.4.1 the general implementation of the block allocator interface is presented. The unsuitability of the buddy allocation algorithm for the presented system is also expressed in this Section. Afterwards, the implementation of the allocator hierarchy is lined out in Section 4.4.2. The implementation of the garbage collector for invasive X10 in Section 4.4.3 finalises the memory allocation topics in this thesis.

4.4.1 Block Allocator

A popular choice for the implementation of the BlockAllocator interface would be a buddy memory allocator [32]. This allocator is in use in many systems, for example in the Linux kernel for physical page allocation. It guarantees $O(\log M)$ bounded allocation and deallocation time where $M$ is the amount of managed memory. It accomplishes this runtime by splitting and merging aligned memory “buddies” whose addresses only differ in a single bit. There are very diverse implementations of buddy allocators but the common algorithm remains the same. Despite the advantages, the implementation of the adaptive address-space management system uses a novel allocation algorithm. To understand the reason for a new allocation algorithm, it is first shown how buddy allocators fail to accomplish the requirement of low external fragmentation defined in Section 3.2.1 and also suffer from high internal fragmentation. Figure 4.5 depicts an example of 64 KiB free memory aligned at 64 KiB. The lowest block size of the allocator is 4 KiB in this instance.
Chapter 4 Adaptive Address-Space Implementation

The first allocation requests 4 KiB and the large free chunk is repeatedly divided into equally-sized smaller buddies until 4 KiB is reached, which is then returned. Note that after this allocation it is not possible to allocate a chunk of a size larger than 32 KiB, although there is enough continuous free-space available. A more extreme example is shown in the line below, if the first and last block is not freed, then even an allocation of a block larger that 16 KiB is not possible any more, although there is a continuous chunk of size \((64 - 8) \text{ KiB} = 56 \text{ KiB}\) available. This is a form of external fragmentation and it can be detrimental in flat address-space systems. Furthermore, the external fragmentation problem also becomes apparent when the free memory is dynamically expanded. If the exemplary 64 KiB chunk is expanded by 16 KiB directly behind the free memory, then the allocator still fails to answer requests greater than 64 KiB. Note that these kinds of external fragmentation do not necessarily depend on the allocation behaviour of the user of the interface, but on the internals of the allocation algorithm. Memory expansion may not happen at the physical-page level, but on higher levels of the allocation hierarchy as illustrated in Section 3.2. High internal fragmentation happens with the last example of an allocation of 20 KiB memory. The buddy memory allocator must choose the next larger power of two size. In this particular case this is 32 KiB leading to 12 KiB internal fragmentation. In general, the worst case overhead of internal fragmentation may be nearly as high as 50% unused space.

![Figure 4.5: Internal and external fragmentation of the buddy memory allocator.](image)

A novel memory allocator was designed and implemented to meet the requirements defined in Section 3.2.1 it is called TreeBlockAllocator. This allocator implements the BlockAllocator interface presented in Section 3.2 as a C++ template class. The detailed implementation is listed in Appendix A.1.1.
The allocator must perform two main tasks with the set of free memory chunks:

1. When allocating memory, search for a chunk with the best fitting size for the allocation request.
2. When memory is freed, search for a predecessor and successor by address to merge the chunks.

These are classical search tasks and utilising appropriate data structures for searching becomes apparent. The implementation uses two red-black trees to manage the free chunks, one sorts by chunk size and the other by chunk start address. To reference these trees, they are termed `SizeTree` and `AddressTree`. The AddressTree effectively implements a set of non-overlapping intervals. Whereas the SizeTree forms a multimap, because the same chunk size may be present multiple times. Listing 4.2 presents the coarse algorithmic ideas for allocation and freeing as pseudo code.

Allocation proceeds by searching a node in the SizeTree which is “greater or equal” in size to the requested number of blocks. This is exactly what the `ceil` red-black tree operation provides. On success, the returned chunk is removed from both trees and a trailing chunk is split off. Splitting off a sub-chunk just involves writing a new header with the corrected block size. This trailing chunk is then inserted again into both trees.

Freeing a chunk is achieved by computing the start and the exclusive end address of the chunk to be freed. The successor chunk may be found by exact `search` for the end address in the AddressTree, since the AddressTree sorts by start address and the successor must start where the current chunk ends. The predecessor if found by using the `floor` function of the AddressTree with the start address of the chunk to be freed. If there is no direct predecessor, a wrong node is returned by `floor`, so it must be checked whether the end address of the predecessor matches the start address of the current chunk. Afterwards, successor and predecessor are removed from both trees. Merging the chunks just involves writing a new header into the lowest chunk, which is either the predecessor or the current chunk, with the accumulated block size. The merged chunk is then inserted into both trees, which concludes the freeing operation.
Chapter 4 Adaptive Address-Space Implementation

Listing 4.2 Algorithms for allocation and freeing as implemented by the TreeBlock-Allocator.

```
1 alloc(blocks)
2 {
3     outChunk = ceil(SizeTree, blocks)
4     if outChunk == NULL
5         return NULL
6     remove(SizeTree, outChunk)
7     remove(AddressTree, outChunk)
8     trailingChunk = split(outChunk, blocks)
9     insert(SizeTree, trailingChunk)
10    insert(AddressTree, trailingChunk)
11    return outChunk
12 }
13
14 free(currentChunk, blocks)
15 {
16     startAddress = currentChunk;
17     endAddress = currentChunk + blocks * BLOCK_SIZE
18     successor = search(AddressTree, endAddress)
19     predecessor = floor(AddressTree, startAddress)
20     if getEndAddress(predecessor) != startAddress
21         predecessor = NULL
22     remove(SizeTree, successor)
23     remove(AddressTree, successor)
24     remove(SizeTree, predecessor)
25     remove(AddressTree, predecessor)
26     mergedChunk = merge(predecessor, currentChunk, successor)
27     insert(SizeTree, mergedChunk)
28     insert(AddressTree, mergedChunk)
29 }
```

As both functions do not use loops or recursion, the runtime complexity is dominated by the red-black tree operations. All of these operations run in \(O(\log N)\) bounded time, where \(N\) is the number of continuous free chunks in the trees. Therefore, the overall runtime of `alloc` and `free` is also bounded by \(O(\log N)\). As the number of free chunks may reach the total amount of memory in a system \(M\) it is obvious, that the presented algorithms are also \(O(\log M)\), which is the same runtime complexity as the algorithms of the buddy allocator.
The requirement of low external fragmentation is fulfilled as can be observed in Figure 4.6, there, the same allocation patterns are shown as with the buddy allocator. As the TreeBlockAllocator is able to merge adjacent chunks efficiently, the free space in the examples is always continuous and entirely allocatable. Expanding memory also works as expected and leads to a larger memory chunk when the new chunk is adjoining. This leads to the property that external fragmentation only happens when the user of the TreeBlockAllocator causes external fragmentation by not freeing intermediate chunks of memory. Internal fragmentation is limited to the smallest chunk-size minus one byte and is therefore limited by a constant, in contrast to the 50% worst case by the buddy allocator.

Figure 4.6: Internal and external fragmentation of the TreeBlockAllocator.

The concrete implementation realises the presented allocation algorithms with some optimisations. Nodes in the AddressTree must only be removed if no trailing chunk can be split off. If a trailing chunk exists, then the existing chunk is just replaced in the tree by the trailing chunk, as the sorted order of free chunks does not change. This prevents most of the remove and insert pairs on the AddressTree and thus rebalance operations on the tree. Furthermore, chunks of the same size are only represented once in the SizeTree and linked to each other in a doubly linked list. This results in improved runtime complexity in the best case when in the alloc function a part of a chunk is split off. In this case, only a constant amount of work is done on the AddressTree and the ceil search on the SizeTree is bounded by $O(\log D)$ where $D$ is the number of differently sized continuous free chunks. $D$ may be substantially lower than $N$, especially if the user of the TreeBlockAllocator often allocates and frees chunks of recurring sizes. This $O(\log D)$ behaviour should happen often, as the allocation requests rarely happen to be the exact same size as free chunks in the tree. Additionally, the TreeBlockAllocator algorithm is optimistic with regards to the number of free continuous chunks. This means that allocations on a TreeBlockAllocator with one large chunk are fast and only get slower by time when more discontinuous chunks accumulate.
Chapter 4 Adaptive Address-Space Implementation

In contrast, the buddy allocator splits the large chunk in a logarithmic number of free chunks in the first allocation, which yields the worst case allocation time, although the situation should be “simple.”

The design of allocator stacking requires the implementation of allocLargest which should allocate the largest chunk with a given minimum alignment. This task is easy, given that the largest chunk is the rightmost in the SizeTree. To fulfil the alignment properties, the largest chunk may be split in up to three parts, where the first and last part must be re-inserted into both trees.

Listing 4.3  Embedded header for memory chunks managed by the TreeBlockAllocator.

```c
typedef struct LinkNode {
    FreeChunk *prev;
    FreeChunk *next;
} LinkNode;

typedef struct FreeChunk {
    // node for the AddressTree
    RBNode addrNode;

    // node for the SizeTree and doubly linked list
    union {
        RBNode sizeNode;
        LinkNode linkNode;
    };

    // pointer to the linked list of nodes of the same size
    FreeChunk *headNext;

    // size of this chunk
    uintptr_t size;
} FreeChunk;
```

The red-black tree nodes are embedded into the managed memory, the respective structures are shown in Listing 4.3. If a node is found in one of the trees, the address of the containing struct FreeChunk can easily be computed, as they are separated by a constant offset defined by the layout of the structure. FreeChunk is intentionally kept at a minimum size, because this size is the minimal chunk size of the TreeBlockAllocator implementation. For this reason, the doubly linked-list structure forms a union with the node for the SizeTree as they cannot both be in use at the same time and other flags are mangled into pointers of this structure. As the red-black tree nodes are three pointers large, the implementation achieves a size of 8 * sizeof(void*) for the FreeChunk structure, which is 64 bytes on 64 bit systems and 32 bytes on 32 bit systems. This is conveniently the size of cache lines on many hardware architectures and especially on x86-64 and Leon. Allocating small amounts of memory in cache line granularity is beneficial on multicore systems, as this avoids false sharing regarding cache lines.
4.4 Memory Allocation

Figure 4.7: Example state of the TreeBlockAllocator implementation.

Figure 4.7 shows a non-trivial example of the TreeBlockAllocator state, managing 7 chunks. These chunks are visually split into the number of blocks that they are composed of. Chunks with the numbers 1, 2, 3, 5 and 7 have the same size and are therefore linked in a doubly linked list with only one representative in the SizeTree. Sharing of the same FreeChunk in both trees is emphasised for FreeChunk 6, as an example. As chunk number 1 has the lowest address, it is the leftmost node in the AddressTree, respectively as chunk 6 is the smallest, it is the leftmost in the SizeTree. Depending on the order of allocations and deallocations the trees may not be perfectly balanced, but the red-black tree algorithm guarantees $2 \times \log(N)$ height in the worst case.

4.4.2 Allocator Hierarchy

The allocator design presented in Section 3.2 is implemented using the TreeBlockAllocator. Coarse variants of the TreeBlockAllocator manage page sized blocks and fine grained variants manage the smallest possible block size that the implementation supports. A coarse TreeBlockAllocator is created for TLM and SHM, each. It is the responsibility of the architecture dependent initialisation code to fill these allocators at boot time with the free physical pages. Each coarse allocator may directly be used in kernel space to allocate continuous pages. This is particularly the case for the creation of page tables and providing buffers to hardware devices.

The flexible allocator is termed WrapperAllocator as it is implemented as a stateless wrapper around fine-grained block allocators. Aligned allocation is slightly more complicated and more general, so Listing 4.4 emphasises pseudocode for aligned memory allocation.
Enough memory is allocated from the underlying block allocator to store an additional header with the information of the starting address and block size of the allocated block. Additionally, the size must be large enough to satisfy the alignment requirements. This computed size is then rounded up to be a multiple of the block size of the underlying block allocator. The right placement of the header is computed by `writeHeader` which is not necessarily the beginning of the allocated block, depending on the alignment requirements. This is also the reason why the header stores the starting address of the memory chunk. To free memory, the user of the WrapperAllocator may just call `free` without any size information, this corresponds to the C-library allocation interface.

**Listing 4.4** WrapperAllocator implementation to support malloc-like operations.

```c
struct Header {
    void *start;
    uintptr_t blocks;
};

wrapperAllocAligned(alignment, size) {
    allocSize = alignUp(size * (alignment - 1) + sizeof(Header), BLOCK_SIZE)
    rawMem = fineBlockAllocator.alloc(allocSize / BLOCK_SIZE)
    if (rawMem == NULL) return NULL
    Header *header = writeHeader(alignment, allocSize, rawMem)
    return header + 1
}

wrapperAlloc(size) {
    return allocAligned(1, size)
}

wrapperFree(ptr) {
    if (ptr == NULL) return
    Header *header = (Header*)ptr - 1
    fineBlockAllocator.free(header->start, header->blocks)
}
```

If no memory can be allocated from the WrapperAllocator, a larger chunk is allocated from the respective coarse grained allocator and it is added to the fine-grained allocator of the wrapper. After that, the allocation from the fine-grained allocator may be repeated and will succeed. Similarly, if memory is freed to the WrapperAllocator, `allocLargest` is called on the fine-grained allocator to check for a larger chunk, that may be returned to the coarse allocator. These strategies fulfil the requirement formulated in Section 3.2.1 to regularly return memory of adequate size down the allocator hierarchy. This strategy is the same for kernel allocation and user-space allocation. Whereby, the coarse allocator of the user space is represented by the `mem_map` system-calls and the coarse allocator of the kernel is the coarse TreeBlockAllocator, directly.
4.4 Memory Allocation

4.4.3 Garbage Collection for Invasive X10

The garbage collector (GC) used in the original implementation of X10 by IBM was the Boehm–Demers–Weiser garbage collector \cite{33}. This collector was not able to satisfy the requirements defined in Section 3.2.2. Particularly, it was not able to cope with different memory types efficiently, had no support for pinning already allocated objects and was not resource aware with respect to memory allocated from the operating system. A new garbage collector was implemented, as the adaptation of the Boehm GC to the concepts of invasive computing was estimated to be unfeasible.

The new GC is, like Boehm’s, a conservative, stop-the-world GC, that uses a mark and sweep algorithm. A conservative GC may be used in weakly typed programming languages like C or C++, given that pointers to dynamic memory remain unmodified. This conservative property is necessary, since the invasive X10 runtime system, which is written in C, also uses the GC internally. Conversely, the GC implementation may be used by any invasive application.

Conservative collectors are not a novel concept and therefore only the special characteristics of the invasive implementation are outlined. The GC represents a memory allocator with the usual allocation facilities but without the need to provide explicit freeing interfaces. To free memory, the GC needs to scan the roots of object pointers first. In the case of invasive computing these roots are:

- Writeable, static application segments: `app\_X.data` and `app\_X.bss` for application X.
- Dynamic application areas, that are allocated using manual memory allocation, this includes iLet contexts and CPU-local storage.
- Additional memory regions that may stem from other invasive research projects, like iCore memory or scratch-pad memory.

These memory regions are scanned for pointers to objects that the GC returned on allocation requests. All found objects are marked as “still in use” and are themselves scanned recursively. Already marked objects are not scanned again. All other objects that the GC returned are guaranteed to be unused and are therefore deallocated. Accidental bit patterns that look like GC pointers are treated as such and the respective objects are not freed, hence the name conservative GC. To perform the described tasks, the GC must store all returned object addresses and perform efficient lookups for these addresses. Additionally, a maximum and minimum value for the served memory ranges is stored and updated, to quickly reject values in memory that are not pointers. This check alone rejects more than 90\% of the scanned values. The implementation utilises a statically sized hash table for these lookup tasks. When hashes collide, a red-black tree is used for chaining. As an additional optimisation, the GC treats “small” objects in a special way.
Chapter 4 Adaptive Address-Space Implementation

The BulkNode type is used and allocated that can hold many small objects. This BulkNode is then inserted into the hash table instead of all containing small objects, individually. To allow finding a small object in the hash table, the hashed index of each small object within a given BulkNode must be equal. This is achieved by aligning BulkNode objects to their size and using this alignment for large objects, too. The hash function is then \( \text{hash}(\text{addr}) = (\text{addr} >> \text{bulkBits}) \& \text{hashMask} \) where \( \text{addr} \) is the object address, \( \text{bulkBits} \) are least significant zero bits of the alignment of BulkNode objects and \( \text{hashMask} \) selects the valid bits for the power-of-two sized hash table. The disadvantage of BulkNode objects is that only if none of the small objects are referenced, the BulkNode may be deallocated. This effectively trades memory consumption for performance, but as the evaluations show, most of the allocated objects are small and short-lived. Empirically, this does not constitute a memory problem.

Pointers to pinned objects are stored in a separate red-black tree. Allocation of the tree’s nodes happen with a fine grained TreeBlockAllocator, to keep the additional memory fragmentation at a minimum.

Garbage collection may happen at any time when the application explicitly requests it. This is the preferred way, as it allows the application to freely choose when to invest time in the collection work. The application may dynamically or statically configure what happens if the system is out of memory. The standard strategy is that a GC run is performed, but at critical phases this might not be the best decision.

The implementation, like the Boehm GC, stops the execution of application code on the whole tile when the garbage collection starts. The reason for this is the need to scan the iLet contexts which includes register contents and the runtime stack. Additionally, concurrent modifications of memory locations may hide used references to valid objects that must not be freed.

Then the collection procedure proceeds as follows:

- Stop execution of application code on the current tile.
- Scan pinned objects and object roots.
- Iterate and scan through all referenced objects and mark more discovered objects.
- Reset the minimum and maximum address range for valid object addresses.
- For each managed object: if the object is referenced, keep it, reset the marker and update the min/max range, else free the object.
- In the debugging case, statistics are collected and printed.

All objects that shall be freed are collected and merged into larger chunks of memory. After the collection run, these chunks are returned to OctoPOS and are free for other uses and applications. This is also in contrast to the Boehm GC, that may lazily free objects into its own free lists and may never return memory to the operating system.
4.5 Adaptive Isolation

First, this Section recapitulates the central objects and components involved in adaptive isolation. Runtime interference considerations and general problems of isolation-mode switching and their solutions are anticipated to meet the goals defined in the design Section 3. Subsequently, the inner workings of protection-mode switching and important services of OctoPOS which are influenced by it are clarified. Ultimately, particular synchronisation problems and their solutions are explained.

Central to the adaptive isolation are the data structures located in the Application class as well as tile-local classes, objects and hardware components, all of which were already presented. The mode switching and resource-allocation algorithms are explained in terms of these data structures and objects, for this reason it is worthwhile to summarise them before turning to the actual algorithms.

- The page-level BlockAllocator serves the physical memory allocation requests passed through from mem_map and mem_unmap system calls. It additionally provides memory to build the page-table hierarchy.
- As a hardware component, the TLB caches page-table entries that must be flushed manually if the respective entries become invalid.
- The tile-local unprotected mapping that grants access to all available memory irrespective of access rights or ownership.
- For each application its protected mapping that holds all memory regions accessible by the respective application in protected mode.
- Per application: a set of memory areas that represent continuous memory regions with common access attributes implemented as a sorted red-black tree of non-overlapping intervals.
- Also per application: the set of CPU cores as a bit-mask designated to the application by OctoPOS.
- Core-local storage to store a pointer to an application object, representing the application currently running on the core.

4.5.1 Implementation Interference Considerations

The central goal for the adaptive-isolation mechanism lies in the lowest possible interference of the unprotected mode of operation. In the statically unprotected mode this is achieved by utilising the C++ preprocessor and the compiler’s dead code elimination to cut out all protection logic of the system at compile time. Respectively, all presented address-space operations follow the pattern depicted in Listing 4.5 and most of the code resides in headers, to allow a maximum of inlining and optimization by the compiler. The preprocessor macro cf_os_proc_app_prot_none may be set in the build system before compilation to force a statically unprotected system. If set to 1 the compiler performs dead code elimination on the following code and as most of the functions are inlinable, they do not even incur a function call.
The benefits also include that the dead code is still subject to compiler analysis and checks for syntactical correctness. This is in contrast to purely preprocessor based techniques, where the configured-out code is cut out before compilation.

Listing 4.5 Example of the systematic use of the preprocessor and the compiler’s dead code elimination for static configuration.

```c
void addressSpaceExampleFunction()
{
if(cf_os_proc_app_prot_none) {
    return;
}
// address-space code following
...
}
```

For the dynamically unprotected system there exists the central problem of bookkeeping the allocated memory regions at runtime with low impact. On a closer look, this raises problems. The page tables are not modified in the unprotected case. Therefore, when switching from unprotected to the protected mode, not only new memory regions have to be added to the page tables but descriptors to freed pages must also be deleted. Otherwise, protected applications may erroneously access memory that they do not own any more.

This implies, for the dynamically unprotected case, that there are three categories of dynamic memory for which three records must be kept:

- For the original, page-table backed memory-ranges at use when switching into unprotected mode.
- One for newly added memory regions, untracked by page tables.
- Finally, for regions that are referenced by page tables but must be removed when switching into protected mode.

The situation is made more complicated by the fact that the application may free memory regions overlapping these categories and repeatedly allocate and free memory again. Keeping track of these sets of regions introduces a lot of complexity in the implementation as well as overhead in the unprotected case. Furthermore, if the protected mode is never activated again for the lifetime of the application, than all the overhead was in vain.

For these reasons, the implementation follows a different path and shifts some of these overheads to the mode switch from unprotected to protected. The idea is to clear the application’s dynamic regions in the page table and write the active regions anew. The advantage is that only the allocated regions have to be stored and managed when mapping or unmapping memory.
4.5 Adaptive Isolation

The downside is linear cost in the \texttt{protect()} operation in the size of the allocated memory for that application at the time of the \texttt{protect()} call. However, the assumption of this implementation is that the time spend in either protected or unprotected mode exceeds the switching time by far.

Another interference aspect concerns the protected mode operation. When page-table entries are modified, the modification is not automatically forwarded to the TLB in hardware. When the modification increases rights on the address space region, operating systems like Linux do not update the respective TLB entries. This may cause page faults when the memory is accessed, but in these cases the page-fault handler can check the particular page-table descriptor and decide that the access was sound. After that the handler invalidates the entry in the TLB and returns normal execution to the application. The depicted strategy may introduce further unpredictability in the execution of the application and was therefore rejected in this implementation. Rather than invalidating such regions on demand, the implementation immediately voids entries from the TLB once the descriptors are changed in the page-table.

4.5.2 Protection Mode Switching

The switching operations itself have a core obligation, that is: broadcasting the information of which page tables to use in the next protection mode and storing the protection state. The algorithms presented here follow the considerations taken in Section 4.5.1.

Switching from protected to unprotected mode is the simpler of the two operations and is explained first. The operation is assumed to be initiated by the privileged agent system with a call to \texttt{octo_app_unprotect()}. First the memory range encompassing the dynamic regions of application memory are stored in the application object. These are the first and last entries in the sorted set of memory areas of the application. The tile-local mapping must now be applied on each CPU core belonging to the application. For this task the function \texttt{multicastPageMap(mapping, coreMask)} is called. This function uses a facility called IPIExecutor which runs asynchronous jobs on remote cores, it will be explained in Section 4.5.6. For each core in the core bitmap of the application, a job is created to activate the tile-local mapping. These jobs are send to the remote cores using the IPIExecutor and the completion of each job is waited for. After the page table was synchronously activated on all of the application’s cores, the application specific flag \texttt{protectionActive} is set to \texttt{false}. This flag influences dynamic decisions in the memory allocation functions, which will be clarified in Section 4.5.3. This completes the unprotect operation and the application is now able to access all memory defined in the unprotected mapping.

Turning protection on is initiated by \texttt{octo_app_protect()}. First, the application page table is cleared off of the old stored memory range of dynamic memory areas. This is an uncritical operation, as the application page-table is inactive when switching from the unprotected mode.
Afterwards, the page-table is filled with descriptors representing the current set of active memory areas. This set may be changed completely while the application was in unprotected mode and performed allocations and deallocations. After preparation of the application page table, a pointer to it is multicast by utilising `multicastPageMap` to the current set of CPU cores. After this operation, the application-specific flag `protectionActive` is set to `true`.

4.5.3 Protection and Memory Allocation

To benefit from unprotected mode and to enforce isolation in protected mode, memory management on the level of the system calls `mem_map` and `mem_unmap` must behave differently depending on the current mode. Therefore, based on the `protectionActive` flag the implementation proceeds as described below, which is in accordance with the design presented in Section 3.3.

In all cases the page-level memory allocation functions like `mem_map` allocate memory from the BlockAllocator presented in Section 4.4.1. All of the allocation functions then call `addDynamicMemRegion`, passing the region interval. This function determines the current application from the core local storage and accesses its memory-area data structure. If possible, the new area is merged into a neighbouring area with fitting access rights and type. Otherwise, a new memory-area object is allocated and added to the set of areas. As a red-black tree is used for the memory areas, these operations are performed efficiently even for a large number of disjunct areas.

Afterwards, the function `mapDynamicIfNeeded` is called and if memory protection is activated, meaning that the `protectionActive` flag is `true`, the function initialises a page-table descriptor and places it into the application’s page table. The access rights of the descriptor allow read and write access for the application. As set out above, the TLB entries are now invalidated, locally and on the other cores of the application on this tile. For this task `flushTLB` makes use of the IPIExecutor to reach all other cores and propagate the range in question. The function must await the invalidation synchronously to make sure that no core causes a trap on access to the memory.

Each unmapping operation first calls `removeDynamicMemRegion` before returning memory to the BlockAllocator. This function also retrieves the current application object and searches its memory areas to cut out the region in question. This operation on the set of memory areas has higher complexity, as the application may unmap multiple memory areas of different type, access rights and even discontinuous areas (unmapping an already unmapped region shall be valid). It is easy to determine the next lower or equally addressed memory area in the red-black tree. From there on the algorithm iterates over all application memory areas until an area is found with a higher starting address than the end address of the range to be unmapped. All encountered areas are removed from the set and the first and last areas are split if this is necessary.
Another case in the removal of a memory region is that it might split an area into two parts, which requires the allocation of another memory area object to be inserted in the set. Thereafter, `unmapDynamicIfNeeded` conditionally adjusts the access rights in the page-tables of the application and again calls `flushTLB`. This, again, happens only in the case when `protectionActive` is `true`. At last, the memory range in question is freed to the BlockAllocator.

The design constructed in Section 3.3 is satisfied, because the additional overhead in the protected mode is entirely encapsulated in the functions `mapDynamicIfNeeded` and `unmapDynamicIfNeeded` both of which depend on the flag `protectionActive`. This flag is set accordingly in the `protect` and `unprotect` operations. In the statically unprotected case the measures described in Section 4.5.1 are effective in `remove/addDynamicMemRegion` and therefore only the raw memory allocation on the BlockAllocator remains.

As illustrated, there is the need to allocate objects to hold the information about memory areas and also to create page-table hierarchies. These allocations are fed from special object caches, that store objects of the same size in a stack data structure. Whenever such a stack gets empty, new memory is allocated from the BlockAllocator in a large chunk, which is then subsequently split up into equally sized objects, both for memory areas and page tables. This has three advantages:

- Allocations are very fast in most cases, as it only requires getting the top element from the object stack. In rare cases one large chunk must be allocated from the BlockAllocator instead of many little chunks over time.
- Fragmentation of these objects is reduced and does not interfere with other allocations.
- As unmap operations may split one memory area in two, they also require an additional memory area object. Resources for unmapping may be reserved in these stacks, as it is not common for users to assume that an unmap operation may fail due to a resource shortage.

### 4.5.4 Protection and Core Invasion

OctoPOS owns a variety of invade, reinvade and retreat functions all of which were extended to call `registerCoreMask` and `deregisterCoreMask` of the address-space management. Once the application owns a new core, the respective page-table hierarchy must be applied on the MMU of that core, before any application code is executed there. This may be the protected mode application table or the tile-local table. The `register/deregisterCoreMask` functions update the core mask belonging to the application and set the currently active application for the allocated cores in the core-local memories. If the flag `protectionActive` was set, the protected-mode page tables are broadcast to new cores using `multicastPageMap`, in a similar manner as the `protect` function.
Chapter 4 Adaptive Address-Space Implementation

In the case of deregistering cores in the protected mode, the information to apply the tile-local page tables is sent. Especially, no inter core communication happens when the application runs in unprotected modes. Again, this enables faster and more predictable execution in the unprotected case.

As with the other functions, register/deregisterCoreMask can be statically configured away, then the address-space management does not interfere with invade or retreat in any way.

4.5.5 Protection Synchronisation

The goal of the synchronisation strategies of the address-space subsystem is to enable a maximum amount of parallelism in the whole system.

The presented functions regarding memory management and core invasion influence and are affected by protection-mode switching. For example, if core invasion happens in parallel to switching from unprotected to protected mode, then the invaded cores may be in either state, depending which IPI reached first. Similarly, memory allocation may miss a protection mode switch that happens in parallel, resulting in an unpredictable outcome, where the new memory range may not be incorporated into page tables.

Attention should be paid to the fact that these dependencies are solely located within the presented functionality of:

- octo_app_protect
- octo_app_unprotect
- addDynamicMemRegion
- removeDynamicMemRegion
- registerCoreMask
- deregisterCoreMask

Other OctoPOS services may proceed independently, such as agent system and scheduler decisions, iLet dispatching, iLet execution and even memory allocation on BlockAllocator level. It would therefore be a pity to use blocking synchronisation in the address-space subsystem. Unfortunately, all the presented functions may perform multiple steps on multiple data structures to complete, including red-black trees and multilevel page tables. This makes it very complicated to come up with a non-blocking or even wait-free solution. For these reasons, the implementation utilises a novel concept named Guarded Sections published in [12]–[14].

Guarded Sections care for the sequential execution of critical sections and simultaneously permit asynchronous progression after the critical sections. This enables the possibility for asynchronous progression of functionality unrelated to the address-space management in OctoPOS, while guaranteeing synchronisation where it is needed.
4.5 Adaptive Isolation

Figure 4.8: Activity diagram for the Guarded Sections synchronisation algorithm.
Chapter 4 Adaptive Address-Space Implementation

The general algorithm of Guarded Sections follows an entry protocol called *vouch* and an exit protocol called *clear*. In Figure 4.8, the operations are combined to a common *GuardDefer* function. Guard objects consists of a special queue and an atomic flag variable. As a first step, the requested operation is encapsulated into an Order which may be for example a function pointer and arguments. This Order is then unconditionally enqueued into the guards queue. Afterwards, the entry protocol checks whether the guard is already occupied, if so, the thread of control may proceed. If it is the first to arrive at the guard, it marks the guard as locked and processes Order objects from the queue, including its own order by dequeuing elements. If no Orders are left, it executes the exit protocol by unmarking the guard and checking afterwards if the queue is still empty. If it is not empty it must try to reacquire the guard.

The implementation of Guarded Sections is especially effective, since the queue operations are wait-free and therefore the entry and exit protocol is also wait-free with respect to synchronisation. This is possible through a novel wait-free multiple-enqueuer single-dequeuer queue, developed in unison with Guarded Sections. The full source code is available in Appendix A.1.2. It must be noted that the enqueue operation is not atomic from the perspective of an outside observer, however, in the case of guarded sections this is handled by the order of checks and with the help of the additional lock flag.

4.5.6 Address-Space Subsystem Interrupt Synchronisation

All interrupt-based communication of the address-space subsystem is handled utilising the IPIExecutor class. This comprises the page-table notification and TLB flushing operations. Both operations represent multicast operations from one origin to one, multiple or all cores, depending on which cores belong to the application at runtime. In addition, both operations are used synchronously, meaning that only when all the receivers acknowledge the reception, may the sender continue.

A novel software-based and synchronous on-chip multicast on this level of operation was researched and evaluated in “Diamond Rings: Acknowledged Event Propagation in Many-Core Processors” [11]. The presented algorithm “Diamond Rings” was also implemented for the address-space subsystem IPI communication.

The idea is to split all nodes in three groups, see Figure 4.9. The first group spans a tree with the sender as its root, this tree terminates in the second group that forms the leaves of the tree. The third group shapes a diminishing tree with the second group as leaves and, again, with the sender node as its root. The information is now passed from the sender down the diamond shaped structure until the sender itself receives messages from the last nodes of the reducing tree. Afterwards, the sender can be sure that all nodes received the message and, therefore, may synchronously continue. Apparently, there are many parameters that may be configured such as the degree of the trees, or even whether the nodes form graphs other than trees.
On an empirical basis, the x86 inter-processor interrupts run for a comparatively long time, are asynchronous and many may be sent to different cores at the same time. As a consequence, the degree of the trees in the Diamond Ring may be very large and ultimately the algorithm becomes indistinguishable from a simple linear multicast. To avoid the overheads of the computation of the three sets of cores, which must be done at runtime, the Diamond Ring algorithm was simplified to a linear multicast that exploits the hardware asynchronicity. The evaluations show the efficiency of this approach.

Particular important functions that use the IPIExecutor are `multicastPageMap` and `flushTLB` functions. For each receiving core an `IPITask` is initialised which consists of a linked-list pointer, a function pointer and arguments to the function pointed to. In the case of `multicastPageMap` one argument is passed which is the pointer to the page table to activate. In case of `flushTLB` this is the memory range to invalidate. This IPITask is then passed to the IPIExecutor’s `triggerJob` function, see Listing 4.6, which enqueues it in a remote core’s queue and triggers an asynchronous IPI. The loop continues and overlaps the time it takes to deliver the IPI while initialising another IPITask and enqueuing it in another core’s queue. Remote enqueueing happens in the same kind of novel wait-free queue that was presented in Section 4.5.5. As the implementation is wait-free, no further synchronisation is needed for the IPIExecutor and therefore for `flushTLB` and `multicastPageMap`, too.

The interrupt handler of the memory-subsystem IPI then executes the `run` function of the IPIExecutor, see Listing 4.6. It is enough to dequeue and call the jobs without any further attention to the lost wakeup problem, as pending IPI requests will trigger the handler even if another job is enqueued right after the decision to leave the loop.
Listing 4.6 IPIExecutor: ’run’ executes jobs in interrupt context while ’triggerJob’ may be called synchronously from privileged mode.

```c
void run()
{
    IPITask *job;
    while((job = queue.dequeue()) != nullptr) {
        job->func(job);
    }
}

void triggerJob(IPITask *job, uintptr_t otherCPU)
{
    // enqueue job on remote queue
    executors[otherCPU].queue.enqueue(job);

    // send IPI afterwards
    os::dev::IRQController::Inst().triggerIRQ(otherCPU, MEM_IPI_VEC);
}
```

The wait-free queue may be used because dequeueing only happens per core in the interrupt handler, which is the ideal situation for this type of queue. Multiple concurrent enqueuers are also certainly supported.

4.6 Inter-Tile Communication

Many variants of the adaptive inter-tile communication were implemented in Martin Gumbrecht’s master thesis [34]. The implementation details are not repeated here, but some specifics are highlighted. All variants follow the general design envisioned in Section 3.4 and integrate into the aspect-oriented configurability of OctoPOS. For this, it uses AspectC++ and like the adaptive memory protection system, this system may be shut off statically at compile time. It intervenes in the `dispatch_claim_push_dma` functionality and, when activated, checks remote access rights before issuing the asynchronous hardware DMA.

The validation of access rights to certain memory regions using DMA is done on the page-table data structures itself. Where also a marker is set on freely available bits in the descriptors to reserve these page until the DMA operation is finished. As the number of these bits is finite, a further waiting mechanism is implemented to wait for access.

Additional peculiarities exists in the implementation of the “cache” and “registry” that store acquired and granted permissions. In the implementation, these are bounded and, depending on the size, the underlying data structure is configurable to be array based or red-black tree based. An “unlimited” cache would be subject to resource shortages at runtime and DMA operations may fail due to the lack of resources.
A downside of the limited cache is that evictions must happen if the cache is full, this may lead to unpredictable delays in further DMA accesses. However, as the cache is not used in unprotected mode, they do not contribute to unpredictability in this case. Section 5 grants insights into the evaluation of the adaptive inter-tile communication system.

4.7 Standard MPI Implementation

Above the asynchronous and low-level dispatch_claim_push_dma functionality a standards compliant MPI [35] library was implemented. The main goal of this was to easily support applications that are already implemented utilising MPI. Amongst those are the NASA Advanced Supercomputing Division (NAS) [28] benchmarks. These benchmarks became very useful in debugging the overall invasive research system and served in many thesis’ in the context of the CRC as evaluation benchmarks. They will also provide insights into the performance of the dynamic inter-tile protection.

The MPI standard names the participants of communication “processes.” In the case of OctoPOS these are iLets on DispatchClaims that belong to the same application but may or may not be on the same tile. This implementation of the MPI library has a two layer architecture. On the lower layer the basic communication functions build an abstraction over OctoPOS’ DMA interface. This layer makes heavy use of invasive features like iLet infections locally and on remote tiles and blocking using the SimpleSignal mechanisms. These lower level functions are for example MPI_Send, MPI_Recv for sending and receiving, their asynchronous variants MPI_Isend, MPI_Irecv and MPI_Wait for waiting on the asynchronous functions to finish. MPI defines communicators, these are identifiers for a group of processes and in the case of OctoPOS, a group of DispatchClaim objects. Communicators may be duplicated and split at runtime. Within a communicator every process is identified by a unique id from 0 to \( N - 1 \). Processes may query the size of a communicator using MPI_Comm_size and their own id with MPI_Comm_rank. OctoPOS’ MPI library maintains a mapping between communicators, ids and the respective DispatchClaims in tile-local data structures. MPI_Barrier waits for all processes in a communicator to reach the barrier only after which execution is continued. This function is implemented as a tree barrier utilising atomic counters, iLet infections and SimpleSignal synchronisation. For each barrier, each process owns a bottom-up atomic counter and a top-down SimpleSignal. Each process and its children decrement the process’ counter, once it reaches zero, an iLet is send to the parent to decrement its counter and the process waits on the top-down signal. Once this chain reaches the root node, the top-down signaling starts to wake up the root’s children, they wake their children, an so forth. After signaling all processes down the tree, execution can be continued as the barrier is complete. This algorithm is also depicted in Figure 4.10 and its detailed implementation may be found in Appendix A.1.3.
Figure 4.10: Bottom-up counting of arrived processes and top-down signaling to continue execution in the MPI barrier implementation.

As these lower layer functions are implemented directly using invasive concepts of OctoPOS, they benefit from the optimised iLet infection functions, asynchronous DMA and lightweight blocking abilities of SimpleSignal objects.

Each send and receive operation gets a tag integer, that must also match when delivering messages. Negative tags are reserved for the MPI implementation and the OctoPOS’ MPI library uses negative tags to implement the higher level “collective” MPI operations, based on the basic operations presented before. These include, among others, \texttt{MPI\_Bcast} and \texttt{MPI\_Reduce}. \texttt{MPI\_Bcast} broadcasts a message within a communicator and \texttt{MPI\_Reduce} applies a global reduce operator that must be both associative and commutative on a set of distributed arrays of numbers. The collective operations also implement a tree shaped communication pattern, either top-down or bottom-up. This has the advantage that communication finishes in a logarithmic time scale relative to the number of processes involved, as long as the communication medium is not saturated. Further, the concept of Diamond Rings as described in Section 4.5.6 was considered but rejected. First, \texttt{MPI\_Bcast} is defined to be asynchronous, so once the root processes finished in participating in the broadcast, it may continue to execute before all processes received the message. Second, processes finished receiving the broadcast may start another broadcast while the first is still active. A tree shaped communication pattern allows for multiple broadcasts to be active, which means that the throughput is higher than with Diamond Rings.
As the collective operations are built on top of the lower level operations, they are more portable to other MPI implementations but only indirectly profit from invasive concepts. Therefore, programs written solely against the MPI interface only implicitly benefit from the invasive software and hardware architecture. However, this is expected as MPI is not a native invasive programming model.
The evaluation provides insights and comparisons of measured features of the adaptive address-space management system, its components and its effect on the overall system. Additionally, a study on predictability implications is outlined for an example application and as a microbenchmark. As an overview, this chapter consists of three parts:

The overall methodology and surrounding hardware and software parameters are made clear in Sections 5.1 and 5.2. The goal of this is to facilitate reproducibility as far as possible and allow the understanding of the results’ certainty. The statistical methodology is outlined to substantiate the assertions regarding evaluation outcomes. It is detailed how measurements were performed including the hardware platforms in question and the software used to compile iRTSS and applications.

Microbenchmarks are presented in Section 5.3. They show performance characteristics for invasive operations affected by the address-space management system including: resource invasion, page-based memory allocation and means to activate and deactivate memory protection. An evaluation of performance characteristics of the red-black tree implementation is included, as this is a central data structure in the system.

Results for application benchmarks conclude this chapter in Section 5.4. They are mainly comprised of X10 applications that were also partially used in evaluations for the dissertations: [20]–[22]. Therefore, these benchmarks are not arbitrary chosen but comparable and form an exhaustive suite of available software in the invasive context. The general goal is to compare the different modes of operation with respect to memory isolation and their effect on the application’s performance.

The scientific and experimental nature of some components, like the AspectC++ compiler, all parts of iRTSS and parts of the Sparc Leon hardware cause some limitations. Because of these limitations, hardware bugs and software defects not all benchmarks could be performed on all platforms, it is explicitly mentioned when this is the case.
Chapter 5 Evaluation

5.1 Hardware and Software Parameters

This Section informs about the hardware and software versions used for evaluation. Table 5.1 presents specifications for the Sparc and x84 machines. Leon is a representative for the invasive reference architecture with many research components as illustrated in Section 4.1.2. As it is run on an FPGA, the clock frequency is limited to 50 MHz. To include typical COTS hardware the second machine is Intel8 which only has one NUMA domain and hence only one tile when run with iRTSS. The machine termed Intel80 represents typical x86 severs with multiple NUMA domains, two-digit number of CPU cores and large main memory.

The compilers used are GCC/G++ in version 7.5.0 and for the Sparc platform in version 8.2.0, AspectC++ compilers ag++ 0.9, ac++ 2.2 (based on clang 3.8.0). Three configurations of iRTSS were taken for all evaluations that correspond to the operation modes described in Section 3.3. These will be referenced in the upcoming sections:

- **Static off** (St-off): The original iRTSS implementation where this work is based on. Section 4.5.1 introduced the required static configuration changes. This version is taken as a base reference with no notion of isolatable applications or the possibility of memory isolation in place.
- **Dynamic off** (Dyn-off): The adaptive memory management system is doing basic bookkeeping about applications’ memory to be able to turn memory protection on at any time. But memory protection itself is turned off at runtime for the application or microbenchmark in question.
- **Dynamic on** (Dyn-on): Memory isolation active, the evaluated applications can only access allocated and permitted regions of memory.

Time is measured on x86 platforms using the rdtscp instruction according to an Intel white paper [36] on time measurement. This “read time-stamp counter” instruction has undergone many improvements and changes in behaviour in the history of x86 processors. In all machines used for evaluation purposes, this instruction reads a 64 bit counter value that is incremented at a very high speed. This speed is nearly the clock frequency of the processor, except that it remains constant under all circumstances, especially with respect to dynamic frequency scaling and sleep states. For these reasons it is not suited to measure the number of executed instructions, but is rather a very high resolution timer. Small timings are therefore given in nanoseconds instead of an instruction count.
Table 5.1: Hardware specifications of the (FPGA-based) Sparc and x86-64 machines used for evaluation. NUMA nodes are equivalent to tiles in the implementation.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Leon</th>
<th>Intel8</th>
<th>Intel80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Sparc V8</td>
<td>Intel Xeon</td>
<td>Intel Xeon</td>
</tr>
<tr>
<td>Variant</td>
<td>Leon3</td>
<td>E3-1275 v3</td>
<td>E5-4640 v2</td>
</tr>
<tr>
<td>Release</td>
<td>1997</td>
<td>2013</td>
<td>2014</td>
</tr>
<tr>
<td>Frequency</td>
<td>50 MHz</td>
<td>3.5 GHz</td>
<td>2.2 GHz</td>
</tr>
<tr>
<td>Cores,Threads</td>
<td>16,16</td>
<td>4,8</td>
<td>40,80</td>
</tr>
<tr>
<td>NUMA Nodes</td>
<td>-</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Tiles</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>RAM</td>
<td>1 GiB</td>
<td>32 GiB</td>
<td>128 GiB</td>
</tr>
<tr>
<td>TLM</td>
<td>8 MiB</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

For the Sparc Leon platform time is measured utilising a timer within the invasive network adapter that connects to the invasive network-on-chip. This timer is synchronized across tiles and enables inter-tile time measurements.

5.2 Statistical Method

Each benchmark with a specific software or hardware configuration was measured many times. To provide transparent results, not only the mean values are reported but a statistical analysis is performed. This analysis serves the purpose to substantiate the validity of the results and allow others independent examination and conclusions. Recommendations from Hoefler and Belli [37] regarding scientific benchmarking of parallel computing systems were followed as appropriate. Furthermore, the reproducibility of these results may be hard in future, if not impossible, because the required computing systems, their operation, availability and required software may not be available or operational any more. In [37] this situation is addressed by suggesting to keep measurements interpretable from a statistical point of view, if reproducibility cannot be guaranteed. To make this interpretability possible, calculations of sample variation are conducted for all measurements. Particularly, plots and tables highlight mean and variation of the measurements.

5.3 Microbenchmarks

Microbenchmarks show the overhead of individual operations within the invasive system. All microbenchmark are conducted sixteen times to gather information about the variability of the measurements and to be able to effectively compare the operations in protected and unprotected mode. For the graphs, measurements were aggregated at each x value and plotted are the mean and the 95% confidence interval around the mean.
So, 95% of all measurements reside in the brighter area, this gives an indication on the variability of the measurements. The statistical computations were performed on the raw measurements using the Python SeaBorn 0.11.1 and SciPy 1.6.3 in default configuration.

Special care needs to be taken when collecting many values in one run of the system. A part of these operations include memory management or depend to some extent on memory allocation. Therefore, the measurement would be heavily influenced when dynamic memory allocation would take place during the benchmark runs to store timings. For this reason, statically allocated buffers are used to record measurements and the output of the data happens after the benchmark run. Similarly, the memory allocators are cleaned up to only contain the largest continuous free block of memory at the start of the microbenchmark.

As timings can become small, the influence of the measurement overhead increases. On all platforms and evaluation machines variations of the measurement overhead exist. Due to that, it is not safe to subtract a fixed number from the measured quantities. The differences between protected and unprotected modes are large enough to get an idea of the performance characteristics without performing corrections.

In many cases the benchmarks behave quite similar on different architectures. For this reason only the results for Leon and Intel80 are presented and discussed as representative examples. In Appendix A.2 all measurements and plots for all remaining cases are available.

On x86 platforms and for certain microbenchmarks, a special mode of OctoPOS was used to treat all CPU cores as a single tile. This has the advantage that the scaling of tile-local operations can be shown not only to the limit of the local NUMA domain but for a maximum of 64 cores. This number (64) is currently a technical limit of OctoPOS and its schedulers.

5.3.1 Invasive- and Memory Management Operations

Figures 5.1 and 5.2 show timings for invade and retreat operations on Leon, the respective graphs for Intel80 are Figures 5.3 and 5.4. The horizontal axis shows the number of simultaneously invaded or retreated cores. This axis amounts to \( N - 1 \) cores as the microbenchmark application already starts on one core. Invade timings largely resemble retreat timings because the actions taken in the address-space management system are similar. A slight linear overhead is associated with core allocation, both, in OctoPOS and in the address-space management. This stems from the bookkeeping of which core belongs to the current application and the data structures that allow core-local access the current application object in OctoPOS. When retreating, this core-local variable is not reset. For this reason, the difference between the Dyn-off and St-off variants is smaller when retreating. The overhead for the Dyn-on variant is much larger as synchronous inter-processor interrupts must be sent to the cores to activate or deactivate the page-table mapping.
Figure 5.1: Invade timings for the Leon architecture. Right-hand side: zoom on the “off” variants.

Figure 5.2: Retreat timings for the Leon architecture. Right-hand side: zoom on the “off” variants.
Chapter 5 Evaluation

Figure 5.3: Intel80 measurements detail how large the gap between static-off and dynamic off variants may be, even though both employ linear growth.

Figure 5.4: Invasive retreat operation, measured time for Intel80.
Deactivation on retreat also happens to not affect other applications that may be in *Dyn-off* mode. So in turn this allows *Dyn-off* to be as fast as it is for invade operations. The *Dyn-on* configuration exhibits larger relative variation. This can be attributed to the additional memory accesses and wait timings that are needed to wait synchronously for the inter-processor interrupts to finish, as well as the time to switch page-tables on hardware side. The actual amount of memory owned by the application does not directly play a role for the timings when invading or retreating.

The following evaluations are made with regard to two dimensions: memory pages and CPU cores allocated to the benchmark program. Beginning with *unprotect* Figures 5.5a and 5.5b show a similar graph with respect to the benchmark’s cores as retreating from these cores. The reason is that nearly the same operation takes place: set the MMU to disable memory protection via inter-processor interrupt. Here, it is also clearly visible that this does not depend on the allocated application-memory size. A special case is with one CPU core, no interrupt is required and only the local MMU is reprogrammed, accordingly, this is much faster. The right-hand side Figures show this situation with the two dimension flipped, clearly showing the large difference between one and two cores. By contrast the *protect* operation is also linearly dependent on the number of pages, visible in Figures 5.6a and 5.6b. This is the case, because of the implementation decision taken to move some of the page-table management overhead to the protection operation for lower latencies in unprotected mode. Concretely, the application’s dynamic regions are cleared in the page table and the currently active regions are written anew, see Section 4.5.1 for the implementation considerations. These results are similar for all architectures, see Appendix A.2 for all considered platforms.

These insights for *protect/unprotect* favour the strategy to start an application in the safe protected mode by default and give it the possibility, within the agent system, to switch to unprotected mode in its initialisation phase. Therefore, incurring only a low constant overhead irrespective of its (initial) static memory usage.

The main system calls to manage page-based memory in user space are *memmap* and *memunmap*. Their performance is similar for equal reasons, so only *memunmap* is explicitly interpreted and presented. These operations combine both linear overheads with respect to the number of CPU cores and amount of memory allocated, see Figures 5.7, 5.8a, 5.8b for *Leon* and Figures 5.9, 5.10a, 5.10b for *Intel80*. When pages are unmapped from the protected application’s address space, the particular page-table entries must be adjusted, which has a linear dependency on the magnitude of memory. Additionally, the corresponding entries from TLB caches on all application’s cores are invalidated. Again, for single cored applications the inter-processor interrupt can be omitted. A very interesting split between different growing rates of the measurements depending on the number of application pages can be observed. Below 32 pages the runtime grows much faster to drop again at that number of pages.
Figure 5.5: Unprotect operation for Leon and Intel80. The horizontal axis on the left shows increasing number of cores plotted for a selection of active pages for the benchmark application. The right side plots pages with graphs for different number of cores.
Figure 5.6: Protect operation for Leon and Intel80. The horizontal axis on the left shows increasing number of cores plotted for a selection of active pages for the benchmark application. The right side plots pages with graphs for different number of cores.
Chapter 5 Evaluation

The reason for this is a heuristic in the TLB invalidation functionality. Amounts of page addresses lower than 32 are invalidated individually, while larger amounts are invalidated by flushing the whole TLB. On one hand this makes the operation itself much faster, but on the other hand it may cause a temporary slowdown in the application, as new TLB entries must be fetched. This heuristic number is based on a similar strategy employed in the Linux kernel. The results for the off cases are very low and constant overheads. These overheads stem from the BlockAllocator and in the case of Dyn-off, also the bookkeeping of the applications memory ranges. Leaving out this bookkeeping reduces the runtime further by a factor of 1.5 to 2.

![Figure 5.7: Unmap operation on Leon for protected mode.](image)

Finally, to bring the results into the right perspective, `munmap` performance on Linux was measured on the same Intel80 machine. The reason for this is to prove that the protected case is not made intentionally slower to make the unprotected cases look disproportionally faster. The graphs and numbers can be found in Appendix A.2. In summary, the overall behaviour with respect to the number of CPU cores and pages is similar and also approximately linear, however the overall times are 10 times slower than using OctoPOS’s allocation functions in full dynamic and protected mode. The point here is not to argue that the implementation within OctoPOS is faster than Linux’ implementation, which is much more elaborate and provides much more features on more architectures. But to show that the same linear behaviour with respect to the number of pages and CPU cores exists and the running time is within reasonable bounds. Ultimately, even the protected-mode implementation for OctoPOS seems to be very fast in the overall context.
5.3 Microbenchmarks

(a) Leon dynamically unprotected.

(b) Leon statically unprotected.

Figure 5.8: Unmap operation on Leon for unprotected cases, dynamic and static.


5.3.2 Runtime Predicability Effects

To study the impact on predicability a small microbenchmark was conducted. Figure 5.11a illustrates the algorithm. An application runs on two cores, on core one repeated allocations and deallocations of page size are performed using memmap and memunmap. On the other core a repeated computation takes place in work() and the time is measured to complete this computation. In this benchmark the computation is a long-running loop, always performing the same calculations. It is expected that the computation always takes approximately the same time. This is true for the off modes of memory protection. Figure 5.11b shows the results when memory protection is repeatedly turned on and off on a third core for this application with some delay in between to make the result more distinguishable. The measurement starts with enabled memory protection which is then disabled this is repeated another time. It is clearly visible that a scattering of the measured values take place once memory protection is turned on. As it is apparent from the implementation, inter-processor interrupts are send when entries in the TLB caches must be invalidated. This happens to all cores where parts of the application run. By doing so, “unexpected” delays of different parts of the application may take place. The described effect is visualised in this plot. Conversely, unprotected applications that run on multiple cores benefit from a more stable runtime with respect to this effect. Finally, more precise predictability measures can be made, as the degree of parallelism and dynamic memory usage have less influence on applications’ runtime on other cores.
5.3 Microbenchmarks

Figure 5.10: Unmap operation on Intel80 for unprotected cases, dynamic and static.
Figure 5.11: Study on the jitter caused by repeated map/unmap calls, while the same application does different calculations on another core. The time for \texttt{work()} should always be the same, but is disturbed when memory protection is turned on.

Figure 5.12: C1 demo of the second CRC inspection. Users can push “Memory Protection ON/OFF” buttons that forward this to the iRTSS system, while that system computes ray-traced images. The total render time and time for the first line do jitter, because of enabled memory protection.
For the successful second inspection of the CRC Invasive Computing a special demonstration program was implemented that showed this effect within a real application. A ray tracing program computed an animation of reflective and textured spheres rotating around a common centre, see Figure 5.12. This application ran on a x86-64 machine with iRTSS/OctoPOS as runtime and operating system. The pictures were sent to a Linux system via an Ethernet connection and displayed using a graphical UI. To compute the resulting pictures of the animation, memory allocations and computations partially overlap, potentially on different cores of the application. Timings were measured for the overall computing time of the picture and for the first line, all on iRTSS side. Measured times vary because the computational effort changes, depending on the visibility of objects. The time for the first line is expected to be always the same as no object reaches this point. Users of the UI could then enable and disable memory protection on the OctoPOS system by clicking on buttons of the user interface. This was forwarded via Ethernet to the iRTSS test system. The visible outcome reproduces the results of the previous example in an interactive way. More importantly, the quality of the rendering could be adjusted in a way that when turning on memory protection, the jitter caused visible lags in the animation. This highlights the effect in a strong and interactive manner. As a side effect, the dynamic nature of the adaptive memory-protection system could be tried out and increased the confidence of the evaluators in the robustness of the dynamic system.

5.3.3 Red-Black Tree vs Sorted List

As a central part of the memory allocators and memory ranges, red-black trees play an important role in the adaptive address-space management system. The asymptotic run times were discussed for the implementation but the question remains how the absolute runtime compares to a much simpler list-based approach. For this comparison a sorted, doubly-linked list was implemented. It is doubly linked to allow efficient removal of items if a reference on this item is retrieved from somewhere else, which happens in the memory allocators. Also the requirement of being sorted stems from the need to efficiently find the next lower or higher item in the memory ranges and allocators for merging.

As this shall not be an extensive study on general data structures, only two benchmarks were performed. In Figures 5.13 and 5.14 the time to insert a random integer into the data structures is measured with respect to each data-structure’s size. As expected, the list-based approach shows linear growth and the tree-based solution grows logarithmically. Similar results can be observed for the second benchmark, where the task is to find a random item which was previously inserted and to remove it. Removal results resemble the insertion results and for this reason the plots are attached in Appendix A.2. The interesting part is the cross-over points of these functions, as this defines the point until a simple list is really faster despite its asymptotic run-time complexity. For Leon this is around 13 items and on Intel80 the junction is approximately at 80 items.
Figure 5.13: Insertion time on *Leon* for a single item into the data structure with increasing size.

Figure 5.14: Insertion time on *Intel80* for a single item into the data structure with increasing size.
To determine the best decision for given applications, the X10 applications, which are introduced in Section 5.4 were run with enabled statistics for the sizes of the allocator and memory-area data structures. In Tables 5.2 and 5.3 it is shown that the relevant data structures really hold more elements than the maximum number of these crossing-over points most of the time across benchmarks.

Table 5.2: Statistics for the X10-based benchmarks on the *Intel80* platform, showing the number non-continuous items in the memory-areas data structure.

<table>
<thead>
<tr>
<th>Application</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>stddev</th>
<th>&gt;80 items</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfsBellmanFord</td>
<td>4</td>
<td>136</td>
<td>63</td>
<td>54</td>
<td>42%</td>
</tr>
<tr>
<td>bfsDijkstra</td>
<td>4</td>
<td>1751</td>
<td>366</td>
<td>331</td>
<td>76%</td>
</tr>
<tr>
<td>byzantine</td>
<td>4</td>
<td>13357</td>
<td>5242</td>
<td>2543</td>
<td>99%</td>
</tr>
<tr>
<td>dominatingSet</td>
<td>4</td>
<td>3934</td>
<td>597</td>
<td>568</td>
<td>94%</td>
</tr>
<tr>
<td>kcommitte</td>
<td>4</td>
<td>1079</td>
<td>522</td>
<td>365</td>
<td>81%</td>
</tr>
<tr>
<td>leader_elect_dp</td>
<td>4</td>
<td>2878</td>
<td>878</td>
<td>564</td>
<td>91%</td>
</tr>
<tr>
<td>leader_elect_hs</td>
<td>4</td>
<td>11308</td>
<td>1541</td>
<td>1991</td>
<td>99%</td>
</tr>
<tr>
<td>leader_elect_lcr</td>
<td>4</td>
<td>7735</td>
<td>1692</td>
<td>1723</td>
<td>99%</td>
</tr>
<tr>
<td>mis</td>
<td>4</td>
<td>1840</td>
<td>238</td>
<td>383</td>
<td>63%</td>
</tr>
<tr>
<td>mst</td>
<td>4</td>
<td>3084</td>
<td>1167</td>
<td>821</td>
<td>96%</td>
</tr>
<tr>
<td>multigrid</td>
<td>4</td>
<td>5125</td>
<td>2357</td>
<td>2478</td>
<td>46%</td>
</tr>
<tr>
<td>vertexColoring</td>
<td>4</td>
<td>1458</td>
<td>186</td>
<td>295</td>
<td>81%</td>
</tr>
</tbody>
</table>

Table 5.3: Statistics on the number of on-continuous blocks managed by the coarse-grained BlockAllocator on the *Intel80* platform, while running the X10-based benchmarks.

<table>
<thead>
<tr>
<th>Application</th>
<th>min</th>
<th>max</th>
<th>mean</th>
<th>stddev</th>
<th>&gt;80 items</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfsBellmanFord</td>
<td>3</td>
<td>208</td>
<td>79</td>
<td>64</td>
<td>47%</td>
</tr>
<tr>
<td>bfsDijkstra</td>
<td>3</td>
<td>1841</td>
<td>339</td>
<td>351</td>
<td>76%</td>
</tr>
<tr>
<td>byzantine</td>
<td>2</td>
<td>13350</td>
<td>5221</td>
<td>2548</td>
<td>99%</td>
</tr>
<tr>
<td>dominatingSet</td>
<td>3</td>
<td>3980</td>
<td>485</td>
<td>618</td>
<td>60%</td>
</tr>
<tr>
<td>kcommitte</td>
<td>3</td>
<td>1083</td>
<td>522</td>
<td>365</td>
<td>79%</td>
</tr>
<tr>
<td>leader_elect_dp</td>
<td>2</td>
<td>2965</td>
<td>899</td>
<td>585</td>
<td>90%</td>
</tr>
<tr>
<td>leader_elect_hs</td>
<td>3</td>
<td>11274</td>
<td>1526</td>
<td>1987</td>
<td>99%</td>
</tr>
<tr>
<td>leader_elect_lcr</td>
<td>2</td>
<td>7770</td>
<td>1684</td>
<td>1721</td>
<td>99%</td>
</tr>
<tr>
<td>mis</td>
<td>3</td>
<td>1873</td>
<td>258</td>
<td>388</td>
<td>85%</td>
</tr>
<tr>
<td>mst</td>
<td>3</td>
<td>3106</td>
<td>1172</td>
<td>830</td>
<td>96%</td>
</tr>
<tr>
<td>multigrid</td>
<td>3</td>
<td>5127</td>
<td>2355</td>
<td>2471</td>
<td>77%</td>
</tr>
<tr>
<td>vertexColoring</td>
<td>3</td>
<td>1495</td>
<td>201</td>
<td>303</td>
<td>85%</td>
</tr>
</tbody>
</table>
For this reason the decision was made to utilise the red-black tree in favour of a simpler approach. One can even argue that with dynamic and previously unknown applications, a robust data structure to handle arbitrary amounts of discontinuous regions of memory or memory ranges, is a foresightful choice.

Interestingly, the timing variation on the list-based implementation increases with larger numbers of items. The reason for this is that inserting an item into a sorted list may stop very early or very late while iterating over the list. The order of magnitude for this is the whole list size. However, the red-black tree always performs operations in the order of the logarithm of its elements. The impact of this is that the measured time is very similar, independent of the actual value of the item. This could be also interesting from a predictability point of view and eventually supports the previously presented low variation results.

5.4 X10 Benchmarks

These benchmarks measure the running time of whole applications. The resulting measure is done within the applications, as all of them are designed to verify their computed result and measure the main part of the respective application logic. The purpose of these measurements is to analyse the overall effects of the adaptive address-space management system with the three modes of statically and dynamically turned off memory protection, as well as activated memory protection. The inter-tile protection implications are analysed separately in Section 5.5. In the following the benchmarks are introduced and references are given to the sources.

The X10 application benchmarks are a suite of type-safe programs. They are candidates for disabled memory protection, because the application cannot escape its allocated objects. This is of course only true if the compiler and X10 runtime system function correctly in this regard and can be trusted. These applications are from IMSuite 8, the “IIT Madras benchmark suite for simulating distributed algorithms” [38]. The suite consists of twelve applications from the field of graph, networking and optimisation algorithms, implemented in a distributed manner.

All applications were executed ten times with each of the three OctoPOS configurations. Figures 5.15 and 5.16 show the measurements on the evaluation platforms as bar graphs. The black line on top of a bar indicates the confidence interval as discussed in Section 5.3. The graphs are normalised with respect to the mean value of the respective Dyn-on case, so all times are given in percent relative to this value. Although the difference between the Dyn-off and St-off variants is small in some benchmarks, it is still noticeable. The gap between protected and unprotected modes is between 2% and 20% and most of the time at about 5%. The large variations in percentage can be attributed to different amounts of memory allocation and deallocation, which is entirely application specific. Furthermore, the off variants are always strictly faster than the Dyn-off variant.
Figure 5.15: X10 application benchmarks on Leon.
Figure 5.16: X10 application benchmarks on Intel80.
5.5 NAS Parallel Benchmarks

The NAS Parallel Benchmarks [28] in version 3.3.1 are a standard benchmark suite developed and maintained by the *NASA Advanced Supercomputing Division*. In the MPI variant these benchmarks use parallel and isolated processes with message passing as a sole communication mechanism. The benchmarks consist of Fortran and C programs for physically-based computations that arise in the field of fluid dynamics.

The applications have two static configuration properties that are the degree of parallelism and problem size. The problem sizes are not clearly defined and have individual meaning for each benchmark but generally double the amount of memory used from A to F, in alphabetical order. The problem size was kept at level “C” which is a large configuration typically filling two figure GiB amounts of memory. Different degrees of parallelism were tested but the overall result are comparable. The timings do not differ between the tile-local protection modes. These application do not perform dynamic allocations and do not change their initial claim size or structure.

Figure 5.17: NAS FT-64 benchmark on *Intel80*. The unprotected case was only measured ten times and is repeated here as a baseline for the protected variant.

However, the benchmarks are subject to the inter-tile protection mechanism as they make heavy use of the push-DMA functionality. It is therefore worthwhile to analyse the behaviour with respect to protection and different permission-cache sizes.
Chapter 5 Evaluation

The effects of switching DMA-based communication protection mechanisms on and off are analysed in [34] in detail for different implementation strategies and will not be repeated here.

Figure 5.17 shows measurements for the Fourier Transform (FT) benchmark with 64 MPI “processes” and an increasing cache size. The behaviour is very similar across all NAS benchmarks and would just repeat the results already presented in [22] and [34] so they are omitted for brevity. The general summary is that only about 2% overhead is added by the inter-tile protection mechanism to the overall runtime of the applications once the cache reaches a certain size. This size seems to always be twice the number of MPI “processes,” so in this case around 128 possible cache entries.

The assumption is that this is influenced by the MPI multicast implementation. This implementation distributes messages in a tree-like manner, so each MPI process has a maximum of two child processes which receive messages. Overall, the impact is not large as most of the work resides in the actual copying of data and starting source and destination iLets.

5.6 Conclusion

The evaluation was performed on multiple different architectures including the reference hardware architecture for Invasive Computing. It was shown that the memory protection, implemented in the adaptive system within OctoPOS is capable to lift and establish memory isolation at runtime.

The measurements, including the statistical analysis clearly shows great performance differences between the “on” and “off” modes, especially with respect to runtime complexity but also in absolute terms. Additionally, the difference between the statically removed possibility of dynamic memory protection and the “dynamic off” case is quite low. Speed increases range from 10x to 136x depending on the operation. This is of course the case because some operations have very low constant overhead in unprotected cases, whereas the runtime complexity is linear in the protected case. So, given enough memory or CPU cores these factors could be arbitrary large.

The operations to acquire and release resources, be it memory via mem_map/mem_unmap or invade/retreat for claims, are cheap in the unprotected case. This enables unprotected applications to be more resource aware and frequently release resources to the overall system, so that other applications that need these resources can benefit, which may lead globally to a more performant system.

Finally, the results show low runtime jitter in the unprotected case caused by memory management. Applications with high demand on predictability may have lower estimates on the worst case execution time and therefore the schedulability for time-restricted applications increases. In the given ray-tracing example, this has the effect that noticeable delays are avoided.
Besides this, the choice of the red-black tree, central to many data structures within the adaptive address-space management system, not only increases performance of the individual operations but may also lower jitter compared to simpler list-based approaches.

In direct comparison to Linux’ `munmap` operation, the fully protected case is faster and shows similar runtime behaviour. This substantiates that the protected mode is not made arbitrarily slow to give a better comparison to the deactivated protection modes.
The conclusion of this thesis about adaptive address-space management for resource-aware applications follows. A summary is given for the work done and the results accomplished in the areas of memory protection, address-space management and synchronisation. Afterwards, ideas presented in this thesis and its implementation are delimited from related work. Finally, an outlook and ideas for possible future work is given at the end.

6.1 Adaptive Memory Protection

Memory protection is essential for safety and security of systems when dealing with unknown applications from untrustworthy sources. It is reasonable that state of the art general-purpose operating systems enable this protection by default. However, memory protection also adds substantial time overhead to certain memory-related operating-system services and may cause predictability issues for parallel applications. The costliest part of memory protection comes whenever a parallel application dynamically changes its memory mapping. Alongside the apparent modifications on the page tables themselves, TLB entries must be invalidated or the TLB must even be emptied completely. Not only this, but also the TLB entries of all other CPU cores belonging to that application at that time must be visited and invalidated.

However, these MMU-based protection mechanisms may be unnecessary under certain circumstances. Programs that are written in a type-safe language using a trustworthy compiler and run-time system can only access their own allocated objects and are unable to modify arbitrary memory locations. Memory protection does not provide significant benefits for security in these cases. Programs for the invasive architecture written in X10 represent this kind of type-safe programs.

A dynamic protection scheme was implemented in OctoPOS that allows to enable and disable protection for multi-core applications on the invasive architecture.
Chapter 6 Conclusion

This enables iRTSS to adapt to the current runtime situation and to provide just the right amount of protection where necessary, and to leave trustworthy applications unisolated.

The decision when to enable or disable memory protection for applications is left for the agent system at run time whenever isolation, performance or predictability demands arise.

To facilitate such an adaptive protection, memory areas belonging to an application are tracked. These regions include the static text and data sections as well as regions assigned by dynamic memory allocation on page level. With this region information at hand, turning on memory protection entails building a page-table hierarchy and applying these page tables on all CPUs currently belonging to this application. The inverse operation removes the page-table mapping from each application CPU core. Depending on the hardware architecture, the MMU can be disabled (Sparc Leon) or, as it is the case on x86-64, a simple mapping is activated that grants access to all user-space memory regions. Consequently, trustworthy multi-core applications that already have access to all local memory do not cause page-table updates, TLB invalidations or TLB shootdowns.

Finally, the approach of this thesis follows the design principle that less demanding applications should not pay for the resources consumed by unneeded features [39] also with respect to memory protection.

Benchmarks were carried out for important system services including invade, retreat and claim construction for protected and unprotected applications, as well as a configuration of iRTSS that does not support memory protection at all and does not track application-memory ranges and serves as a baseline for the costs of memory protection.

Timings increase linearly with the amount of memory and cores involved in protected mode. Whereas in unprotected mode the same operations take constant time with respect to both of these parameters. Speed increases from protected to unprotected modes range from 10x to 136x depending on the operation. Because of the differences in computational complexity, these factors can be arbitrary large given enough cores or memory.

However, it is not just the run time that can be lowered for unprotected applications, but also predictability improved, since the runtime of an invade, retreat or reinvade does not strongly depend on the number of cores that the application currently utilises. Furthermore, application cores that are currently in computation are not interrupted to do expensive TLB invalidations when some other application part on a different core changes the shared address space.
X10 application benchmarks show performance benefits in unprotected cases and as user-space programs get more resource aware with respect to memory, even a greater difference might be observable. Currently, performance improvements in the range of 2% to 20% can be observed in unprotected modes for X10 applications.

### 6.1.1 Cross-Tile Protection

Applications can span multiple tiles and OctoPOS’ communication mechanisms were adjusted to cope with memory-protection rules.

Hardware platforms often provide asynchronous DMA units for communication between tiles. The performance of these units is leveraged to enable asynchronous user-space communication in a secure and flexible way, even in the face of variable protection states of the same application on different tiles. This unit does not obey any memory-protection rules as defined in the page tables by itself. For this reason, memory protection rules were enforced in software by OctoPOS when using the push-DMA functionality.

The implementation consists of a distributed software-cache mechanism to cache and invalidate access rights for remote memory regions. DMA transfers to recurring remote regions just cause a lookup in the local cache without the overhead of doing expensive checks on the remote tile.

### 6.1.2 Solved Problems Related to Address-Space Management

In reference to the initial motivation in Section 1.1, all problems could be solved. The struggle with physical page allocation on NUMA machines was solved by a constructive approach. The tiled invasive architecture was mapped onto x86-64 NUMA systems, so that each tile is represented by a NUMA domain. Therefore, employing the invasive mechanisms of exclusive resources, inter-tile communication and remote execution of i Lets. In this way the layout of physical memory is obvious to applications and accessing remote locations is explicit. In contrast, state of the art systems use automatic distribution strategies for physical pages that may lead to non-optimal outcomes for applications and introduce complexity in the OS.

Performance and predictability problems in the field of parallel applications was tackled by the main contribution of this thesis, which is adaptive memory protection.

Fragmentation and caching of large memory regions was solved by introducing a novel memory allocator with beneficial fragmentation behaviour but, ultimately, in allowing applications to be resource aware with respect to memory without sacrificing performance or predictability. This is the case for unprotected modes and is practised by the implemented user-space memory libraries and the invasive garbage collector for X10, that both release memory early and often.
6.1.3 Guarded Section Synchronisation

To synchronise dependent operations in the address-space management a novel concept and implementation named Guarded Sections was conceived. Guarded Sections care for the sequential execution of critical sections and simultaneously permit asynchronous progression after the critical sections. This enables the possibility for asynchronous progression of functionality unrelated to the address-space management in OctoPOS, while guaranteeing synchronisation where it is needed.

6.1.4 Multi Architecture Support

The concept of adaptive memory protection and the whole implementation part of this thesis was applied to two main hardware architectures. The reference architecture of the Sparc Leon with invasive additions like the iNoC and in-hardware tiles without cache coherency. As a prevalent architecture for high performance computing, iRTSS and especially OctoPOS with the extensions introduced by this thesis, was brought to x86-64 systems as a result of this thesis. These systems incorporate dozens to hundreds of CPU cores and hundreds of GiB of memory. Not only is this the first thesis in the context of OctoPOS to achieve support, and evaluate results, on both architectures, it also laid the foundation for other theses and papers and allows further research on these systems in and around the Collaborative Research Centre.

6.2 Related Work

Related work in the field of memory protection and associated costs is presented here and differences to this thesis are emphasised. No other paper or thesis presents an operating system or part of it that is able to dynamically switch protection modes of applications at runtime.

Related material can be grouped into three categories:

- Approaches that analyse the benefit of lifting memory protection under certain assumptions and situations.
- Changing memory isolation with respect to a subset of other applications or parts of multiple address spaces.
- Keeping memory protection but lowering the impact of remote TLB invalidations.

6.2.1 Lifting Memory Protection

In [40] Golm et al. developed a Java-based operating system named JX in which Java-only applications are not isolated by hardware means but by the fact that these programs are type safe and thus the compiler and runtime system can guarantee the isolation between applications. The system is furthermore grouped into domains that employ their own memory and thread management. A special DomainZero contains native and type-unsafe operating-system and low-level code.
6.2 Related Work

JX does not use a MMU at all and relies on the type safeness of Java and its implementation. Especially DomainZero must be in the minimal trusted computing base. In contrast to the work of this thesis, JX is not able to establish memory protection, neither through static nor through dynamic configuration, let alone execute native applications.

A step further [41] Aiken et al. present in their paper “Deconstructing process isolation” an approach to have hardware-isolated and software-isolated processes in one system. Their operating system Singularity runs on commodity x86 hardware and notable performance differences in favour of software-isolated processes were detected. However, a distinctive difference to this thesis is the static configuration of which processes run isolated and which do not. The problem of dynamic switching between these states was not addressed. Furthermore, Singularity never ran on a multicore platform, the performance and predictability impact and problems of TLB synchronisation were not tackled and never measured.

Finally, based on Lohmann’s work [42], Stilkerich et al. [43], [44] develop a configurable system with optional memory protection for individual components of the system. These may either be Java based or native application parts and protection may be MPU based, software based or completely disabled. The operating system CiAO can be configured via Aspect-C++ and other static configuration means to guarantee the required protection level. CiAO is a family of AUTOSAR like operating systems in the field of embedded and single-core real-time systems. Similarly to Aiken’s work, the decision which applications are isolated is configured statically ahead of run time and cannot be changed dynamically. Again, the problems of TLB synchronisation do not arise here.

6.2.2 Dynamic Application Isolation

In his dissertation Parmer [45] presents a component-based operating system where certain user-space components serve some operating system functionalities. These components are isolated between one another but may communicate via messages. To increase communication performance, selected components can be merged into one address space and form one shared-memory region. This seems to be possible at run time. When multiple applications reside in one protection domain their relative isolation is voided. In some sense this is an extreme form of traditional shared memory that can be established between processes with conventional operating systems like Linux. However, these applications are still within one protection domain. All overheads of TLB invalidation and page-table manipulation still apply. Disadvantages regarding memory-related operations and the implications regarding performance and predictability still remain for each of the “merged” applications.
6.2.3 Lowering TLB Invalidation Costs

One of the driving factors of this thesis are the costs associated with TLB management, especially multi-core TLB consistency. Other research groups focused on keeping memory protection but reducing TLB invalidation costs. Solutions that are viable with current hardware include:

Lazy TLB consistency, Chang and Koh [46], that couples TLB consistency with memory consistency. Applications have to issue special release and acquire operations before accessing pages that are used by multiple concurrent threads of execution. Only simulations were performed but showed lower overheads for TLB invalidation.

Lazy translation coherence, Kumar et al. [47], gives up the synchronous nature of unmapping a page. As a consequence only local invalidations happen plus some bookkeeping of pages that need remote invalidation. At scheduler events on those remote cores these invalidations again happen locally.

In [48] Amit shows a strategy to reduce the number of invalidations needed by utilising the page-access bit, present in many MMU implementations.

Hardware-assisted solutions include: A shared TLB directory cache presented by [49] that stores and synchronises a system-wide TLB state and liberates the OS from TLB synchronisation. A hardware extension by Awad et al. [50] to have an expiration time with TLB entries. If the expiration time is in past then no remote TLB invalidation is needed. This also just reduces the number of TLB invalidations and does not prevent them entirely.

In fact all of these approaches do not prevent some TLB invalidation to happen either in hardware or at reduced occurrences and ramifications in software. For this reason it can be argued that disabling memory protection and entirely circumventing TLB synchronisation is not only a completely different approach in this thesis, but may also be more efficient for applications that need no hardware-based isolation. Furthermore, the cost of maintaining and synchronising the page-table data structures was not tackled at all by these papers and is included in the work of this thesis.
6.3 Outlook

This chapter finishes with ideas on future work that could not be tackled during the work of this thesis, at last “All we have to decide is what to do with the time that is given us.” [51]. Some of these ideas may have already been picked up for other theses or papers.

6.3.1 Memory Virtualisation

A natural step forward is to extend the existing memory protection system towards full memory virtualisation. Always with the possibility to fall back to simpler kinds of address-space management, like a flat but protected address-space and even disabled memory protection. With such a system in place applications would only pay the price of main-memory overcommitment, swapping, related data structures and synchronisation when it is really needed and some scheduler or iRTSS agent system decides that the constraints of the applications in question allow for these kinds of overheads and unpredictabilities.

6.3.2 Dealing With an Abundance of Memory Types

The introduction of many different types of memory in invasive systems may make it hard to efficiently exploit all possibilities of the system. At the time of writing, the reference hardware platform for InvasIC based on Sparc Leon incorporates SHM, TLM, remote TLM, iCore TLM, TCPA memory and the possibility to enable a core-local scratchpad memory. These types represent a network of memory with different properties. The contents of a memory page may be moved freely and transparently within this network as long as the requirements of the application are known, e.g. regarding cache coherency or accelerator accessibility. So instead of letting the application allocate a specific kind of memory and let it copy or handle problems itself, the application may specify its functional or non-functional requirements on a particular amount of memory. Combined with a light version of memory virtualisation, this would enable iRTSS to transparently “swap” out memory pages from fast local memory regions into slower regions and back again, as long as the requirements allow for this regarding cache coherency but also regarding predicability. Therefore, fast but small storage types could be transparently over-provisioned or, in turn, partially reserved. All depending on the requirements of applications.

6.3.3 Distributed Shared Memory

A distributed shared memory system on the invasive architecture could benefit from custom invasive hardware components that may accelerate the operation of such a system. The basic functionality for MMU manipulation and page-fault handling were implemented for this thesis and can serve for such undertakings. Additionally, in combination with adaptive memory protection within and between tiles, trusted applications could benefit from increased performance or predicability.
A.1 Source Code

A.1.1 Tree Block Allocator

```c
struct EmbeddedFreeBlock
{
    struct LinkNode
    {
        EmbeddedFreeBlock *prev;
        EmbeddedFreeBlock *next;
        uintptr_t parent;
    };

    lib::adt::RBNode<EmbeddedFreeBlock> addrNode;

    union {
        lib::adt::RBNode<EmbeddedFreeBlock> sizeNode;
        LinkNode linkNode;
    };

    EmbeddedFreeBlock *headNext;
    uintptr_t size;

    uintptr_t getStartAddress()
    {
        return (uintptr_t)this;
    }

    static EmbeddedFreeBlock* create(uintptr_t start, uintptr_t blockSize)
    {
        EmbeddedFreeBlock *newBlock = (EmbeddedFreeBlock*)start;
        newBlock->size = blockSize;
        newBlock->headNext = nullptr;
        return newBlock;
    }

    static void destroy(EmbeddedFreeBlock *block)
    {
        (void)block;
    }
};
```
```cpp
static EmbeddedFreeBlock* recycle(EmbeddedFreeBlock* block,
  uintptr_t start, uintptr_t blockSize)
{
  (void)block;
  return create(start, blockSize);
}

template< uintptr_t BLOCK_BITS, typename FreeBlock, typename Locker>
class TreeBlockAllocatorGeneric {
private:
  template<bool COMPARE_SIZE>
  struct Comparator {
    static int cmp(uintptr_t val, FreeBlock* block) {
      uintptr_t blockVal;
      if(COMPARE_SIZE) {
        blockVal = block->size;
      } else {
        blockVal = block->getStartAddress();
      }
      if(val < blockVal) return -1;
      if(val > blockVal) return 1;
      return 0;
    }
    static int cmp(FreeBlock* a, FreeBlock* b) {
      if(COMPARE_SIZE) {
        return cmp(a->size, b);
      } else {
        return cmp(a->getStartAddress(), b);
      }
    }
  };
  Locker locker;
  lib::adt::RBTree<FreeBlock, &FreeBlock::addrNode, uintptr_t, Comparator<false>> addrTree;
  lib::adt::RBTree<FreeBlock, &FreeBlock::sizeNode, uintptr_t, Comparator<true>> sizeTree;
public:
  uintptr_t getBlockSize() const {
    return ((uintptr_t)1) << BLOCK_BITS;
  }
private:
  static uintptr_t alignUp(uintptr_t numToRound, uintptr_t multiple) {
    uintptr_t mask = multiple - 1;
    return (numToRound + mask) & ~mask;
  }
};
```
A.1 Source Code

```c
void linkBlock(FreeBlock *oldBlock, FreeBlock *newBlock)
{
    FreeBlock *ring = oldBlock->headNext;
    newBlock->headNext = oldBlock;
    sizeTree.replace(oldBlock, newBlock);
    if(ring == nullptr) {
        oldBlock->headNext = ((FreeBlock*)((uintptr_t)newBlock) | 1);
        oldBlock->linkNode.prev = oldBlock;
        oldBlock->linkNode.next = oldBlock;
    } else {
        oldBlock->headNext = ((FreeBlock*)((uintptr_t)newBlock) | 1);
        oldBlock->linkNode.prev = ring;
        oldBlock->linkNode.next = ring->linkNode.next;
        ring->linkNode.next->linkNode.prev = oldBlock;
        ring->linkNode.next = oldBlock;
    }
}

void unLinkBlock(FreeBlock *oldBlock)
{
    const bool inRing = (((uintptr_t)(oldBlock->headNext)) & 1) == 1;
    FreeBlock *head = ((FreeBlock*)((uintptr_t)(oldBlock->headNext)) & (~1ul));
    if(inRing) {
        if(head == nullptr) {
            oldBlock->linkNode.next->linkNode.prev = oldBlock;
            oldBlock->linkNode.prev = oldBlock;
        } else {
            oldBlock->prev = oldBlock->linkNode.prev;
            if(prev == oldBlock) {
                head->headNext = nullptr;
            } else {
                prev->headNext = oldBlock->headNext;
                head->headNext = prev;
                oldBlock->linkNode.next->linkNode.prev = oldBlock;
                oldBlock->linkNode.prev = oldBlock->linkNode.next;
            }
        }
    } else {
        FreeBlock *prev = oldBlock->linkNode.prev;
        if(prev == ring) {
            ring->headNext = nullptr;
        } else {
            prev->headNext = (FreeBlock*)((uintptr_t)ring) | 1);
            ring->headNext = prev;
            ring->linkNode.next->linkNode.prev = ring->linkNode.prev;
            ring->linkNode.next = ring->linkNode.next;
        }
    }
}
```
void removeFromSizeTree(FreeBlock *block) {
    if(block->headNext != nullptr) {
        unLinkBlock(block);
        block->headNext = nullptr;
    } else {
        sizeTree.remove(block);
    }
}

void remove(FreeBlock *block) {
    addrTree.remove(block);
    removeFromSizeTree(block);
}

void addToSizeTree(FreeBlock *block) {
    FreeBlock *oldBlock = sizeTree.insert(block);
    if(oldBlock != block) {
        linkBlock(oldBlock, block);
    }
}

void doAlignmentSplit(FreeBlock *outBlock, uintptr_t alignment, 
                      uintptr_t allocSize) {
    uintptr_t startAddr = outBlock->getStartAddress();
    uintptr_t blockSize = outBlock->size;
    uintptr_t blockEnd = startAddr + blockSize;
    uintptr_t alignedChunk = alignUp(startAddr, alignment);
    uintptr_t alignedChunk = alignUp(startAddr, alignment);
    uintptr_t leadingSize = alignedChunk - startAddr;
    uintptr_t leadingBlocks = leadingSize >> BLOCK_BITS;
    uintptr_t trailingBlocksStart = alignedChunk + allocSize;
    uintptr_t trailingSize = blockEnd - trailingBlocksStart;
}
A.1 Source Code

```c
uintptr_t trailingBlocks = trailingSize >> BLOCK_BITS;

// if there are some leading blocks then the start address does not
// change, so do not remove the old free block from the address tree
if(leadingBlocks != 0) {
  // remove this block just from the size tree, start address will
  // not change
  removeFromSizeTree(outBlock);

  // set the new size of this block
  outBlock->size = leadingSize;

  // add this block back to the size tree
 addToSizeTree(outBlock);

  // there are some trailing blocks and we kept the leading block
  // -> create a new free block and add it
  if(trailingBlocks != 0) {
    FreeBlock *newBlock = FreeBlock::create(trailingBlocksStart, trailingSize);
    add(newBlock);
  }
  else {
    // there are no leading blocks
    if(trailingBlocks != 0) {
      // see alloc() for the strategy that is used here
      removeFromSizeTree(outBlock);

      FreeBlock *newBlock = FreeBlock::create(trailingBlocksStart, trailingSize);
      addrTree.replace(outBlock, newBlock);

      addToSizeTree(newBlock);
    }
  }
}
else {

  // there are also no trailing blocks, the block is removed
  // completely
  remove(outBlock);
}
FreeBlock::destroy(outBlock);

return (void*)alignedChunk;
```
// look for a FreeBlock >= 'size' in the size tree
FreeBlock *outBlock = sizeTree.ceil(size);
if(outBlock != nullptr) {
    // start of this free block
    const uintptr_t startAddr = outBlock->getStartAddress();
    // size of this block in bytes
    const uintptr_t blockSize = outBlock->size;
    // end of this free block, not inclusive
    const uintptr_t blockEnd = startAddr + blockSize;
    // a continuous block is completely removed
    // remove the old block from both trees
    remove(outBlock);
    }
else {
    // there are trailing blocks, we can add the trailing block to
    // the size tree and replace the node in the address tree
    // without removing and readding it into the same position in
    // the addrTree
    removeFromSizeTree(outBlock);
    FreeBlock *newBlock = FreeBlock::create(trailingBlocksStart, trailingSize);
    addrTree.replace(outBlock, newBlock);
    addToSizeTree(newBlock);
    }
FreeBlock::destroy(outBlock);
out = (void*)startAddr;
}
locker unlock(&item);
return out;
}
void* allocAligned(uintptr_t alignment, uintptr_t blocks)
{
    // if not power of two
    if((alignment == 0) || ((alignment & (alignment - 1)) != 0)) {
        return nullptr;
    }
    if(blocks == 0) {
        return nullptr;
    }
    if(alignment <= getBlockSize()) {
        return alloc(blocks);
    }
    // at this point alignment is a larger power of two than the
    // block size
// the size to allocate
uintptr_t allocSize = blocks << BLOCK_BITS;

// allocate (blocks + alignment - 1) blocks
uintptr_t extraBlocks = (alignment >> BLOCK_BITS) - 1;

// look for this size to allocate
uintptr_t size = (blocks + extraBlocks) << BLOCK_BITS;

// get the lock
typename Locker::Item item;
locker.lock(&item);

// look for a FreeBlock >= 'size' in the size tree
FreeBlock *outBlock = sizeTree.ceil(size);
if(outBlock == nullptr) {
    // there is no such free block
    // try allocating exactly the desired size - maybe the resulting
    // chunk happens to have proper alignment
    outBlock = sizeTree.ceil(allocSize);
    if(outBlock == nullptr || ((outBlock->getAsStartAddress() % alignment) != 0)) {
        // also not successful
        locker.unlock(&item);
        return nullptr;
    }
}

// we found a free block large enough
void *out = doAlignmentSplit(outBlock, alignment, allocSize);
locker.unlock(&item);
return out;

bool free(void *s, uintptr_t blocks)
{
    uintptr_t start = (uintptr_t)s;
    uintptr_t size = blocks << BLOCK_BITS;
    const uintptr_t end = start + size;

typename Locker::Item item;
locker.lock(&item);

    // try to merge an already existing free block at the end of the
    // memory to be freed, search for its successor
FreeBlock *pred;
FreeBlock *succl = addrTree.search(end);
if(succ != nullptr) {
    // if there is a successor, grab its predecessor, this is an
    // optimisation as BTree::prev() is (in average) faster than
    // BTree::floor()
    pred = addrTree.prev(succ);
}
else {
    // if there is no successor, search for the predecessor
    pred = addrTree.floor(start);
}
```cpp
if(pred != nullptr) {
    const uintptr_t floorStart = pred->getStartAddress();
    const uintptr_t floorEndAddr = floorStart + pred->size;
    if(floorEndAddr != start) {
        pred = nullptr;
    }
}

if(pred != nullptr && succ != nullptr) {
    // append the current block and the successor to the predecessor,
    // start address does not change
    removeFromSizeTree(pred);
    remove(succ);
    pred->size += size + succ->size;
    addToSizeTree(pred);
} else if(pred != nullptr) {
    // append the current block to the predecessor, start address does not change
    removeFromSizeTree(pred);
    pred->size += size;
    addToSizeTree(pred);
} else if(succ != nullptr) {
    // replace successor in the addrTree without removing and re-adding
    removeFromSizeTree(succ);
    FreeBlock* newBlock = FreeBlock::create(start, size + succ->size);
    addrTree.replace(succ, newBlock);
    addToSizeTree(newBlock);
} else {
    // the blocks to be freed cannot be merged, create a new block
    FreeBlock* newBlock = FreeBlock::create(start, size);
    add(newBlock);
}

locker.unlock(&item);
return true;
}

bool grow(void* s, uintptr_t oldBlocks, uintptr_t newBlocks)
{
    uintptr_t start = (uintptr_t)s;
    bool resizeDone = false;
    const uintptr_t oldSize = oldBlocks << BLOCK_BITS;
    const uintptr_t addBlocks = newBlocks - oldBlocks;
    const uintptr_t additionalSpace = addBlocks << BLOCK_BITS;
    const uintptr_t end = start + oldSize;
    typename Locker::Item item;
    locker.lock(&item);
```

FreeBlock *extBlock = addrTree.search(end);

if(extBlock != nullptr) {
    if(extBlock->size >= additionalSpace) {
        const uintptr_t diff = extBlock->size - additionalSpace;
        if(diff > 0) {
            removeFromSizeTree(extBlock);
            FreeBlock *newBlock = FreeBlock::create(end + additionalSpace, diff);
            addrTree.replace(extBlock, newBlock);
            addToSizeTree(newBlock);
        } else {
            // a continuous block is completely removed
            remove(extBlock);
            FreeBlock::destroy(extBlock);
        }
        resizeDone = true;
    } else {
        // a continuous block is completely removed
        remove(extBlock);
    }
}
locker.unlock(&item);
return resizeDone;

void* allocLargest(uintptr_t minAlign, uintptr_t *minBlocks) {
    void *out = nullptr;
    const uintptr_t minSize = *minBlocks << BLOCK_BITS;
    typename Locker::Item item;
    locker.lock(&item);
    FreeBlock *block = sizeTree.max();
    if(block != nullptr) {
        const uintptr_t start = block->getStartAddress();
        const uintptr_t blockSize = block->size;
        if(blockSize >= minSize) {
            const uintptr_t alignStart = alignUp(start, minAlign);
            const uintptr_t alignWaste = alignStart - start;
            if(blockSize >= alignWaste) {
                const uintptr_t remainSize = (blockSize - alignWaste) & ~((minAlign - 1));
                if(remainSize >= minSize) {
                    const uintptr_t remainBlocks = remainSize >> BLOCK_BITS;
                    out = doAlignmentSplit(block, minAlign, remainSize);
                    *minBlocks = remainBlocks;
                }
            }
        }
    }
    locker.unlock(&item);
    return out;
};
A.1.2 Guarded Sections

```c
struct Order {
    Order *next;
    void (*func)(void);
    void *args;
};

struct Guard {
    Order *head;
    Order *tail;
    Order dummy;
    uintptr_t lock;
};

void GuardDefer(Guard *guard, Order *order) {
    enqueue(guard, order);
    for(;;) {
        if(atomic_swap(&guard->lock, 1) != 0) {
            break;
        } else {
            for(;;) {
                order = dequeue(guard);
                if(order == nullptr) {
                    break;
                } else {
                    order->func(order->args);
                }
            }
            atomic_swap(&guard->lock, 0);
            if(empty(guard)) {
                break;
            }
        }
    }
}

void enqueue(Guard *guard, Order *order) {
    order->next = 0;
    Order *prev = atomic_swap(&guard->tail, order);
    atomic_write(&prev->next, order);
}

bool empty(Guard *guard) {
    return guard->head->next == 0;
}
```
A.1 Source Code

```c
1 Order* dequeue(Guard *guard) {
2     Order *item = guard->head;
3     Order *next = guard->head->next;
4     if(next == nullptr) {
5         return nullptr;
6     }
7     guard->head = next;
8     if(item == &guard->dummy) {  // Enqueue
9         enqueue(guard, item);
10        if(guard->head->next == 0) {
11            return 0;
12        }
13        guard->head = guard->head->next;
14     } return item;
15 }

A.1.3 MPI Barrier

```c

```c
1 static void signal_down_func(void *r, void *i) {
2     uintptr_t rawMPIComm = reinterpret_cast<uintptr_t>(r);
3     uintptr_t id = reinterpret_cast<uintptr_t>(i);
4     MPI_Comm mpiComm = (MPI_Comm)rawMPIComm;
5     Comm *com = TileGlobal::getComm(mpiComm);
6     CommBarrier *commBarrier = &com->barrierSync[id];
7     binary_signal_signal_and_exit(&commBarrier->topDownSignal);
8 }

```c

```c
1 static void signalDown(MPI_Comm mpiComm, Comm *com, const int id) {
2     dispatch_claim_t *claim = reinterpret_cast<dispatch_claim_t *>(com->getClaim(id));
3     simple_ilet ilet;
4     dual_ilet_init(&ilet, signal_down_func,  // Reinterpret casts
5                    reinterpret_cast<void*>(mpiComm),
6                    reinterpret_cast<void*>(id));
7     dispatch_claim_infect_single(*claim, &ilet);
8 }

```c

```c
1 static void signal_up_func(void *r, void *i) {
2     uintptr_t rawMPIComm = reinterpret_cast<uintptr_t>(r);
3     uintptr_t id = reinterpret_cast<uintptr_t>(i);
4     MPI_Comm mpiComm = (MPI_Comm)rawMPIComm;
5     Comm *com = TileGlobal::getComm(mpiComm);
6     CommBarrier *commBarrier = &com->barrierSync[id];
7     countAndInfect(mpiComm, com, commBarrier, 1, id);
8 }
```
Appendix A

static void signalUp(MPI_Comm mpiComm, Comm *com, const int id)
{
    dispatch_claim_t *claim = reinterpret_cast<
        dispatch_claim_t *>(com->getClaim(id));
    simple_ilet ilet;
    dual_ilet_init(&ilet, signal_up_func,
        reinterpret_cast<void*>(mpiComm),
        reinterpret_cast<void*>(id));
    dispatch_claim_infect_single(*claim, &ilet);
}

static void countAndInfect(MPI_Comm mpiComm, Comm *com, CommBarrier *commBarrier, const int val, const int id)
{
    const uintptr_t oldCounter = fetch_and_sub(
        &commBarrier->bottomUpCounter, val);
    const uintptr_t myCounter = oldCounter - val;
    if(myCounter == 0) {
        store(&commBarrier->bottomUpCounter, 3);
        if(id == 0) {
            binary_signal_signal(&commBarrier->topDownSignal);
        } else {
            const int parent = (id - 1) / 2;
            signalUp(mpiComm, com, parent);
        }
    }
}

int MPI_Barrier(MPI_Comm communicator)
{
    Comm *com = TileGlobal::getComm(communicator);
    const int size = com->getSize();
    if(size <= 1) {
        return MPI_SUCCESS;
    }
    const int id = com->getID();
    CommBarrier *commBarrier = &(com->barrierSync[id]);
    const int chld[2] = {id * 2 + 1, id * 2 + 2};
    int validChildren = 0;
    if(chld[0] < size) {
        validChildren ++ 1;
    }
    if(chld[1] < size) {
        validChildren ++ 1;
    }
    countAndInfect(communicator, com, commBarrier, 3 - validChildren, id);
    binary_signal_wait(&commBarrier->topDownSignal);
    binary_signal_init(&commBarrier->topDownSignal);
    if(chld[0] < size) {
        signalDown(communicator, com, chld[0]);
    }
}
A.2 Evaluation Data

A.2.1 Microbenchmarks for the Intel8 Variant

Figure A.1: Intel8 measurements detail how large the gap between static-off and dynamic off variants may be, even though both employ linear growth.
Figure A.2: Invasive retreat operation, measured time for Intel8.

Figure A.3: Unprotect operation for Intel8. The horizontal axis on the left shows increasing number of cores plotted for a selection of active pages for the benchmark application. The right side plots pages with graphs for different number of cores.
A.2 Evaluation Data

Figure A.4: Protect operation for Leon and Intel80. The horizontal axis on the left shows increasing number of cores plotted for a selection of active pages for the benchmark application. The right side plots pages with graphs for different number of cores.

Figure A.5: Runtime of memunmap on Intel8 for a dynamically protected system.
Figure A.6: Runtime of memunmap on Intel8 for a dynamically unprotected system.

Figure A.7: Runtime of memunmap on Intel8 for a statically unprotected system.
Figure A.8: Runtime of munmap for Linux on Intel80.
A.2.3 Red-Black Tree vs Sorted List

Figure A.9: Search and removal time on Leon for a single item from the data structure with decreasing size.
A.2 Evaluation Data

Figure A.10: Search and removal time on Intel80 for a single item from the data structure with decreasing size.

Figure A.11: Insertion time on Intel80 for a single item into the data structure with increasing size.
Appendix A Appendix

Figure A.12: Search and removal time on Intel8 for a single item from the data structure with decreasing size.

A.2.4 X10 Benchmarks for Intel8
Figure A.13: X10 application benchmarks on Intel8.
A.3 Embedded Resources

The PDF version of this document contains an embedded GZIP archive of the sources that are needed to build this document, as well as all evaluation data. This archive is encoded as plain-text base64, as some programs deny opening a PDF with an embedded binary archive for historical reasons.

Following software (in addition to base64, gzip and tar) is needed to recreate the PDF from the sources:

- Latin Modern Fonts
- Pandoc with pandoc-crossref and pandoc-citeproc
- A LaTeX installation compatible with pandoc
- GNU Make to use the Makefile

Figures were created using Libreoffice and drawio.
Bibliography


Appendix A Appendix


A.3 Embedded Resources


Appendix A


