Software-based Memory-Consistency Mechanisms
for Non-Coherent Many-Core Systems

Softwarebasierte Speicherkonsistenzmechanismen für
Nichtkohärente Many-Core Systeme

Der Technischen Fakultät der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

zur Erlangung des
Doktorgrades Dr.-Ing.

vorgelegt von
Tobias Langer
Als Dissertation genehmigt von
der Technischen Fakultät
der Friedrich-Alexander-Universität
Erlangen-Nürnberg

Tag der mündlichen Prüfung: 07.03.2024
Gutachter:
Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Prof. Dr.-Ing. Jörg Nolte
To Matts & Rebecca
Abstract

Multi- and many-core processors have become virtually indispensable. Some of today’s manycore processors already have cores counts in the higher two digits numbers. As the performance hunger of modern applications further increases, this trend towards more parallel processors will likely continue. However, this development requires significant changes in computer architectures. As a consequence, modern processor designs already deviate considerably from that of classical SMP systems, as components such as bus systems do not scale with the core count, and memory connects increasingly become a bottleneck. One effect of these changes is that cache coherence mechanisms become increasingly complex. These systems already cause high engineering efforts, as they require long design times and are complicated and time intensive to verify. Furthermore, they become increasingly resource intensive as their chip area and energy consumption grow with parallelism.

In the meantime, a different trend can be observed for system software. Cache coherence mechanisms have here been identified as a source of performance overheads. Therefore, many modern operating and runtime systems thus employ strategies to limit memory accesses to NUMA domains so that only the coherence mechanisms there are used. Cross-NUMA coherence, in return, is avoided, if possible. These software systems thus effectively avoid the coherent shared memory that the hardware level provides at great expense.

However, shared memory programming still provides a more convenient programming model than explicit communication schemes, such as message passing. Considering this situation, this dissertation proposes that coherence mechanisms should be realized as part of the software layer instead of providing them in hardware. The coherence mechanism can thus adapt to the applications’ requirements, making it far more flexible than hardware implementations can be. Therefore, this dissertation explores the design space for software-based coherence solutions. The proposed designs thereby build on techniques known from Virtual Shared Memory (VSM) systems to realize a memory consistency mechanism as an on-demand feature. This notion of memory consistency also meets the requirement for cache coherence.

The results and contribution of this thesis are two variants of software-based consistency mechanisms. The first of which is the Software Consistency System (SCS), which follows the principles of classical VSM systems by leveraging the features of virtual memory management to create a consistent shared memory abstraction. The second variant is the Adaptive Software Consistency System (ASCS), a novel adaptive consistency mechanism. The ASCS builds upon the SCS and extends it with the ability to exploit memory-sharing information to switch adaptively between page and hardware caches at runtime.

Finally, this thesis demonstrates the functionality and effectiveness of the consistency systems with a performance evaluation on a tile-based prototype system implemented in FPGA logic.
at the example of different shared memory workloads. The results show that the SCS and the ASCS provide good scalability for the parallel workloads and, thus, that they are feasible alternatives to existing hardware-based coherence mechanisms. Furthermore, the experiments show that for all but one evaluation scenario, the ASCS mechanism outperforms the SCS.
Kurzzusammenfassung


Das Ergebnis dieser Arbeit sind zwei Arten softwarebasierter Konsistenzmechanismen. Die erste Variante der Mechanismen ist das Software Consistency System (SCS), welches den Prinzipien klassischer VSM-Systeme folgt. Das SCS nutzt Funktionen der virtuellen Speicherverwaltung, um eine konsistente geteilte Speicherabstraktion zu errichten. Die zweite Variante ist

Acknowledgements

First, I'd like to thank my adviser, Wolfgang Schröder-Preikschat. Thank you for allowing me to pursue my research in your research group. With the systems-software group, you've created a fruitful research environment. To this day, I'm impressed that I learned something new from every one of our discussions.

Furthermore, I'd like to thank Jörg Nolte for taking the time to review my thesis. Many thanks to the fantastic people at the systems-software group with whom I had the chance to work. I'm particularly grateful for Peter Wägemann and Tobias Distler's support when preparing my defense. Additional thanks go out to my roommates. Having you around to discuss and vent about technical, philosophical, and political topics made the time so much more enjoyable. Furthermore, I want to mention the members of the OctoPOS team: Sebastian Maier, Gabor Drescher, Christoph Erhardt, Florian Schmaus, Jonas Rabenstein, Benedikt Herzog, and Phillip Raffeck. I enjoyed working with you guys, and even if there were more than a handful of unnerving bug hunts, I don't want to miss this experience!

Furthermore, I'd like to shout out to all the fantastic people with whom I collaborated throughout the InvasIC. I learned a lot during this time, including compiler engineering and hardware design. If I had to choose a few of you to name, then I would like to express particular thanks to Marcel Brand, Christian Heidorn, Oliver Lenke, Sven Rheindt, Fabian Lesniak, Andreas Fried, Marcel Mettler, Franziska Schirrmacher, and Akshay Srivatsa. You guys were always approachable, patiently answered questions, and helped when the prototyping system had hiccups.

Over the years, I have had the opportunity to work with many bright students, some of whom I later could call colleagues and friends. Maximilian Ott and Eva Dengler especially come to mind. Thank you for the insightful discussions we had. I hope I was able to teach you only half as much as I learned from the time we spent together.

Finally, credit is due to my parents, Gudrun and Karl-Heinz. Your continuous support has brought me to where I am today: over these years, you've always encouraged me to pursue my interests. You are the people who initially sparked my interest in computer science and put me on track for the journey that culminated in this thesis. Many thanks go moreover to Birgit, Berthold, and Tamara; I'm glad that I can consider you a part of my family. Throughout my dissertation, you have provided me with so much more support than I would ever dare to ask for.

Last but certainly not least, a thousand thanks go to my partner, Rebecca. You've helped me navigate the ups and downs of these last few years and successfully kept me motivated. Especially close to the finishing line, when naturally, everything got a little busier, and Matts came into our small family, you had my back. Thank you for who you are. I love you.
## Contents

1 Introduction  
   1.1 Problem Statement ........................................... 3  
   1.2 Contributions of this Dissertation .......................... 5  
   1.3 Relevant Publications ....................................... 6  
   1.4 Thesis Structure ............................................ 6

2 Fundamentals  
   2.1 Principles of Invasive Computing ............................ 9  
   2.2 Cache Coherence and Memory Consistency .................... 16  
   2.3 Conclusions ................................................. 22

3 State of the Art  
   3.1 Parallel Programming Models .................................. 25  
   3.2 Coherence in Parallel Computer Architectures ................ 30

4 A Software Consistency System  
   4.1 A Software Approach for Coherence in Tiled Architectures .... 39  
   4.2 Design and Architecture of the Software Consistency System .... 45  
   4.3 Implementation of the Software Consistency System ............ 48  
   4.4 Résumé ..................................................... 66

5 An Adaptive Software Consistency System  
   5.1 An Adaptive Approach to Caching and Consistency ............ 70  
   5.2 An Adaptive Cache Design Incorporating Software and Hardware Caches . . . 75  
   5.3 Implementation of the Adaptive Software Consistency System .... 76  
   5.4 Résumé ..................................................... 82

6 Runtime System for Shared Memory Programming  
   6.1 Design of the Runtime System .................................. 83  
   6.2 Synchronization Primitives .................................... 84

7 Evaluation and Discussion  
   7.1 Preliminary Considerations .................................... 95  
   7.2 Experimental Setup .......................................... 96  
   7.3 Fundamental System Characteristics .......................... 97  
   7.4 Evaluation with Real World Applications ...................... 108  
   7.5 Conclusion ................................................. 121
List of Figures

2.1 State machine of the phases in the InvasIC execution model. 10
2.2 Conceptual overview of the tile-based InvasIC MPSoC architecture. 11
2.3 Memory layout of the InvasIC platform. 15
2.4 Examples of cache inconsistencies without cache coherence protocols. 17

4.1 Illustration of the SCS's functionality. 42
4.2 Illustration of the multiple-writer support mechanism. 43
4.3 Visualization of how the SCS ensures consistency. 45
4.4 Overview of the SCS components. 46
4.5 State chart of a memory page's live cycle. 47
4.6 Refined overview of the SCS's components. 49
4.7 Visualization of the page synchronization object. 52
4.8 Activity diagram for cache miss handling in the SCS. 55
4.9 Activity diagram for the writeback operation in the SCS. 58
4.10 Visualization of fast- and diff-based writebacks. 59
4.11 Activity diagram for page cache entry invalidation in the SCS. 61
4.12 Activity diagram for the acquire fences in the SCS. 63
4.13 Activity diagram for the release fences in the SCS. 65

5.1 State machine of the states of a memory page. 71
5.2 State machine of the exclusive caching strategy. 72
5.3 State machine of the read-shared caching strategy. 74
5.4 Overview of the ASCS components. 75
5.5 Activity diagram for cache miss handling in the ASCS. 77
5.6 Activity diagram for write fault handling in the ASCS. 79

6.1 Structure of the primitives in the shared memory programming runtime. 85
6.2 Visualization of the global barrier communication pattern. 89

7.1 Graph of median execution time for remote memory accesses. 106
7.2 Plots of the runtime for the FFT kernel. 111
7.3 Plots of the runtime for the LU (Contiguous Blocks) kernel. 112
7.4 Plots of the runtime for the LU (Non-Contiguous Blocks) kernel. 114
7.5 Plots of the runtime for the Radix kernel. 115
7.6 Plots of the runtime for the Ocean (Contiguous Partitions) application. 116
7.7 Plots of the runtime for the Water Spatial application. 117
List of Figures

7.8 Plots of the relative execution times of the software consistency mechanisms. . 118
A.1 Overview of ë-let scheduling. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128
List of Tables

7.1 Overview of the used architecture configuration. .......................... 96
7.2 Overview of the L2 and page cache configurations used for the evaluation. 97
7.3 Statistics for measurement overheads. ........................................ 98
7.4 Runtime statistics for entering and leaving page fault handlers. .... 98
7.5 Runtime statistics for entering and leaving memory fences. ........ 99
7.6 Runtime statistics for DMA operations. ................................. 100
7.7 Runtime statistics for serving cache hits. ................................. 100
7.8 Runtime statistics for serving compulsory misses. .................... 101
7.9 Runtime statistics for serving capacity misses. ......................... 102
7.10 Runtime statistics for writing back cache lines. ....................... 102
7.11 Runtime statistics for invalidating cache lines. ....................... 103
7.12 Runtime statistics for serving cache misses under exclusive caching. 104
7.13 Runtime statistics for serving cache misses under adaptive-read-shared caching. 105
7.14 Used SPLASH-3 kernels and applications for benchmarking. .... 109
## List of Listings

1. Example of threads in a producer-consumer relationship. .................. 19
2. Example of an application that leads to unexpected results under TSO. .. 20
3. Synchronization example of threads in a producer-consumer relationship. . 21
4. Syscall interface to create replicas of remote pages. ......................... 56
5. Syscall interface to make page cache entries read-write. .................... 57
6. Syscall interface to write back page cache entries. .......................... 59
7. Syscall interface to invalidate page cache entries. ......................... 61
8. Syscall interface to flush page cache entries. ............................... 62
9. Syscall interface of the acquire memory fence. .............................. 64
10. Syscall interface of the release memory fence. ............................. 64
11. API of the family of barrier functions. .............................. 88
12. API of the family of lock functions. ................................. 90
13. API of the family of condition variable functions. ..................... 92
14. API of the family of event functions. ................................. 93
15. Micro-benchmark kernel to establish timestamp overheads. ................. 98
1 Introduction

Gordon Moore postulated in 1965 that the number of transistors in integrated circuits would double every 12–18 months [Moo65]. Today, this prediction is better known as “Moore’s Law”¹—and albeit not being a physical law, it has since proven somewhat accurate. Taken together with Dennard Scaling [Den+74], the observation that the power density of integrated circuits stays constant with transistor size, Moore’s law had promised quasi-free performance gains.

This apparent free performance increase has put software developers in a convenient situation. For years to come, every hardware release cycle would bring faster Central Processing Unit (CPU)s that would fulfill their increasing performance demands. If a problem set had hit the capabilities of the current computer system, undoubtedly, the next processor generation would easily solve it.

However, this comfortable situation was only short-lived: first, the so-called “memory wall” [WM95] changed the organization of memory and processors in computer architectures. Earlier computer systems used a common system clock for all hardware components. However, this design quickly turned out to be far too limiting in terms of performance: by the 1990s, processor frequencies had reached multiples of those of memory, rendering the memory the system bottleneck. In order to prevent long stalling times when fetching data, deep cache hierarchies became commonplace in computers.

Secondly, Dennard Scaling appeared to break down. Although it was expected for Dennard Scaling to hit its limits someday, by the mid-2000s, it became clear that higher integration densities would no longer equate to decreased power consumption. Hardware manufacturers had hit the so-called “power wall”: transistors had reached sizes where higher frequencies led to even higher leakage currents. Ensuring the resulting thermal dissipation became increasingly difficult and energy-consuming [Bos11]. Whereas the previous years showed increasing frequencies with every new processor generation, increasing frequencies further had become uneconomic.

To keep satisfying Moore’s law, processor manufacturers, therefore, turned to new strategies to deliver performance increases. Instead of raising the complexity and frequency of a single processor, they increased the number of processor cores in processors, thus creating so-called Symmetric Multiprocessing (SMP) architectures. This parallelism, however, came at a price. In the words of Herb Sutter, “[t]he free lunch [was] over” [Sut09]. Dual-core processors, for

¹One could argue, though, that this law has become a self-fulfilling prophecy, as manufacturers now target this “law” to fulfill consumer expectations.
example, do not automatically make programs run twice as fast. Instead, parallel programs require different designs to profit from the provided parallelism.

This parallelism has become commonplace nowadays. For example, it is typical for embedded platforms, such as those used in mobile phones, to integrate at least eight cores on their processors, and highly parallel Multiprocessor System-on-Chip (MPSoC) featuring much higher core counts have become commonplace. Whereas the Intel SCC [Mat+10] with 48 cores was an early, experimental architecture primarily serving as a research platform, similar integrated systems are market ready as of today. For example, as of 2021, Tesla announced their Dojo D1 chip [Tes21], which features a similar architecture. Likewise, the Kalray MPPA, which has 80 cores [DH20], is commercially obtainable.

All these systems share one characteristic: their architectures differ considerably from that of simple SMP machines. One reason is that the connection of the processor cores to the shared memory can no longer be realized efficiently with many cores. Instead of a single shared memory addressable over a commonly shared bus, these systems feature multiple distributed shared memories. As such, these systems are commonly coined Distributed Shared Memory (DSM) systems.

A common variant of DSM architectures is the Non-Uniform Memory Access (NUMA) architecture [HP17]. NUMA architectures organize memory in NUMA domains, where each domain has a close link to a single CPU. From the CPU’s perspective, the memory of all domains is equally addressable. However, access to the remote domains takes longer as it requires multiple interconnect traversals. Consequently, these systems promote the exploitation of locality: applications should favor memory accesses in the local NUMA domain if possible and rely on cross-domain communication only if necessary. One emergent technology that further amplifies this style of memory organization is High Bandwidth Memory (HBM). HBM organizes the memory banks vertically to connect them even closer to the CPUs. Together with wide memory interfaces, HBM thus achieves high memory access bandwidths while lowering at the same time the overall power and chip area consumption. However, these properties come at the price of increasing heterogeneity for memory access times.

A complementary architectural approach that can, for example, be found in the Intel SCC, Kalray MPPA, and Tesla Dojo D1 systems mentioned above is that of the tile-based architecture. In these architectures, the embedded MPSoC is made up of multiple tiles. Each consists of a small number of cores, local memory, and possibly additional hardware. Function-wise, tiles are often similar to traditional SMP systems, as they also feature a local bus. However, on the other side, the communication between the tiles is often realized by a Network-on-Chip (NoC). NoCs thereby have the advantage that they scale better than bus systems, especially for high tile numbers. Furthermore, this design encourages the co-location of data and program code to tiles to exploit the locality without suffering from the delays that memory access to other tiles would entail.

However, although NoCs generally provide good scalability, these interconnects come with the downside that they complicate cache coherence protocols. While snooping-based coherence protocols are relatively trivial to realize for bus systems, it is difficult to adopt them for networks.
Conversely, directory-based coherence protocols often involve complex state machines and large tag storages, resulting in increased hardware complexity, communication resources, energy consumption, and verification overheads at design time. Although scalable cache coherence systems exist [MHS12], several tile-based architectures, therefore, drop cross-tile coherence altogether. These architectures typically restrict coherence within the scope of a tile and instead rely on an explicit message-passing model for cross-tile communication. This absence of a global coherence mechanism does not change much from the perspective of a process running on a single tile: both local and remote memory accesses appear cache coherent as they go through the same cache hierarchy. However, if a process spans multiple tiles, it can no longer rely on a coherent view of data shared between the tiles.

So, while tile-based architectures promise scalability for even highly parallel systems, their lack of cache coherence poses a challenge for programmers. Whereas the data organization within the scope of NUMA domains and tile boundaries works as customary, the question remains as to how data access and synchronization across tile boundaries should be organized. This dissertation takes this question as a starting point and investigates the design space for operating system-side support means to provide a simple programming interface. The dissertation, therefore, presents a lightweight operating system service that aims to aid parallel applications to exploit the potential of tile-based architectures.

1.1 Problem Statement

The trends outlined above show that computer architectures and memory hierarchies, in particular, undergo significant changes. Tile-based architectures are a promising approach for scalable, highly parallel systems. However, as it has been with parallelism, these architectures will not automatically yield an increase in performance. Instead, careful reconsideration of the design and implementation of parallel applications is indispensable. Consequently, a suitable programming model for tile-based architectures that considers the architecture and, more importantly, the lack of cache coherence between tiles is necessary.

To better understand the possible design space, it is sensible first to get a better understanding of the problem at hand and, in addition, to revisit established techniques.

In tile-based architectures, communication between tiles is significantly more expensive than communication within the scope of a tile. Drawing from that and their similarities with NUMA systems, it stands to reason that applications for these architectures will foremost exploit local parallelism and communicate across tile boundaries less frequently. Consequently, a suitable programming model should allow for fine-grained parallelism in the tile scope and coarse-grained communication between tiles.

When looking into existing parallel programming models, the best-known and understood models are shared memory and message passing\(^2\). However, the discussion regarding the

---

\(^2\)Of course, there are other programming models, as well. One noteworthy example is the Partitioned Global Address Space (PGAS) model, which we will cover in more detail in later chapters. It can be seen as some middle ground between message passing and shared memory.
suability of one model over the other is far from new, and both models have been found to be equivalent [LN79].

- **Shared memory**: This model probably is what most programmers consider the natural approach to parallelism. It typically expresses control flows as threads sharing a single shared address space. Concerning the hardware, communication happens implicitly by storing to and reading from a shared address space, and the control flows explicitly coordinate their access to shared data. Shared memory thereby supposes an underlying mechanism that maintains a consistent view of the shared memory. Typically, but not necessarily, this mechanism is realized by the hardware with the cache coherence protocol.

- **Message passing**: In this model, all communication happens explicitly by message exchange. Consequently, this model does not assume a shared memory. In fact, for example, in MPI (the Message Passing Interface), control flows are realized as separate processes. Furthermore, message-passing systems are typically realized in software and thus only require essential hardware support to realize the communication.

Fundamentally, these programming models differ in which system layer they situate the task of control flow communication. In shared memory programming, communication is implicitly realized with loads and stores to the shared address space. The underlying layers implement this interface, typically by the coherence system. Message passing moves this responsibility to the software domain, sidestepping hardware besides the communication infrastructure.

However, both models fail to capture the peculiarities of tiled architectures fully. Consequently, both programming models are not ideally suited for these architectures.

As outlined above, coherence is a necessity for shared memory. Though experimental and commercial tile-based platforms exist that feature global cache coherence, these coherence mechanisms generally limit scalability for tile-based architectures rather than bringing a benefit.

More importantly, however, is, drawing from NUMA platforms, the fact that parallel applications already bypass their cache coherence systems as much as possible. By keeping memory accesses in NUMA domains, locality can be exploited and costly cache invalidations mitigated. This is also observable in the growing interest in “NUMA-aware” operating and runtime systems. These systems, for example, feature elaborate strategies to place memory pages in the correct NUMA domains or to migrate them between domains. Similarly, operating systems implement schedulers [Theb], allocators [Thea], and lock algorithms [KMK17; DK19] that take into account NUMA configurations.

Therefore, reintroducing global coherence protocols to tile-based systems to maintain coherence between tiles would mean reintroducing a hardware feature already actively avoided by software.

On the other hand, the explicit communication of message passing is only suitable for some problem sets. For example, complex data structures require time-consuming serialization and deserialization efforts. These serialization and deserialization operations often include allocations and translating memory addresses from one address space to another, making them more
1.2 Contributions of this Dissertation

As established in Section 1.1, this dissertation identifies a need for a suitable programming model for tile-based architectures. We found neither hardware-based shared memory nor message passing to be sufficiently suited, as they fail to capture the identity-forming characteristics of said architectures. Other programming models, such as PGAS, address this issue. With these, however, similar to message passing, communication involves high costs for serialization and deserialization.

The contribution of this dissertation is a suitable parallel programming model for tile-based architectures. It, therefore, follows the “end-to-end argument” of Saltzer, Reed, and Clark [SRC84] and argues for situating the means of communication and, thus, the programming model at a higher system level. Instead of relying on hardware-based cache coherence as the basis for shared memory programming, this dissertation proposes shifting the issue of coherence in the form of memory consistency into the software domain. The hardware level does not necessarily know when cache coherence is mandatory, so it must be overly conservative per design. On the other hand, the software domain has the necessary contextual information to implement consistency, and thus coherence, more efficiently.

To this end, this dissertation describes the design and implementation of a scalable software-based consistency system. The consistency mechanism thereby draws inspiration from techniques first proposed and established for VSM systems [Li88]. The resulting consistency system...
is integrated as part of the operating system and provides consistency with weak memory consistency guarantees.

Furthermore, this dissertation proposes an adaptive software consistency mechanism. This mechanism builds upon the software-consistency system mentioned above to provide consistency for shared data while using the non-coherent hardware caches to provide fast memory access to unshared. Thus, the adaptive software consistency approach allows for better scalability and reduced memory overheads.

The consistency system described in this thesis comes close to the programming model that tile-based architectures suggest. It focuses on exploiting locality and parallelism within the scope of a tile. At the same time, memory access and data sharing across tile boundaries are possible. This shift of responsibility for coherence from the hardware domain to the software domain means that the consistency mechanism can be realized as a flexible on-demand feature.

1.3 Relevant Publications

Parts of the fundamental software coherence mechanism described in Chapter 4 and some of its optimizations have been presented as a workshop article [Lan+21]. I was the first author of this article and performed the implementation and evaluation with application benchmarks.

1.4 Thesis Structure

The following outlines the structure of this dissertation with a brief description of each chapter’s contents and its incorporation into the thesis.

Fundamentals

This chapter gives an overview of this work’s context. Therefore, it establishes the terminology used in this dissertation and the essentials of the used technologies and concepts. A particular focus thereby lies on the Invasive Computing (InvasIC) platform, as this system was used as a basis for the concept developed in the scope of this dissertation. In addition to that, the chapter, furthermore, gives an introduction to the concepts of cache coherence and memory consistency.

State of the Art

State of the Art gives an overview of research and technologies related to the research in this dissertation. As this thesis is concerned with cache coherence, the primary concern of the proposed approach is the shared memory programming model. Appropriately, this chapter gives an overview of the particularities of this programming model. In addition, this chapter discusses the differences between this model and other programming models for parallel systems. Secondly, the chapter presents an in-depth discussion of the current state of research for different approaches to cache coherence. The focus of this discussion, thereby mainly, lies on coherence
1.4 Thesis Structure

mechanisms for highly parallel platforms. Finally, this chapter discusses research in the adjacent field of VSM systems, which are also the base for the consistency concept described in this thesis.

A Software Consistency System

This chapter presents the SCS as an alternative approach to coherence. The SCS thereby is realized as part of the operating system. The chapter first describes the general approach and strategy of the SCS. This mechanism approaches the issue of cache coherence through virtual shared memory, similar to VSM systems. Then, the chapter describes the architecture and implementation of the SCS. Thereby the chapter details several optimization strategies that the SCS integrates.

An Adaptive Software Consistency System

This chapter presents an adaptive extension of the SCS, the ASCS. This extension takes advantage of the memory-sharing information that is available at runtime. Depending on how applications access their memory, this extension can use hardware last-level caches and thus achieve better performance than the SCS. The chapter, therefore, firstly motivates the adaptive caching strategies with a performance comparison of the SCS and the hardware caches. Then, the chapter describes the strategy for the adaptive extension and how this strategy is integrated into the ASCS architecture. Finally, the chapter discusses the implementation of the ASCS.

Runtime System for Shared Memory Programming

This chapter introduces a runtime system for parallel applications. This runtime system builds upon the memory consistency mechanisms of the two preceding chapters and provides the means to realize fork-join applications in the InvasIC prototype system. Therefore, the chapter first details the overall design of the runtime system. After this, it describes the design and implementation of the synchronization primitives that it provides.

Evaluation and Discussion

This chapter details the evaluation results of the proposed system designs. For the evaluation, the SCS and its adaptive extension are studied in detail regarding their runtime performance. Their overall timing behavior is examined utilizing micro-benchmarks. Furthermore, this chapter presents experimental results for the overall performance and scalability that the SCS and ASCS provide, at the example of benchmarking applications from the SPLASH-3 benchmark suite.

Conclusion

The conclusion, finally, presents a summary of the thesis and gives an outlook on possible further research directions.
2 Fundamentals

In order to get a better understanding of the issues this thesis addresses, it is helpful to start with the technical and theoretical topics this dissertation touches on. Ultimately, we can thus develop a better understanding of the design space for possible solutions.

Regarding this dissertation’s subject, we can make out two subjects that we will discuss in the following. The first is the research project that sets the context for this work. This research project provided the hardware platform and operating system that are the basis for this work. As such, these components significantly influenced the solution that this thesis presents. The second subject derives from the nature of this dissertation’s topic, namely the principles and concepts of cache coherence and memory consistency. This topical divide serves as the structure for the remainder of this chapter.

2.1 Principles of Invasive Computing

In Chapter 1, we established that computer architectures must increase in parallelism to keep on achieving performance growth. This increase in parallelism, unfortunately, poses new challenges for the design and organization of computer systems. One issue that, for example, stands to reason is resource management. As parallelism increases, centralized strategies become bottlenecks and thus hamper further scalability.

The German Transregional Collaborative Research Centre 89 “Invasive Computing” set out to explore novel solutions to address the challenges inherent to the design of highly parallel computer systems. InvasIC, therefore, has the ambition to create new methodologies for system design, organization, and programming, as well as paradigms for MPSoCs in which core counts exceed the hundreds [Tei+10].

The concept fundamental to the InvasIC research project is that of resource awareness. Resource awareness describes the property of applications to be aware of the system’s resource conditions and act accordingly. More concretely, a resource-aware application can evaluate the status of the systems hardware resources, like its availability, its utilization, or physical attributes like temperature and power consumption, and choose a suitable execution strategy or even adapt its present execution strategy at runtime in order to achieve its functional and non-functional requirements. This adaption of the execution strategy may vary from allocating additional resources to adjusting parallelism or even the precision of calculation results. Resource-aware applications, thus, are in a constant feedback loop with the hardware [Tei+10].
However, more than making applications resource-aware is needed to address the scalability issue in many-core systems. The InvasIC research project, therefore, argues that in the presence of a plethora of computing as well as other resources, the necessity to virtualize and share them among competing applications ceases to be [Tei+10]. Ideally, each application should have exclusive access to the system’s resources. This resource distribution has the positive side effect of reducing virtualization costs, such as overheads for context switches. For example, it even obliterates the need for performance-intensive isolation and security mechanisms to prevent cross-application side-channel attacks. The exclusivity, thereby, is considered to be the default. Of course, applications may still opt to share and virtualize their resources, for example, by specifying constraints in their resource allocations or using annotations. In the remainder of this dissertation, however, we will primarily consider the case of exclusive resource allocation.

InvasIC builds upon this insight and proposes the eponymous InvasIC paradigm. This paradigm describes a complete system design consisting of specialized hardware and software components, realized with hardware-software co-design. The resulting InvasIC system thereby involves multiple specialized hardware units and software systems, such as a compiler, operating system, and middleware infrastructure.

Applications following the InvasIC paradigm (also referred to as InvasIC applications) are granted exclusive access to hardware resources. The allocation and utilization of the resources take place in three phases that InvasIC applications repeatedly traverse. A closer investigation of the lifetime of an application in the InvasIC illustrates this concept best. Figure 2.1 illustrates the lifetime of such an application as a state diagram. An InvasIC application starts in the invade phase. In this phase, it allocates the resources it needs for its execution. These are usually processing elements but can also include other resource types, such as NoC bandwidth. In InvasIC terminology, this allocation is called invasion. When the invasion succeeds, the operating system makes the resources available to the application in the form of a claim. A claim is a collection of resources that exclusively belongs to an application. After the allocation, the application may invade further resources or continue in the infect phase, where it uses them for its execution. In the case of processing elements, the application will bring so-called \( \ell \)-lets\(^1\), lightweight control flows, to execution. Finally, the application frees the resources of the claim in the retreat phase. Individual applications may run through these steps independently of

---

\(^1\)See Section 2.1.2 for a more accurate description of the \( \ell \)-let concept.
2.1 Principles of Invasive Computing

each other. A resource-aware InvasIC application thus can manage its resource requirements autonomously.

2.1.1 Invasive Computer Architecture

From an architectural perspective, InvasIC assumes a custom hardware platform as the basis of an InvasIC system.

Henkel et al. [Hen+12] give an overview of this hardware platform and describe the functionality of the parts it comprises. As can be seen from the overview in Figure 2.2, the InvasIC system organizes the respective processing resources and memory in a heterogeneous tile-based architecture. A NoC thereby facilitates the communication between the tiles. Furthermore, this NoC provides additional support for the InvasIC paradigm in the sense that it provides invadable quality-of-service guarantees [Hei+14]. Generally, the architecture primarily consists of so-called compute tiles, mainly made up of general-purpose CPUs.

In order to facilitate a test environment, the InvasIC hardware design is realized with an Field Programmable Gate Array (FPGA) system. However, while the InvasIC project envisions MPSoC platforms comprising hundreds to thousands of processor cores, realizing such scope with prototyping FPGA systems is infeasible. Hence, at the time of writing, the InvasIC concept is typically evaluated at a smaller scale. This, however, is not to say that the devised concepts do not scale but rather that the test environment does not provide the necessary capabilities.

So, for example, a typical evaluation system configuration comprises compute tiles that each feature five LEON3 cores [Gai02]. The LEON3 cores thereby realize a 32-bit Reduced Instruction Set Computer (RISC) architecture implementing the SPARC version 8 Instruction
Fundamentals

Set Architecture (ISA) [SPA92]. Each of the SPARC cores has its own L1 cache. These caches are kept coherent. Besides the SPARC cores, the tiles additionally integrate 8 MiB Static RAM (SRAM) called the Tile Local Memory (TLM). The cores can access this memory across the Advanced Microcontroller Bus Architecture (AMBA) bus [Fly97]. Finally, tiles may integrate special-purpose hardware accelerators such as the t-Core [Hen+11] or the tightly-coupled processor array (TCPA) [Mud+12], bus systems, e.g., to attach additional memory and IO devices.

In addition to hardware accelerators for computational tasks, the InvasIC architecture integrates several specialized hardware components that assist operating system tasks. The Core i-let Controller (CiC) [Puj+11], for instance, takes care of i-let scheduling under the consideration of multiple system monitors. It does so by taking into account factors such as system load and temperature when choosing the core to assign the i-let to [Puj+11]. Furthermore, the CiC takes care to wake sleeping cores when it assigns i-lets to them. Other hardware units, such as the Direct Memory Access (DMA) unit, make use of the CiC to signal state changes by scheduling i-lets directly instead of raising interrupts or setting memory-mapped registers.

For NoC communication, the tiles furthermore integrate a Network Adapter (NA), which is connected to their buses. This NA resolves memory accesses to remote memory and realizes them as NoC communication. In order to decrease costs for memory accesses across the NoC, each tile additionally integrates a shared L2 cache with a write-back policy. The L2 cache caches any remote memory accesses. The contents of the L2 caches, thereby, are not kept cache coherent. From a tiles processing elements’ point of view, thus, both the TLM and memory integrated into other tiles are accessible. Memory accesses from a single tile to remote memory appear cache coherent—as long as no cores on other tiles are involved.

The NA furthermore facilitates a mechanism to invoke i-let execution on remote tiles. Using this feature, the NA also implements a DMA mechanism that allows asynchronous transfers of continuous memory buffers. Upon completion of a DMA, the NA therefore schedules i-lets on the sending and receiving side. Typically, this feature is used to execute completion i-lets. On the initiating side, the completion i-let can free the memory of the sent buffer. In contrast, the completion i-let on the receiving end continues to process the received data.

A further notable accelerator is the Software-Defined Hardware-Managed Queues (SHARQ). This component addresses a shortcoming of the DMA unit: typically, when using a DMA transfer, an application must first reserve a buffer on the receiver side and transfer the location of this buffer back so that the actual DMA can be initiated. This procedure is both cumbersome and expensive. The SHARQ improves over this by combining both steps into a single hardware unit. The receiving end provides a list of free buffers and a queue where filled buffers are enqueued. When the sender now initializes a transfer, the SHARQ unit fetches a free buffer from the far side, performs a DMA to it, enqueues it into the queue, and finally triggers a handler i-let on the far side. This handler i-let can then dequeue the data and process it [Rhe+19b].
2.1 Principles of Invasive Computing

2.1.2 OctoPOS — The Operating System for Invasive Computing

So, as we have seen above, the InvasIC paradigm aims to grant applications the ability to autonomously adapt to system parameters following their resource demands. Whereas the InvasIC hardware attempts to assist this approach where possible, resource augmentation and management traditionally are responsibilities of the operating system. This division of responsibilities also holds for the InvasIC system. Therefore, the InvasIC system employs a custom designed developed operating system OctoPOS [Oec+11; Oec18].

In the following, we will explore the peculiarities and properties of this operating system. In particular, we will focus on the areas of the operating system that have significantly influenced the design of the solutions developed in the context of this dissertation. We will do so by closely examining the operating system’s fundamental design principles. Secondly, we will have a closer look into how OctoPOS realizes resource management. In the course of this, we will take a look at how OctoPOS implements the claim concept. Next, we discuss the execution model realized by the operating system. Finally, we will discuss in more detail how OctoPOS organizes the memory of the InvasIC platform.

Design Principles of OctoPOS

To take into account the tiled architecture of the InvasIC architecture, OctoPOS is realized as a distributed operating system. Therefore, it follows the multikernel principle popularized by Barrelfish [Bau+09]: instead of a single, centralized kernel, a distributed operating system consists of multiple kernel instances. Each kernel thereby governs only a fraction of the system. This design allows for the application of specialized kernels for different system parts and, at the same time, mitigates the kernel being a bottleneck.

OctoPOS realizes this paradigm by providing a kernel instance per tile. Instead of using shared memory for the communication between kernel instances, OctoPOS uses an asynchronous Remote Procedure Call (RPC) mechanism [BN84], and makes use of the DMA mechanism. This design comes naturally, given that the InvasIC hardware architecture does not provide cache coherence across tile domains.

Finally, the OctoPOS kernel is realized as a library operating system. The kernel is shipped as a library, which then must be linked against application binaries to form the final system image.

Resource Management

As described earlier, InvasIC assumes platforms that consist of a sufficiently large number of computing elements so that, most of the time, applications do not need to share them. If an application chooses not to share resources, it can make exclusive allocations. Such an exclusive resource allocation eliminates the necessity for techniques such as temporal multiplexing to virtualize the resources. Instead, the operating system provides spatial multiplexing.
OctoPOS realizes this spatial multiplexing in a way that makes applications apparent of their current share of allocated processing elements. In contrast to contemporary general-purpose operating systems, applications, thus, have pools of multiple processing elements at their disposal. OctoPOS realizes these pools with in claims.

As applications in OctoPOS may span multiple tiles, they must be aware of the locality of computational resources, as remote resources inherently involve different usage semantics than local resources. Therefore, OctoPOS distinguishes between elementary claims, which are claims to local resources, and proxy claims, that facilitate the means to manage resources and execute i-lets on distant tiles. Finally, in addition to proxy claims, OctoPOS additionally provides dispatch claims as a lightweight handle to a remote claim. The primary purpose of dispatch claims is to be transferred between tiles.

While not directly involved with this thesis, the agent system [Kob+11; Kob15] should also be mentioned at this point for the sake of completeness. The agent system augments the above operating system interface with a richer resource management interface. This interface enables applications to allocate resources by formulating resource requirements through constraints. These constraints can, for example, describe requirements like the number of processing elements, features, and their location in the system. Finally, multiple constraints can be expressed as propositional logic to capture possible configurations. The agent system then uses an agent-based solver to find allocation configurations that fulfill these constraints. Taken together with OctoPOS, the agent system forms the Invasive Runtime Support System (iRTSS).

### i-let Execution under Exclusive Resource Allocation

When OctoPOS executes an application on an exclusive processor allocation, it can leverage a lean execution model with support for microparallelism. This execution model allows applications efficiently parallelize even short code paths, such as loop bodies. As mentioned before, the CiC thereby takes the place of the scheduler. What remains for OctoPOS is the execution of i-lets, featherweight threads. The operating system executes these threads in run-to-completion semantics. This means that once an i-let is dispatched, it is not preempted unless it explicitly yields control over the processor. Doing so allows it to reduce the number of context switches significantly. Moreover, OctoPOS can thus create and manage i-lets cost-efficient even in large numbers.

By taking into account the run-to-completion semantics, OctoPOS can implement scheduling quite efficiently, as it can reduce context switch overheads and the size of execution stacks. Furthermore, the kernel can use the fact that control flows that will not block can be stored efficiently in high numbers, as they can share an execution stack. Consequently, in OctoPOS, it is beneficial to realize synchronization by creating handler i-let's that are scheduled once the waiting condition is fulfilled—as opposed to the more “traditional” approach, where control flows block until their waiting conditions are fulfilled. See [Oec18] and Appendix A for a more detailed explanation of how the execution model is realized in OctoPOS and how this affects both, the scheduling system, and the construction of synchronization in applications.
2.1.3 Memory Organization

Provided with a better understanding of the core concepts of OctoPOS, we now come back to the primary concern of this dissertation: the way that the operating system organizes and presents memory to the applications. To do this, we first reach back and look in more detail into the memory architecture of the InvasIC system.

As stated earlier, the InvasIC architecture consists of multiple tiles, whereas each of these tiles integrates local memory called TLM. The structure of the system, thereby, is comparable to a NUMA system. OctoPOS uses the TLM to store its text and data segment and those of applications. Furthermore, it is the primary source for dynamic memory allocations.

However, as the TLM is limited in size, one or more tiles of the system additionally integrate Double Data Rate (DDR) memory controllers to which external memory is attached. The DDR is also accessible from within the tile it is connected to and across the NoC. Per convention, OctoPOS statically portions the DDR memory among the system’s tiles: it divides the memory into even shares and allots those to the respective tiles. This configuration reflects in the denotation of this memory as Shared Memory (SHM).

As described earlier, each tile has an L2 cache that caches memory access across the NoC. However, there is no coherence mechanism for the L2 caches in place. However, as all memory accesses from a single tile traverse the tile’s L2 cache, remote memory appears to be cache coherent from the perspective of this tile’s cores. Only when cores of different tiles are involved the view of shared data becomes incoherent.

The InvasIC system presents each tile with a unique memory map. These memory maps consist of three types of mappings: the mapping for the SHM, the local TLM mapping, and the global TLM mapping. Figure 2.3 exemplifies this from the perspective of tile 0. The memory maps of the other tiles follow this structure. As can be seen from this example, the memory maps start with the mapping of the SHM. This mapping is then followed by the local TLM mapping. Then follow the global mappings of all tiles’ TLMs. This means that each tile’s
2 Fundamentals

The memory map contains the tile's TLM twice, once in the local and once in the global mapping. The global portion of the memory map thereby is constructed similarly for every tile and thus provides globally valid addresses for all TLM memory portions. This allows addresses to be passed freely between tiles without requiring translations between address spaces.

So while the global mapping provides globally valid addresses, the local TLM mapping is only for use by the local operating system instance and applications. The additional local mapping provides some advantages. The local mapping allows using the same operating system image for each tile without additional memory virtualization, as the local addresses for each tile are the same. Drescher [Dre21] uses this, for example, to offer memory protection as an adaptive on-demand feature. Having the same text segment addresses for every tile furthermore simplifies the RPC mechanism. The RPC mechanism requires no additional indirection and can be implemented with plain function pointers. The downside is that applications must explicitly convert between addresses of the local and global mapping if they want to share them with other tiles. However, this conversion process is relatively simple and inexpensive. One can easily convert between the mappings using the size of the tile mapping, the global and local mapping offsets, and the respective tile's numeric identifier.

The memory view realized by the memory map effectively shares similarities with the PGAS programming model: tiles may transparently access both local and remote memory as both memories reside in the same unique global address space. The significant difference between the mappings is whether cache coherence is maintained.

With the description of the memory map, we conclude the overview of the InvasIC system. In the following, we will discuss cache coherence and the related concept of memory consistency.

2.2 Cache Coherence and Memory Consistency

In Section 1.1, we saw that tile-based architectures sacrifice cache coherence to achieve higher degrees of parallelism. This dissertation aims to provide a shared memory programming model with a notion of cache coherence where the hardware provides none. This section aims to build an intuition of the necessary concepts for such a coherent shared memory model.

The following will, therefore, provide a working definition for cache coherence and derive some considerations and implications that come with this definition. Furthermore, this section will show that more than coherence is necessary for a functional shared memory system. Therefore, it discusses the concept of memory consistency and shows that weak memory consistency is sufficient for cache coherence on-demand as the problem statement requires.

2.2.1 Cache Coherence

In principle, the concept of caches is simple. Caches are fast memory buffers that store copies of frequently accessed memory in cache lines to reduce memory access latencies. Under closer consideration, however, this concept prompts the question of how caches should handle writes. This question is considered under the term write policy. Generally, there is the write-through
2.2 Cache Coherence and Memory Consistency

and the write-back write policy. In the former strategy, writes are directly performed on cache and main memory as they occur. Hence the name, as the operation, writes “through” the cache into the main memory. With this strategy, any write directly becomes visible in the main memory. The downside is that every store operation thus experiences the main memory access latency. In the write-back policy, on the contrary, a store only modifies the cache contents. The modifications thus stay in the cache as long as the affected cache line remains in the cache. They are written back to memory when the cache evicts the cache line. The modifications are, therefore, only visible to the cores that share the cache. With either strategy, caches are transparent for the CPU: besides differences in memory access times, a CPU cannot tell whether its memory access was serviced by the main memory or by the cache [SHW11, p. 2].

Figure 2.4: Picture visualizing situations that lead to inconsistencies in caches without cache coherence protocol. Left is a cache with write-back, and right is a cache and write-through policy. With write-back policy, stores are written to the cache but not to memory. Subsequent loads thus result in stale values. With write-through caches, earlier cached values are not updated and become stale, when stores modify the cache and the memory.

However, this illusion of transparent caches does not hold up in parallel systems. Here, each CPU core typically has its own cache. This can lead to a situation where some data from the main memory is stored in multiple caches. If not handled correctly, these cached copies can become inconsistent, and, as a consequence, the program may fail to function correctly. Figure 2.4 illustrates these inconsistencies. On the left, two CPU cores, A and B, each with a cache with write-back policy, try to access memory. CPU core A fetches the data word W from memory which is then stored in As cache. When A now modifies its local copy. Following the write-back policy, this modification is only performed in the cache. As only As cache knows the most current state of W, its value in the main memory is outdated. It has become stale. If B now fetches the stale value of W, it will not see As latest changes. The view to B has become inconsistent, and the program is likely to malfunction. A similar problematic situation can be constructed with write-through caches, as demonstrated in the right of Figure 2.4.

These examples emphasize the necessity to maintain coherence for the contents of all caches, so-called cache coherence. Modern parallel architectures, therefore, typically implement cache
coherence protocols, or coherence protocols for short. In his dissertation Gharachorloo [Gha95, p. 22] defines the task of such a cache coherence protocol as:

“A cache coherence protocol typically ensures that the effect of every write is eventually made visible to all processors (through invalidation or update messages) and that all writes to the same location are seen in the same order by all processors.”

Researchers and manufacturers proposed diverse approaches to realize cache coherence. In principle, all these models function similarly: they track which cache line is stored in what cache and whether the lines are shared. In addition, they track modifications of cache lines. When a shared cache line is modified, the coherence protocol either updates all replicas with the new value or removes it from the respective caches. The latter operation is called an invalidate in cache coherence terminology.

For this purpose, these protocols typically implement a finite state machine per cache line to track its current state. A cache line’s state information typically includes data such as the number of sharers. Coherence protocols are commonly coined after the names of their defining cache line states, such as the Modified Owned Exclusive Shared Invalid (MOESI) protocol [SS86].

Cache coherence protocols can further be distinguished by whether they are either snooping-based or directory-based. In snooping-based coherence protocols, the caches watch, or “snoop”, the bus for modifications of the cache lines that they currently store. If they see a store that modifies one of their cache lines, they either update its value or invalidate it. This type of coherence protocol, therefore, requires that stores are broadcasted. As stores in bus-based systems effectively are broadcast, these systems allow for efficient snooping-based coherence implementations.

On the contrary, in directory-based cache coherence protocols, a directory stores the state and sharers of cache lines. Upon modification, a cache must first access the directory to look up whom to notify about the changes in the cache line. This design eliminates the need for a shared bus and thus is better suited for interconnects, such as NoCs. However, this design also comes at the cost of storage for the directory. We will discuss the particularities of different coherence protocols in more detail in Section 3.2.

However, coherence protocols also introduce problems, such as false sharing [BS93; Bol93]. In the case of false sharing, semantically independent data words are located on the same cache line—that is, they share a cache line. As a result, write access to one of the data words causes the coherence protocol to invalidate the cache line in the caches of all sharers, even if the sharers do not access the written data word. Thus, the coherence mechanism does not result in a gain of information—but in a performance loss.

At first glance, coherence protocols fulfill the necessary conditions to realize functional shared memory systems. However, this is not the case. When we revisit the definition above, we see that, in fact, cache coherence only gives guarantees regarding the visibility of modifications to single cache lines. In the next section, we show that the ordering guarantees of coherence protocols are insufficient to prevent inconsistencies.
2.2.2 Memory Consistency Models

The programmer’s intuition dictates that loads from and stores to memory happen in the order specified in the program. This order is appropriately called program order.

Most of the time, this intuition is correct—or, more precisely, most of the time, programs behave as if the intuition were correct. For example, many modern CPUs optimize the execution speed of single-threaded execution under the premise that “everything goes” as long as the result is the same as for unoptimized execution\(^2\). Commonly applied examples for such optimizations are out-of-order execution and write coalescing. The problem with these optimizations arises in parallel systems. While for a single thread, the optimized code appears to be in program order, threads running on other CPUs may see the effect of the optimizations and thus the reordering of the memory accesses.

```java
1. auto node = new Node {42};
2. node->next = head;
3. head = node;
4. count += 1;
```

Listing 1: Example of two threads with a producer-consumer relationship. The thread on the left, \(C_1\), adds a new element into a stack-like data structure and then, on line 4, signals that a new element has been added to the data structure. The thread on the right, \(C_2\), blocks in line 1 until the stack contains elements and then removes an element from it.

The example in Listing 1 illustrates how this may affect the correctness of programs. The example shows excerpts of two threads \(t_1\) and \(t_2\) in a producer-consumer relationship. Consider that the threads are executed on individual processor cores. \(t_1\) enqueues an item into a list and signals the waiting \(t_2\). From the code of \(t_2\) becomes apparent that the order that \(t_1\) performs its stores in (and, for this matter, the order in which \(t_2\) observes them in) is vital for the correctness of \(t_2\). If \(t_2\) were to read the flag variable before the stores of \(t_1\) to enqueue the element are visible, \(t_2\) would continue with incorrect data.

Typically, programmers can enforce program order, if necessary, with special instructions better known as memory fences. These special operations ensure the completion of memory operations and thus ensure (partial) ordering. To be able to write correct code, programmers must thus be aware of what types of reorderings may occur in order to be able to place the memory fences correctly. Hence, hardware manufacturers provide memory consistency models or shorter memory models that specify what assumptions regarding memory ordering can and cannot be made. Typically, these memory models describe the types of memory access reorderings they consider valid. As such, they fulfill two purposes: firstly, they act as an interface description for the programmer. A programmer considering the memory model can ensure their code behaves

\(^2\)The same holds for compilers, as well. Consequently, large parts of this section also hold for programming languages. See, for example, the specification of the supposed memory model introduced to C++ with the C++11 standard [ISO12].
2 Fundamentals

as expected. Secondly, the memory models give hardware manufacturers guidelines on what guarantees their optimizations must provide\(^3\).

The memory model concept thereby contrasted with the concept of cache coherence we discussed above: cache coherence is concerned with the processors’ view of a single word in memory, whereas memory consistency is concerned with the view of the processors of all memory words. In fact, no cache coherence protocol is necessary for a parallel system to provide a well-defined programming interface as long as the memory consistency model is provided [SHW11, p. 21].

The following presents a discussion of some prominent consistency models. We start with the sequential consistency model [Lam79], with which Lamport first established the concept of memory consistency. Sequential consistency can be understood as an extension of program order for parallel systems. It allows all those memory orders resulting from any interleaving of the processors’ respective memory accesses in sequential order.

```c
1 a = 1;
2 printf("b: %i\n", b);
3 b = 1;
4 printf("a: %i\n", a);
```

Listing 2: Example of an application that leads to unexpected results under TSO.

In praxis, this sequential consistency has shown to be quite restrictive. In order to achieve better performance, modern processors, therefore, implement weaker memory models. While relatively short, the program in Listing 2 is sufficient to demonstrate how a program may behave differently depending on the underlying memory model. For example, suppose that both \(a\) and \(b\) are initialized with 0. Then, according to sequential consistency, the program may produce three possible results: \(a: 1\) \(b: 0\), \(a: 0\) \(b: 1\), or \(a: 1\) \(b: 1\).

While the LEON3 processor used in the InvasIC platform is relatively conservative regarding optimizations, the example above is sufficient to demonstrate the impact of weaker consistency models. In the case of the LEON3, this is the TSO model. This model allows optimizations, such as buffering outstanding stores in a store buffer, allowing loads to bypass the buffered stores. When taken together, these allow pending stores in the store buffer to be reordered with loads that come after them according to program order. As a result the example in Listing 2 can also result in \(a: 0\) \(b: 0\) under TSO. To prevent this reordering for example on the LEON3, programmers, therefore, have to explicitly use atomic read-modify-write operations [SPA92, p. 64].

Stronger memory models, such as TSO, are convenient as they are intuitive and thus easily allow reasoning about program correctness. These properties, however, come at the price that they allow only a few memory access reordering and thus inhibit many possible optimizations. However, manufacturers and researchers noticed that most programs do not necessarily need such strict ordering requirements [Mos93; Adv93; Gha+90]. Even though parallel shared memory applications use said memory to establish their communication, this communication most often

\(^3\)In reality, sadly, the situation is not as clear-cut. Often manufacturers implement and document various optimizations while the developer is left to reason about the specifics of the resulting memory model. Take as an example the Intel x86 architecture for which researchers “believe” that it is similar to Total Store Order (TSO) [SHW11, p. 44].
only makes up a small part of the code. Parallel programs usually spend most of their time exclusively on their respective working sets. Communication between control flows occurs far less frequently, either with explicit synchronization primitives, such as locks or with flag variables through atomic read-modify-write operations.

Consequently, one can distinguish memory accesses by their purpose in the program. Synchronizing memory accesses realize inter-thread synchronization, whereas data memory accesses are “regular” memory accesses. Listing 3 uses the producer-consumer example from earlier to illustrate this difference. As we know, \( t_1 \) enqueues data into a queue and signals the availability of new data afterward. \( t_2 \) in return, waits for this signal to happen. While all memory accesses are necessary for the program flow to work correctly, only the store to the flag and the read from it establish synchronization. We can see from the example that for the program to work correctly, it is sufficient that no stores are reordered after the synchronizing store on the producer side.

In return, no loads must be reordered on the consumer side before the synchronizing load. It suffices for the program to work correctly to “restore” consistency at these synchronizing memory accesses. All other memory accesses may be freely reordered as long as this condition holds.

The class of relaxed memory consistency models draws from this insight. Compared to the strong consistency models, these memory models allow by far more possible memory access orderings and, consequently, more optimizations. Weak consistency [DSB86], for example, explicitly distinguishes data accesses and synchronization accesses and requires them to be labeled as such. The only ordering restriction of weak consistency is that all synchronization accesses must be executed in program order and that data accesses may not be reordered with synchronization accesses of the same processor. Synchronization accesses thus act as a fence that enforces the ordering of the data accesses. Thus, data accesses between two synchronization accesses can be reordered freely. From the programmer’s perspective, the fence fulfills two purposes: first, it ensures that all other threads see the stores that the thread has performed up to the point of the fence. Secondly, it ensures that the executing thread becomes aware of the stores that other threads performed up to this point.

These fences are further differentiated in release consistency [Gha+90]. When considering the producer-consumer example again, it becomes clear that the threads perform two different types of synchronization accesses. The producing thread modifies the global state and subsequently notifies the reader about this modification, while the consuming thread wants to see this
notification. It suffices for the producer to ensure that its modifications are globally visible. For the consumer, it suffices to ensure that it sees the outstanding global modifications. To this end, release consistency differentiates two types of fences, the *release* and the *acquire* fences. The release fence enforces that all previous data stores have been completed but allows subsequent data accesses to be reordered with it. The acquire fence, in return, enforces that no subsequent data load operations are reordered with it. However, it allows reordering with previous data accesses. As with weak consistency, these synchronization operations must execute in program order. Informally, a release fence “publishes” the results of stores, while the acquire fence allows “fetching” the most recently published changes\(^4\).

Modern processors integrate further optimizations that result in even weaker memory models. However, the weak and the release consistency models are widely implemented and supported. Now the question remains of where to place memory fences correctly. The correct placement of fences is vital for the correctness of parallel code. The concept of *data-race-freedom* under sequential consistency [AH90; AH93] addresses this issue. Adve et al. showed [AH93] that it is possible to construct programs to behave sequentially consistently under even weak memory models, such as weak consistency and release consistency. The premise for this is that the program is, what the authors call, data-race-free under any sequential consistent execution. A data race in this context means situations in which two or more parallel data accesses can happen concurrently to the same data word, and at least one of them is a store. Programmers typically use atomic operations or mutual exclusion to ensure data race freedom. To guarantee data-race-freedom under sequential consistency, it then suffices to use appropriate memory fences for these synchronization cases.

### 2.3 Conclusions

With the description of cache coherence and memory consistency, we have established the necessary foundations for the remainder of this dissertation. However, before continuing, let us recapitulate the essential concepts described above.

As we have seen, the InvasIC system is tile-based and organized similarly to a distributed system. Additionally, it integrates numerous hardware units to accelerate, among others, the communication between tiles. OctoPOS is the operating system designed for the InvasIC platform and realizes the concept of exclusive resource allocation. The operating system is designed to support high parallelism efficiently and emphasizes a synchronization model that is best described as synchronization by control flow activation. Instead of letting single control flows or \(i\)-lets block until a synchronization primitive has been signaled, the synchronization primitive instead spawns a new control flow once signaled. Doing so allows for reducing context switch overheads and memory overheads. Software for the InvasIC system, consequently, should be structured to exploit this execution model.

\[^4\text{For a relatively intuitive explanation of these concepts, see Jeff Preshing’s blog post about the semantics of memory fences [Pre12]. Here he introduces the excellent analogy of memory fences being similar to the operations of a source control system.}\]
Finally, we described the concepts of cache coherence and memory consistency. As we have seen, parallel programs do not necessarily need cache coherence if provided with a memory model of the underlying platform. Furthermore, memory consistency models can be designed to allow for many optimizations by the underlying platform or compiler and still be programmable. While these models are not trivial, recent years have shown a commonplace adoption of these weak consistency models.
"There are very few starts. Oh, some things seem to be beginnings. The curtain goes up, the first pawn moves, the first shot is fired - but that's not the start. The play, the game, the war is just a little window on a ribbon of events that may extend back thousands of years. The point is, there's always something before. It's always a case of Now Read On."

— Terry Pratchett

3

State of the Art

Cache coherence mechanisms constitute a central part of parallel systems. Due to their crucial role, their efficient implementation has probably been a research topic for almost as long as parallel computers existed. In the following, we will glance at this research and discuss current trends, approaches, and alternatives to cache coherence. Therefore, we will first look closer at the current programming models for both systems with cache coherence and programming models that suffice without such an interface. Then, we proceed with an overview of different approaches to realize cache coherence in hardware and software.

3.1 Parallel Programming Models

Over the years, many parallel programming models emerged. The most significant and influential programming models, however, to this day remain message passing and shared memory programming. These models constitute extremes of a continuum regarding their memory representation. Message passing, sometimes referred to as shared nothing, provides a private memory per control flow. Shared memory programming, on the contrary, supposes a single global memory that all control flows share.

The distinction between these programming models is best understood when considering the historical context of computer architectures\(^1\). High Performance Computing (HPC) systems in the late 80s and early 90s typically had a low number of cores and, thus, naturally, were limited in their degree of achievable parallelism. To lift this limitation, multiple computers, commonly referred to as nodes in this context, were combined into a so-called computer cluster, as it is still praxis in today’s HPC systems. However, these cluster configurations had the downside that not all memory was equally accessible. Hence, the nodes in these clusters made up for the missing shared address space by explicitly exchanging or passing messages for communication. Consequently, these systems are commonly referred to as message passing systems. It would be the later increase of parallelism within the single compute nodes that led to the dissemination of the shared memory paradigm.

\(^1\)As mentioned in Section 1.1, considerations regarding the application of shared memory or explicit message passing for communication have been made at least a decade earlier [LN79].
A further noteworthy programming model that has gained popularity for HPC is the PGAS model. This programming model is noteworthy because it bridges the divide between shared memory programming and message passing. This model presents memory to the programmer as a single global address space, similar to shared memory programming. However, the physical distribution and, thus, possible higher accesses delays of memory become visible.

In Section 1.1, we provided a brief overview of message passing and shared memory computing. In the following, we want to contrast the features of these models. Hence, we will discuss message passing and shared memory. Finally, we will extend this discussion and take a closer look at the PGAS.

3.1.1 Message Passing

As suggested by its name, message passing systems facilitate communication between control flows through the explicit exchange of messages. The term message passing thereby describes a family of concepts. In the following, we use the term to describe a programming model that uses explicit messages exchange to realize communication. As we will see further below, this mechanism thereby does not necessarily suppose a specific implementation—for example, it is common to implement message passing with shared memory.

In the simplest case, message passing expresses its exchange with a send() primitive that sends data packages and a receive() primitive that receives packages. Besides establishing data exchange, these routines thereby fulfill a second purpose: they establish synchronization of control flows. As programming with these elementary routines alone is cumbersome, message passing implementations typically also provide more expressive broad- and multicast and synchronization routines.

There is quite some room for exploration in implementing the message passing primitives. For example, one grave decision, performance-wise, is whether the send() operation is synchronous or asynchronous. A synchronous send() blocks until the buffer containing the to-be-sent data can be used again in the application. An asynchronous send() returns directly, even if the buffer is not ready to be modified. Doing the former may entail additional delays for a context switch, while the latter requires some mechanism to signal when the buffer becomes reusable again. Similarly, the receive() primitive can be constructed synchronously or asynchronously. Tanenbaum and Bos [TB15, pp. 553–556] discusses further designs and their advantages and disadvantages.

Message passing implementations generally have only small requirements regarding the underlying communication systems and are thus quite versatile. For example, Open Message Passing Interface (MPI) [Sat02], a widely used open-source implementation of the MPI standard [For93], among others supports TCP sockets, Infiniband, and shared memory [OM19] as communication backends. Depending on the underlying backend, the performance thereby can vary.

---

2One way to realize this would be to have the send() procedure return a promise object [LS88] upon which the control flow can block until the buffer is free to be used again.
3.1 Parallel Programming Models

The most notable property of message passing is that it imposes a style of programming that can be relatively cumbersome at times and adds to program complexity. Furthermore, there appears to be some debate whether the increased complexity leads to error-proneness or, on the contrary, even prevents certain classes of errors. Steen and Tanenbaum [ST17, p. 26], for example, state that message passing has a reputation for being error-prone. In contrast, some modern programming languages adopt message passing for inter-thread communication to avoid certain error classes. Go, for example, states, “Do not communicate by sharing memory; instead, share memory by communicating.” [Go] and Rust follows this approach [KN18]. The principle idea here is that when the application state is not shared implicitly, no such state may be modified unsynchronized by accident.

All in all, message passing finds widespread use, for example, to program compute clusters, as is the case of HPC systems. Here, the MPI standard is well established and has become a synonym for the concept of message passing itself. This standard was proposed as a multi-programming language Application Programming Interface (API) in 1993 by a consortium of both research and industry [For93]. Commonly used implementations of the MPI standard are MPICH [Gro+96] and Open MPI [GWS06; Gra+06].

With the principles and application of message passing explained, it is time to look at shared memory programming. In contrast to message passing, this programming model supposes that all control flows share a mutual view of memory, and thus, communication occurs implicitly via this shared memory.

3.1.2 Shared Memory

In these shared memory programming systems, multiple processing units operate on a commonly shared memory. Communication thereby takes place implicitly via shared memory. This programming style arguably is the most intuitive realization of parallelism, as it extends sequential programming concepts to parallel computer systems. Similar to the execution of a control flow in a sequential application, the control flows of a shared memory application share a single view of memory.

Communication in these systems is considered implicit, as the control flows realize it with stores to and loads from the shared memory. As a result, shared memory programming generally benefits from a more approachable programming interface than message passing. As we have seen in Section 2.2.2, however, the premise for the correctness of shared memory programming is that the programmer has a far better understanding of the hardware they operate on and its consistency model.

In contrast to message passing, communication in shared memory programming typically does not imply synchronization. Control flows, therefore, typically must take matters into their own hands to synchronize their accesses to the shared data. For single data words, atomic operations suffice, whereas more expressive synchronization concepts are necessary to guarantee mutual exclusion for access to shared data. It should also be noted that atomic operations can be used to design algorithms that do not require explicit blocking [Her88].
Different platforms provide shared memory programming environments: in principle, however, they come down to multiple control flows sharing a common view of memory. Operating systems, therefore, typically provide means to map certain memory ranges as shared between processes. However, it is more common to realize shared memory applications with multithreading. The most prominent programming languages, libraries, and frameworks that implement this style of shared memory parallelism are Cilk [Blu96], oneAPI TBB [Rei07], OpenMP [DM98], and Pthreads [IG18]. However, these systems are not concerned with memory but with realizing parallelism, task creation, management, scheduling, and synchronization. The actual means to realize the shared memory is typically—but not necessarily—realized in hardware by coherence protocols. However, realizing this coherent memory becomes increasingly complex for hardware manufacturers as parallelism increases. As such, some researchers predict that shared memory programming will not be the model for highly parallel systems [SGG13, p. 124].

One programming paradigm that shares quite a lot of similarities with shared memory programming is the PGAS model. Similar to shared memory, it supposes all control flows have a common view of memory. However, the access semantics in this memory is different. In the following, we will discuss this model in more detail.

3.1.3 Partitioned Global Address Space

The PGAS programming model is a programming model used in a class of parallel programming languages predominantly used for HPC applications. The programming model realized by PGAS languages is mainly concerned with the presentation of the memory of a computing system (be it a single computing node or a compute cluster) to the application.

Informally, the principle idea of PGAS programming languages is to represent memory in a single global shared address space\(^3\), quite similar to the shared memory model. However, this address space is partitioned and assigned to processing elements or thread groups. Typically, the memory partitions are made along physical boundaries, for example, between cluster nodes. In addition, PGAS languages typically facilitate means that allow remote memory to be accessed as if it were local, whereas remote accesses, of course, can be slower. What distinguishes PGAS languages from shared memory systems is that PGAS languages make the memory partitions visible to the programmer, thus making the programmer aware of data access costs.

While the exact origins of this programming model are elusive, there are some traces regarding the terminology. De Wael et al. [De +15] attribute the terminology to Culler et al.\(^4\).

De Wael et al., furthermore, identify the following four properties as core features of PGAS languages:

1. the language is intended for parallel programming […]
2. the language makes data access cost explicit by describing a partitioning of the global address space […]

\(^3\)Some PGAS languages also provide private memory segments in addition to the globally shared memory.

\(^4\)See [Cul+93].
3.1 Parallel Programming Models

3. the language specifies how the data is or can be distributed over the different memory partitions […]

4. the language allows for data access with the perception of shared memory

Properties 1, 2, and 4 should be clear from the informal definition above, while definition 3 requires some explanation. This item refers to a feature that is commonly found frequently in PGAS languages, distributed data structures. These distributed data structures allow specifying the mapping of the data structure elements across the PGAS system’s partitions.

When taking a second look at the PGAS properties, it is noticeable that the definition does not state how the shared address space should be realized. This is because these languages commonly realize remote memory access in their runtime system or by instrumenting remote memory accesses at compile time. The remote memory access is then typically realized through message passing.

Summarized, PGAS programming languages share many similarities with shared memory programming. They provide a single global address space where (at least for most languages) all data is globally accessible. The intention of this type of memory organization is similar to that of NUMA-aware operating systems and programming environments, as it emphasizes data locality. In contrast to NUMA-aware systems, however, the location of objects in memory is not transparent to the user. Instead, programmers are always apparent of data locality and should optimize their programs accordingly. The difference between PGAS languages to shared memory programming lies in the fact that the underlying memory system does not realize access to remote data but, instead, by the language’s runtime or compiler.

3.1.4 Conclusion

In this section, we discussed the currently most widespread programming models for parallel systems, namely message passing, shared memory programming, and PGAS. As we have established, the introduced programming models primarily differ in their conception of memory. For example, message passing implements a shared-nothing approach: the control flows have private memories and communicate exclusively by exchanging messages. On the contrary, in the shared memory programming model, all control flows share a common memory for communication. Finally, PGAS can be located in between these extremes, as it provides a global address space to all control flows, similar to shared memory. This address space, however, is partitioned, and access semantics between different partitions may vary.

However, the choice of the parallel programming model seldom is a choice by preference. Instead, it typically grows from necessity, driven by the capabilities of the underlying hardware. For example, in SMP systems where the system’s memory is shared among processors, shared memory programming is natural. On the contrary, if the system does not provide a shared
memory interface, as is the case between nodes of a cluster, message passing or PGAS languages often are the only viable option\(^5\).

While the above mainly applies to computer clusters, current architectural trends blur the lines. For example, as architectures drop cache coherence and the heterogeneity in memory access increases, message passing and PGAS become attractive alternatives even for on-chip systems. However, as noted by Lauer and Needham [LN79], it is crucial to remember that at the end of the day, message passing and shared memory programming are two sides of the same coin—these considerations apply to PGAS as well. Depending on the system, message passing can be implemented in various ways, including through shared memory. Conversely, shared memory relies on the coherence system under the hood to present a coherent view of memory. The operation of the coherence protocol can be understood as a form of message passing. These concepts differ primarily in the interfaces they provide to the user.

Finally, in modern parallel applications, the choice of programming models is not clear-cut. Instead, it is common for HPC applications to apply both shared memory computing frameworks, such as OpenMP, and message passing, typically MPI when crossing compute node boundaries. This hybrid approach allows the coherent memory interface of a single computing node to be exploited with shared memory programming. Multiple nodes can work jointly on larger problem sets and coordinate via message passing. Of course, this comes with increased program complexity as multiple programming interfaces have to be used. More recent versions of the MPI and OpenMP standards adopt this style of programming as they start to add support for shared memory or message passing, respectively.

3.2 Coherence in Parallel Computer Architectures

Having discussed how programming models perceive memory, we now focus on how parallel computer architectures realize this perception. In particular, we will discuss the realization of coherent shared memory. The discussion will thereby revolve around the question of how past and current hard- and software architectures take care to provide coherent views of memory. Therefore, we will first discuss the current state of research on cache coherence systems and their limitations. Secondly, we will discuss an alternative, more application-centered approach to coherence that has recently gained much attention. Finally, we will have a closer look into coherence mechanisms utilizing software by taking a look into the class of VSM systems.

3.2.1 Scalable Cache Coherence Systems

We have established earlier that today’s highly parallel computer architectures increasingly struggle to maintain cache coherence. Nevertheless, this programming model’s apparent simplicity and effectiveness make cache coherence appealing. So it is unsurprising that many researchers investigate designs for more scalable cache coherence systems.

\(^5\)There were attempts to provide a shared memory programming interface for compute clusters. This research is more commonly known as Virtual Shared Memory and was popular in the late 80s and early 90s. A more in-depth discussion of these systems follows later in this chapter.
As mentioned earlier in Section 2.2.1, there are two approaches to cache coherence: snooping-based and directory-based coherence protocols [HP17, pp. 379–380]. In snooping-based coherence systems, the cache controllers directly broadcast modification of cache lines to all other cache controllers. While, at first sight, this naive approach appears to be prohibitively expensive, it is, in fact, the most popular coherence mechanism for small to medium-sized SMP machines. These systems can exploit the properties of the bus interconnect to implement snooping inexpensively. A store on the bus is visible to all connected devices, meaning a store implicitly constitutes a broadcast. It is sufficient for the cache controllers to snoop for stores on the bus. This approach, however, falls short when the system does not feature shared buses, for example, when interconnect topologies become more complex.

On the other hand, directory-based systems do not require broadcasts and are, therefore, particularly suitable for system architectures that do not have a central bus system. These directory-based cache mechanisms were first proposed in the late 70s [Tan76; CF78] and later implemented in the Stanford DASH system [Len+90]. Common to these coherence systems is that they use a (logically) centralized directory that stores a list of active copies for every cache line. In the following, we refer to this as the cache line's sharer list. In addition, the directory is governed by a functional unit called directory controller.

The functioning of directory-based coherence systems can be described as follows: when a core wants to modify a cache line, its cache controller sends a modification request to the directory controller for the cache line. The directory controller looks up the entry for the address and, using the sharer list, sends an invalidation request to every other cache that currently holds a copy of the cache line. When all notified cache controllers have invalidated their copies and acknowledged the invalidation request, the modifying core is granted exclusive access to the cache line and can proceed with its modifications.

While conceptually simple, these coherence protocols provide ample design space. Implementations of these coherence mechanisms vary severely in effectiveness and scalability. Consequently, directory-based coherence mechanisms have been the subject of significant research interest. As we will see in the following, the research on directory-based coherence is mainly concerned with directory storage overheads, performance, and energy consumption.

One way to organize the directory data is as a duplicate-tag directory. These directories are implemented as fully associative lists that store tags and sharer lists of any currently cached cache lines. Therefore, the directory must be sufficiently large to comprise the cache capacities. Duplicate-tag directories, consequently, can have a significant size. Early directory-based coherence systems located their directory in the main memory [Len+90]. However, this approach makes some cache miss operations extremely expensive. For example, upgrading a cache line to the exclusive state requires a lookup in the main memory.

Therefore, many researchers propose schemes to decrease the necessary storage costs. For example, systems with a shared last-level cache can store the directory information alongside the cache lines to conserve the storage for duplicated tag information. However, this approach entails the storage of metadata for every cache line, even if it is unshared. Martin et al.
propose in [MHS12] a similar approach, and the Intel i7 stores metadata for the directory-based coherence protocol in the last-level cache [HP17, p. 404].

Alternatively, the storage sizes can be reduced with lossy encoding schemes for sharer lists at the cost of superfluous invalidation messages. Common concepts thereby include grouping cores [LL97] or bloom filters [Zeb+09] to reduce the necessary storage. Alternatively, some systems provide a fixed number of pointers that express sharers. If the number of pointers is exhausted, these systems fall back to broadcasts [Aga+88; ISO00].

The strategies above mainly rely on clever encoding schemes to reduce the size of directory entries statically. Cuesta et al. [Cue+11], instead, propose exploiting runtime knowledge to store only directory entries for those pages currently in use. To this end, the coherence mechanism classifies pages as private or shared, whereas coherence is only maintained for shared pages. The operating system is then in charge of explicitly turning the coherence mechanism on if multiple applications should access a page. Fensch and Cintra [FC08] propose a similar concept. We will refer to this concept later in Chapter 5.

While the main design focus in earlier directory-based coherence systems lays on memory consumption and performance, the energy and chip area consumption in terms of complexity increasingly gained attention more recently [Cho+11; Fer+11; Yao+15]. Organizing the directory with duplicate tags is costly regarding energy consumption, as tag lookup in fully associative structures is energy-intensive due to many comparison operations. Conversely, sparse directories [GWM92] realize the directory as a hash map, which entails significantly fewer comparison operations and thus is not as energy-intensive, however, at the risk of hash collisions. When such a collision is encountered, the already stored directory entry is invalidated prematurely. This, of course, results in a higher number of invalidations and, thus, worse performance. In order to reduce the number of collisions, sparse directories typically are overprovisioned, resulting in area consumption. The Cuckoo directory takes an alternative approach and addresses the issue of hash collisions using a Cuckoo hash [Fer+11].

Further optimizations for directory-based coherence protocols are possible by distinguishing data access patterns. For example, in [Sri+17], the authors suggest a coherence system in which applications can configure coherent memory regions. The authors claim that this allows an overall directory size reduction.

Concerning the computer architectures covered in this dissertation, looking into cache organization for tile-based architectures is also sensible. One significant advantage of the organization in tiles is that tiles are typically small enough to implement snooping-based coherence. In systems with small numbers of tiles, directory-based coherence can efficiently provide coherence between the tiles. A similar scheme was realized by the Stanford DASH system [Len+90]. This approach, however, is only sufficient as the number of tiles stays relatively small. A more recent trend in cache organizations is Non-Uniform Cache Access (NUCA) architectures [HP17, p. 371]. Motivated by similar observations as for NUMA systems, these cache architectures split the cache into multiple partitions. No two cache partitions cache the same data. The aggregated cache partitions act as a shared cache. However, access latencies to a cache partition are determined by processor proximity [KBK03]. NUCA caches are well suited
for tile-based architectures: the cache partitions are mapped to the tiles in the system, and each
partition is responsible for parts of the memory. To maintain coherence for the NUCA cache,
directory-based coherence mechanisms can be used, for example, by storing sharing information
directly into the partitions. The now-discontinued Tilera system, for example, implements such
a NUCA system. This platform implements additional NoC lanes for coherence messages to
speed up coherence traffic [Cor11]. The downside of these architectures, however, is that similar
considerations as or NUMA systems apply. Data should be cached locally to the processors
that access it and possibly migrate as the access patterns change. Realizing these strategies in
hardware, however, results in significant complexity.

3.2.2 Cache Coherence with Self-Invalidation and Self-Downgrade

One major drawback of write-invalidate coherence systems is that they have to be conservative
by design. They have to provide coherence for all cache lines, even for those where coherence
is not necessary. Besides the storage and energy consumption costs, coherence protocols
thus entail significant additional effort during runtime. For example, in a directory-based
coherence system, a processor that wants to modify a non-exclusive cache line must wait until
it gains exclusive access permissions. This involves the directory controller sending invalidation
messages to all caches that share copies and waiting until the respective cache controllers have
performed and acknowledged the invalidation. This waiting time, of course, increases with the
number of sharers of the cache line.

The self-invalidate approach, therefore, instead shifts the responsibility to maintain coherence
to the software level, thus reducing the impact of invalidation and acknowledgment messages
[LW95]. The principle idea is that threads invalidate cache lines when they anticipate that other
cores modify them. Self-invalidates thus keep the sharer lists short, which reduces the number
of invalidation messages. Points in code where invalidations can be expected are synchronizing
memory accesses (see Section 2.2.2), such as atomic test-and-set operations. Additionally, there
exist alternative approaches to detect cache lines to invalidate with compiler support [CV90] or
through the virtual memory system [KS95].

Recently, this idea has gained more traction and has been extended in several works using the
optimizations enabled by weaker consistency models [Cho+11; Kax+15; RK12; Ros+17]. These
concepts additionally introduce self-downgrades that function similar to self-invalidates. The self-
downgrade operation thereby explicitly flushes modifications from the caches. Conceptually,
the self-invalidation and self-downgrade pair have similar semantics to the acquire and release
fences in the release consistency model.

Overall the performance in systems relying on self-invalidate/self-downgrade depends on
the number of invalidations and downgrades it causes to provide coherence. Hence, there is
an incentive to avoid invalidations and downgrades as much as possible. Therefore, many
self-invalidate/self-downgrade systems employ heuristics to determine the necessity of these

---

6In the MOESI and related coherence protocols, a cache line flush involves a downgrade from the exclusive state to
the shared state. The term self-downgrade derives from this operation.
3 State of the Art

operations. For example, one approach classifies shared and unshared data depending on whether the memory page the cache line resides on is shared [RK12]. Other systems use special locking instructions that filter those memory operations performed within the scope of critical sections [Ros+17].

In contrast to traditional coherence systems, where only one core can modify a cache line at a time, self-invalidate/self-downgrade systems require special handling for false sharing. In self-invalidate/self-downgrade systems, multiple modified copies of the same cache line can exist. Data would be lost if these were to be written back without coordination. Self-invalidate/self-downgrade systems commonly address this issue by tracking dirty bits at word granularity to facilitate a selective writeback of modified words [Cho+11]. This approach, of course, entails additional storage costs.

3.2.3 Virtual Shared Memory Systems

As we have established above, it is common in computer cluster environments to resort to message passing due to the absence of a shared memory interface. Understandably, the interest was considerable when Li proposed the idea of VSM with the IVY system [Li86; Li88; LH89]. IVY implements a shared memory abstraction for computer clusters realized in software. Therefore, the IVY system realized a concept quite similar to virtual memory: the address space of this system would comprise the memory of all nodes in the cluster. When a node accesses a physically remote memory page, it triggers a page fault that creates a local copy of this page. Multiple nodes could hold read-only copies of a single memory page, whereas only a single node could have write access to a page. With this multiple-reader/single-writer system, IVY provided sequential consistency over the shared address space. Conceptually, IVY realized a coherent cache with cache lines of the page size, a concept that would subsequently be known as VSM.

While the IVY system did not match the performance of hardware cache coherence and message passing systems, Li’s work initiated several similar research projects. Covering this entire body of research would exceed the scope of this dissertation and exhaustive and extensive surveys have already been performed [NL91; Esk96; CKK95]. The extensive overview of Eskicioglu [Esk96] on VSM systems alone lists over 400 entries up to the year 1996. Instead, in the following, we will focus on some key insights and strategies from the domain of VSM systems.

VSM systems like IVY commonly provide coherence at a relatively coarse granularity (i.e., at the granularity of memory pages). As a result, the performance of these systems suffers. Hardware-based shared memory systems experience performance losses with effects such as thrashing [Den68], where a cache line ping-pongs between the caches of two or more cores; these effects are devastating with large cache lines. Furthermore, using cache lines of the size of pages increases the likelihood of false sharing, thus adding further to cache thrashing. Mirage [FP89] addressed these issues by introducing time slices. Access to shared pages would be
3.2 Coherence in Parallel Computer Architectures

guaranteed for a time slice, thus allowing nodes to perform more operations until they lost access permissions.

However, it would be weak consistency models that boosted VSM performance significantly. While weak consistency models provide significant performance advantages in systems with small cache lines, these models shine as cache line sizes increase. As weak consistency opens the door for many optimizations in the coherence protocols, frequent invalidation messages and, thus, network traffic and copy operations were avoided. In consequence, write-invalidate VSM systems soon came out of favor. Munin, for example, was the first VSM system to use release consistency [CBZ91; Car95; BCZ90]. In order to implement release consistency, Munin maps the shared memory pages read-only. The attempt to modify read-only cache lines triggers a page fault, which records the modified page in a modified list and maps it as read-write. Munin drains the modified list on memory fences by writing the modified page contents back. Keleher, Cox, and Zwaenepoel [KCZ92] built on this concept by making the release operation lazy. As acquire-release pairs constitute a consumer-producer relationship, it is sufficient to delay the writeback of modifications until the data is required, that is, until it is acquired. This concept was later realized for the TreadMarks system [Kel+94]. The entry consistency model [BZS93], as implemented in Midway, refines the release consistency further. Here, modifications are passed directly from processors leaving critical sections to those entering them (hence the name). This approach, however, requires that synchronization variables are explicitly associated with the memory objects guarded by them. Scope consistency [ISL98] follows a similar approach. This model uses user-defined scopes to maintain consistency to provide better programmability than entry consistency.

While weaker consistency models helped a lot to increase performance in VSM systems, their coarse granular cache lines meant that these systems still suffered significantly from (false) sharing of pages. Consequently, researchers proposed techniques to enable multiple writer access to shared pages. Therefore, the proposed systems typically maintain two local copies of their cache lines, a working copy and a golden copy. Modifications are performed solely on the working copy. In order to write changes back, the VSM determines a Difference Set (diff) between the working and golden copy and publishes this diff. Provided that diffs of multiple nodes do not non-overlap, the VSM system can merge modifications of multiple nodes without inconsistencies. The latter condition is guaranteed as long as applications are data-race-free. This concept appears to have been independently invented by Bennett, Carter, and Zwaenepoel [BCZ90] and Giloi et al. [Gil+91], while the former use run-length-encoding. In contrast, the latter use binary exclusive or operations. The TreadMarks system later adopted a similar strategy [Kel+94].

Besides VSM systems that base their coherence system upon demand paging, researchers have explored several other system designs, as well. One research avenue is the use of compilers or programming languages. For example, in [ZSB94], the authors mitigate page faults and diff calculation overhead by tracking modifications to cache lines of the hardware caches. This approach, therefore, relies on a compiler extension that adds special instructions at those stores that modify shared data. Orca [BKT92], on the other hand, is a programming language for
distributed systems that lifts the matter of coherence to the granularity of objects and method invocations upon them.

Despite significant research efforts, VSM systems hardly left the academic space. The lack of success is because their performance never kept up with specially tailored message passing applications [ST17, p. 26]. One factor contributing to this was likely due to the capacity constraints of the used interconnect technologies (i.e., ethernet).

The more recent introduction of interconnects such as NoC architectures renewed interest in VSM concepts. In [Nür+14], a comprehensive overview of the possible design space for VSM systems in tile-based architectures is given. The MetalSVM system [Lan+11] is a distributed hypervisor that governs the MPSoCs tiles and provides a virtual memory interface for the hosted operating system. The hosted operating system sees the tiled architecture as a cache coherent SMP. The vNUMA system [CH09] follows a similar hypervisor approach. A conceptual extension of the vNUMA system for tile-based architectures is described in [Hei09]. The vNUMA system furthermore provides specific means to implement atomic operations for the emulated NUMA memory. Other current hardware trends also show an impact on VSM research. For example, Kaxiras et al. [Kax+15] use remote direct memory accesses as a backbone for their VSM mechanism. Their system is based upon a self-invalidate/self-downgrade mechanism.

3.2.4 Conclusion

A key component for functional shared memory systems is cache coherence. In the above, we visited different mechanisms that realize coherence. Thereby we focused primarily on highly parallel embedded systems.

We thereby discussed the traditional approach to cache coherence: hardware-based coherence protocols. Snooping-based coherence mechanisms are barely suitable for more complex interconnects and memory architectures, and these architectures use directory-based coherence systems. In theory, directory-based coherence systems scale well. However, we have seen that it becomes increasingly difficult to store the sharing information for cache lines in systems with increasing core counts and memory. Current research acknowledges this issue. However, the proposed coherence mechanisms do not address the underlying scalability problem. Instead, they shift it towards increased runtime overheads, power, or area consumption. However, scalability is one of many issues with these coherence protocols. In addition, the proposed directory-based coherence systems grow in logical complexity. As straightforward as the described coherence systems appear, these systems require additional state logic to handle concurrent accesses to cache lines [Cho+11]. The resulting increase in the state space significantly adds to the verification costs for directory-based systems [ASL03]. Consequently, the recent trend to forgo global coherence mechanisms altogether, as tile-based architectures do, is understandable.

Those scalability considerations motivate the research for the self-invalidate/self-downgrade systems as an alternative to hardware coherence mechanisms. In these systems, the necessity for a centralized directory is eliminated as cores independently invalidate and flush their cache entries by taking advantage of weak consistency models. These systems, however, require
special care to resolve inconsistency errors arising from false sharing. A simple solution would be allocating distinct data objects to separate cache lines. Doing so, however, would significantly increase memory consumption and hamper cache effectiveness. So instead, self-invalidate/self-downgrade systems require the caches to track modifications in the cache line per word. This approach, however, also means additional storage overhead for every cache line, as the cache must track dirty bits for every word of a cache line. Consequently, similarly to directory-based approaches, self-invalidate/self-downgrade systems impose further storage overheads.

Another issue with self-invalidate/self-downgrade coherence schemes is their conservative approach to ensuring coherence, making them likely to issue unnecessary invalidations and flushes. This issue is addressed with different heuristics, such as specialized instructions from the programmer or compiler. Finally, context switches in self-invalidate/self-downgrade systems require special care. A thread may not continue executing the processor it ran on when interrupted. As a consequence, it may see outdated cache entries. In order to prevent inconsistencies, the operating system, therefore, must flush and invalidate any cache entries upon context switches. However, this significantly limits the benefits of caches and is thus costly performance-wise.

Finally, we discussed VSM systems that provide shared memory interfaces for distributed memory architectures. Historically, these systems were invented to solve a different, albeit related, issue: providing a shared memory programming interface in HPC systems. With the recent advent of MPSoCs that feature distributed memory architectures, as well, there is renewed interest in VSM systems. Different approaches to VSM systems have been discussed and evaluated over the years. In general, however, a VSM system implements a virtual address space atop the nodes of a compute cluster. Access to remote memory triggers a demand paging mechanism replicating the accessed page. In addition to that, these systems provide coherence mechanisms for the replicated pages. Furthermore, these systems typically realize weak memory consistency models to reduce overheads for data transfers and support multiple writers by determining the binary differences after modifications.

Despite the significant body of research, VSM systems did not become prevalent. The main reason is the significant overheads caused by frequently copying memory pages and creating diffs. Furthermore, programming under weaker consistency models is significantly more complex than programming under sequential consistency. Communication latencies in earlier VSM systems furthermore added their share, as these systems typically used interconnects such as ethernet. While in MPSoCs communication latencies should be significantly lower than in those early systems, hardware-based coherence is likely to perform better.

Therefore, we conclude that shared memory programming in increasingly parallel architectures is at a crossroads. While small to medium-sized architectures likely benefit from

---

7One could argue that it is sensible to invalidate cache entries upon every context switch in the face of the many side-channel attacks found for modern processors. In fact, Linux has an optional feature to turn L1 data cache invalidation upon context switches on at a per-process basis [Sin21]. However, for context switches between threads of an application, such a mechanism is barely useful. Furthermore, the widespread implementation of simultaneous multithreading where a single processor core may execute multiple control flows further defies this mechanism, as these control flows share the same cache.

37
hardware-based cache coherence systems, whether these approaches gracefully scale as core counts increase and memory architectures diversify or hamper the overall system scalability is debatable. Furthermore, stricter power and chip area budgets will likely further pressure these coherence mechanisms. We have seen that alternatives to the “traditional” coherence protocols exist. However, both introduce significant runtime overheads and self-invalidation/self-downgrade. Furthermore, require additional storage space to track dirty bits at word granularity. Nevertheless, these coherence mechanisms likely perform sufficiently well under the assumption of suitable data to memory mapping.
In the previous chapter, we looked into the approaches the current state of the art offers to realize cache coherent parallel systems. However, we also saw that the crux with all these mechanisms typically is that they scale only poorly—be it due to increased complexity, protocol overheads, or chip area and power budget consumption. So, while cache coherence is a favorable property for SMP and NUMA architectures, efficient implementations are difficult to devise. Some MPSoC architectures thus choose to forgo a coherent shared memory and thus require applications to communicate explicitly via message passing. This abstraction, however, is only ideal for some application scenarios. The basic assumption of this thesis is that, despite these complications, cache coherence systems remain valuable and desirable features. However, their principle design assumptions must be rethought if we want them to perform well.

Let us, therefore, take a closer look at current shared memory applications and systems: although these profit from shared memory, they rarely share data extensively, especially across the boundaries of NUMA-domains. Therefore, it suffices to provide coherence only when applications share dataproviding this notion of coherent memory in hardware when unused is wasteful regarding chip area. This leads us to conclude that coherence systems should instead be realized as an on-demand feature only paid for when in use. Such a coherence mechanism would be flexible enough to act when data is actually shared. Ideally, it would also only cause costs in these cases. Furthermore, this mechanism would best be realized in software, thus additionally leveraging this flexibility and adding next to no hardware resource footprint.

In this chapter, we present the design and implementation of a Software Consistency System (SCS). The SCS thereby is specially designed for the application in tile-based architectures without coherence between the tiles, just as the InvasIC architecture. We will first discuss the general approach that underlies the SCS. Then, equipped with this knowledge, we will have a closer look at the particular components of such a system and their interplay as we discuss the design of the SCS. Finally, we will closely examine the implementation details of said components and conclude with a brief résumé.

4.1 A Software Approach for Coherence in Tiled Architectures

To provide a basis for the architecture of the SCS, let us first revisit the core attributes of our target architecture. As we have seen in Section 2.1.1, we consider computing systems consisting of multiple tiles. These considerations thereby also apply to other tile-based architectures,
such as the Intel SCC research system [Cla+13] or Tilera’s TILE processor family [Cor11]. In a tile-based system, we understand a tile as a group of cores and memory. The memory of a tile is globally accessible, both from a core located on the same tile and remotely by cores residing on other tiles. In the former case, we say that the memory is local concerning the core, whereas in the latter case, the memory is said to be remote concerning the core. Remote memory accesses are facilitated by the NoC.

While the NoC thus presents all memory as equally accessible, this results in a heterogeneous memory interface. Moreover, remote accesses are generally more expensive than local accesses, as a memory access must travel across NoC switches. Overall, the NoC communication latency thereby is subject to the applied routing and switching algorithms. As such, it can vary significantly depending on the total load of the NoC. Some NoC architectures, therefore, implement multiple networks for different purposes, such as cache coherence and peripheral or memory access. The InvasIC system, on the other hand, implements only a single NoC for all transfer types.

Finally, NoC architectures, like the InvasIC platform, usually also offer guaranteed service transfers to obtain guaranteed transmission bandwidth. This is particularly interesting for applications that require a certain degree of predictability for their transfer, such as media processing applications. However, this feature is not used in the following.

The heterogeneity mentioned above of the memory interface also finds expression concerning cache coherence. Our considered architecture integrates a shared last-level cache per tile, the L2 cache, to lessen the difference between local and remote memory access times. Depending on how the application is now structured and deployed across the architecture, its perception of the memory varies. An application on a single tile perceives local and remote memory as coherent, as it is subject to the tile’s coherence protocol. However, memory is not coherent for applications spanning multiple tiles, as there is no coherence protocol for the L2 caches.

When comparing this to related architectures, a similar picture emerges, for example, for the Intel SCC [Cla+13]. NUCA architectures that we discussed in Section 3.2 are a notable exception. The TILE architectures are an example of such an approach [Cor11]. In a NUCA architecture, the entirety of the tile caches represents the last-level cache. This means that a memory access from a core on one tile can also be served by a cache on another tile. However, this also means that cache access times can vary greatly depending on the network load and the distance of the cache.

Our architecture, however, does not feature such a shared last-level cache. Multi-tile applications, therefore, perceive the memory as a non-coherent NUMA system in which tiles act as “coherence islands”. The coherence system should consider this setup. It should thus be flexible enough to provide coherence to those applications that require a coherent view of memory, for which the hardware mechanisms, however, are insufficient.

Therefore, it would be obvious to approach the issue of coherence with a similar strategy as the self-invalidate/self-downgrade systems we discussed in Section 3.2.2. Such a system could maintain coherence simply by invalidating or flushing entries from the last-level caches. This strategy is appealing as it allows applications to benefit from local parallelism in the tile’s scope.
In contrast, invalidations and flushes happen less frequently when accessing memory shared with other tiles. However, as we have seen, issues like false sharing require these systems to implement more complex caching logic.

The main issue with self-invalidate/self-downgrade systems is that they somehow have to compensate for their lack of control over “traditional” caches. Typically, they do so by extending the caches’ writeback logic, thus, however, requiring that the caches store plenty of additional metadata per cache line. To circumvent such requirements, we, therefore, instead adapt this strategy and apply software caches instead of hardware caches. In the SCS, every tile has a page cache, which is maintained by a software mechanism. What remains for the SCS is now, instead, somehow maintaining coherence for these page caches. As we have seen in Section 2.2, memory consistency is sufficient to provide a usable shared memory interface. As we will see in the following, this allows us to implement an effective and efficient shared memory abstraction.

Effectively, the SCS thus realizes the consistent shared memory interface through a VSM system. Thus, the SCS implements much more than just shared memory. It provides an abstraction for the different types of memory in the system in the sense of virtual memory. In light of the system properties of the invasive system, this memory abstracts the different consistency guarantees and properties of the respectively attached memories. It can offer a uniform memory interface for them, just as the applications require.

At the core of the SCS thus lie two components: firstly, it provides a software cache. The SCS then uses this cache to realize a memory consistency mechanism on top of it. These two components will serve as a general guiding outline for the remainder of this chapter.

4.1.1 Software Cache

The software cache is shared among the cores of a tile. This design is thus not different from the tiles’ L2 caches. Internally, a software cache consists of a cache storage that stores replicas of remote memory blocks and a cache controller that manages entries in the cache storage and handles cache operations such as cache misses. For example, when a processor accesses a datum in remote memory, the cache controller intervenes, fetches the corresponding memory block, and stores it in the cache storage. The cache storage then serves the memory access of the core with the local replica. Conversely, the software cache provides the facility to write modified entries back to memory and evict entries. In the following, we discuss how the software cache realizes these functionalities.

Caching

As we have seen, the cache controller must be able to intercept the memory accesses issued by the processors. Therefore, the SCS adopts an idea popularized with virtual memory [Fot61] systems. Virtual memory systems realize a virtual address space that extends the dimensions of the actual main memory. Thereby, these systems extend the main memory with the background memory; an address in the virtual address space can reference either a datum in the main or background memory. The virtual memory system transfers the data from the background to
the main memory and vice versa. Nowadays, processors commonly realize virtual memory via paging. One approach in these systems is to load memory from the background memory on-demand: these pages are mapped in the page table as non-present. Access to a non-present page will cause the Memory Management Unit (MMU) to trigger a page fault. In the page fault handler, the virtual memory system swaps the accessed page into the main memory.

The SCS adopts this demand-paging approach to track access to uncached pages. Figure 4.1 shows an example of this process in action: each tile stores a page cache in its TLM. The capacity of the page cache is limited to keep some memory for applications and the operating system. The virtual address space is then set up in the following way: the TLM is mapped present and read-writeable, whereas remote memory addresses are mapped as non-present. Thus accesses to the remote memory trigger page faults, which can then be handled by the SCS’s demand paging mechanism. The demand paging mechanism then creates a local replica of the accessed page, which is then stored in the local page cache and made available to the application.

![Figure 4.1: Illustration of the SCS's functionality upon access to a remote memory page. The figure shows an example system with shared memory (at the bottom) and a tile with two cores, a page cache, and its address space at the top. The page cache in the SCS stores replicas of remote memory pages. The MMU triggers a page fault upon access to remote memory pages. The handler of this page fault then replicates the page into the page cache and adjusts the mapping so that further accesses go to the newly created local replica.](image)

When the cache controller attempts to create a replica but finds the cache has reached its capacity, it must first evict an existing entry from the cache. Hence, the cache controller chooses an entry that has not been used for a while. Then it removes this entry from the cache, writes its contents back to the original page, and maps the respective address as non-present.

The caching strategy of the page cache thereby realizes a write-back policy; that is, it writes replicas only back to their original memory once they are evicted or explicitly flushed.

42
Writeback with Multiple Writer Support

Our software cache can now make remote memory pages locally accessible. A side effect is that this potentially reduces access times. However, for the cache to be practical, it must also provide the means to allow the replicas to be modified and to write them back to the original memory page later. In other words, the software cache needs a writeback mechanism.

The most apparent approach to such a mechanism is probably copying the replica’s contents directly back into the original page. However, this would require some single-writer protocol to enforce at most one write-modifiable replica of each memory page. Otherwise, multiple tiles could concurrently modify the same page and thus cause their modifications to be lost at writeback.

Figure 4.2: Illustration of the mechanism for multiple writers per page. Displayed in the figure are three tiles. The center tile owns the green page 1. The left and the right tiles hold primary and golden copies of this page. The copies in the example have non-overlapping modifications shown as hatched areas in red. When writing the modifications back, both tiles determine a diff between primary and golden copy. Both diffs are then merged into the original page. As the modifications are non-overlapping, the center page now contains both modifications. The blue page 2 furthermore illustrates the copy-on-write mechanism that creates golden copies on-demand.

In Section 3.2.3, we have seen that some VSM systems implement precisely such an approach. These systems, however, suffered from poor performance due to effects like thrashing and false sharing. Therefore, the SCS adopts an idea from [Gil+91] to grant write access to a page to multiple processors, or in the case of the SCS, multiple tiles at a time. The principle idea of this approach is illustrated in Figure 4.2: instead of copying the entire page contents of the replica back to the original page, which would overwrite earlier modifications, this strategy writes back the modifications selectively. Therefore, the caching mechanism updates the original page only with exactly those data words that have been modified in the local replica—it merges
its modifications into the original copy. If no two tiles have overlapping modified segments, merge conflicts can be ruled out; thus, this strategy allows multiple tiles to modify their replicas simultaneously. Data-race-free applications thereby fulfill the necessary precondition of only performing non-overlapping modifications.

In order to realize the selective writeback mechanism, the SCS stores two copies per replica: a primary copy on which the cores operate and a golden copy. The latter reflects the state of the page since the replica was created or written back for the last time. Upon writeback, the SCS can now determine the diff, between primary and golden copy. To determine the diff, we can now make use of the fact that the Exclusive Or (XOR) function is self-inverse: we can determine the diff by performing word-wise XOR operations between the primary copy and the golden copy and apply this diff then to the original copy, again by word-wise XOR. After the changes have been written back, the writeback operation also updates the golden copy to the current status of the primary copy. Thereby it can be reused for the next diff calculation.

While the diff calculations in this strategy cause runtime overheads, these costs are compensated for by the fact that multiple tiles may write-share pages. Thus, the strategy increases the overall concurrency in the system and mitigates page thrashing.

4.1.2 Maintaining Consistency

Functionally our software cache is now comparable to a last-level cache: it can create local replicas of remote memory and provides a mechanism to write back modifications of these replicas. However, it still has one shortcoming: it is not coherent. In the following, we will show how the SCS addresses this issue.

The simplest solution to maintain coherence between cached replicas is probably a writeinvalidate protocol, similar to IVY [Li86]. However, as we have seen in Section 2.2, the consistency guarantees of that strategy are far stricter than most applications require. Even worse, overly strict memory models commonly lead to poor performance.

However, as we also established earlier in Section 2.2, programs do not necessarily require a strict ordering for their memory accesses at all times. Instead, and this is what weaker memory models typically exploit, it suffices when the accesses only appear as if they were in a specific order. Typically, this ordering thereby is sequential consistency. What matters for applications is that they see accesses to their shared data in that ordering. Weak memory models, therefore, provide the applications with fence instructions to annotate the required memory ordering.

Interestingly, a side effect of maintaining this ordering is that the applications also observe their shared data coherently. Consequently, it suffices for a shared memory system to define and implement a memory model under which the memory appears consistent. As a result of this memory model, applications will perceive the memory also as coherent.

Coming from this line of thought, the SCS realizes the entry consistency model [BZS93]. Similar to the release consistency model, this model uses two memory fences, an acquire and a release fence. Their semantics are thereby similar to their respective counterparts in release consistency. However, the main difference between entry and release consistency is that the
4.2 Design and Architecture of the Software Consistency System

Fences work selectively under entry consistency: they operate only for memory segments instead of the whole memory. Applications can thus enforce memory access ordering for specific data objects. Furthermore, release consistency can be emulated with entry consistency by acquiring/releasing the whole memory.

```c
int data[256];
acquire(data, sizeof(data));  // 1
data[24] = factor * data[24];  // 2
release(data, sizeof(data));   // 3
```

Figure 4.3: Visualization of how the SCS ensures consistency for applications. The left-hand side shows the outline of a typical application, the right-hand side the resulting operations on memory. Applications following the protocol of entry consistency first have to acquire shared memory that they want to read with a call to an acquire() fence. This removes any possibly cached copies of this data from cache. Upon next reference, the latest version of the data is fetched into the page cache. Finally, modifications are explicitly written back by release() fence.

To provide a means to maintain consistency, the SCS, therefore, provides acquire and release memory fences with entry consistency semantics. The acquire fence ensures that modifications in the requested remote memory ranges become locally visible. In contrast, the release fence ensures that local modifications to the target range become visible for the other tiles. In the simplest case, this can be facilitated by invalidating and flushing entries from the page cache. Figure 4.3 shows an excerpt of code using these and their effect on the page cache.

4.2 Design and Architecture of the Software Consistency System

Based on the approach for the SCS, we can now proceed to define the design and architecture of this system. We will, therefore, first discuss the general components that constitute the system. Then we look closely at how these components interact to realize the caching mechanism and maintain consistency.

In the following, we suppose a system with a single application, as this assumption greatly simplifies the discussion. However, the principles in this section can be extended for setups
with multiple applications by replicating the described core data structures, such as the page caches. As such, the described approach also remains applicable to these systems.

4.2.1 Components of the Software Consistency System

When we take a closer look at the parts of the SCS, as presented in Figure 4.4, we see that the software consistency system shares many principles with hardware-based coherence systems. Therefore, we use terminology borrowed from directory-based coherence mechanisms where appropriate.

These principle components of the SCS architecture are:

**Page Cache** The per-tile cache contains replicas of remote memory pages and associated metadata. The page cache's data is stored in the private memory segments of the respective tiles’ TLM.

**Cache Controller** The cache controller manages the contents of the page cache. It does so by intercepting and interpreting the processor’s memory accesses. In addition, the cache controller takes care to realize the memory fences. It does so by flushing and invalidating page cache entries.

**Page Directory** The directory is the (logically) centralized lookup table that stores metadata for every memory page. The most prominent part of this metadata is the list of tiles that currently hold replicas of a page.

**Directory Controller** The directory controller has two tasks in the SCS: firstly, it manages the local memory that other tiles can cache. Therefore, it processes the requests of cache controllers. Secondly, it manages the directory’s contents and thus tracks the replicas of each memory page in its domain of responsibility.

As we can see from above, the SCS works with two components: the cache controllers and the directory controllers. The directory controllers thereby act as management units for the memory. Each tile in the system where physical memory is attached has an instance of the
4.2 Design and Architecture of the Software Consistency System

directory controller and an associated page directory. This directory controller is responsible for said memory. On the other side are the cache controllers and their page caches. Every tile with access to the shared memory has its page cache and a cache controller instance managing it. The cores of a tile thereby share both units. To illustrate this, assume two cores, core 1 and core 2, on the same tile. When core 1 causes a page to be replicated into the page cache, the replica also becomes accessible for core 2.

When an application now accesses remote memory, the cache controller on the tile it is running intercepts the access and interprets it on its behalf. The cache controller first identifies the tile on which this page resides. Then it sends a request for the page to the corresponding directory controller, which finally responds with the replica.

4.2.2 Caching Mechanism

The main functionalities of the caching mechanism consist of replicating remote memory pages when applications access them and writing back the modified contents of the replicas. As cache space is a limited resource, the caching mechanism must also manage the cache’s contents. This management task also involves making room for new replicas by evicting old and stale page cache entries.

![State Chart of Memory Page Life Cycle](image)

**Figure 4.5: State chart depicting the live cycle of a memory page from the cache controllers perspective.**

The interplay of these functionalities can be best described with the example of a memory page. From the cache controller’s point of view, a memory page managed by the SCS passes through the states shown in the state chart in Figure 4.5 during its lifetime.

Memory pages start out as *uncached*—the SCS maps these memory pages non-present so that the caching mechanism is notified about accesses to them. Access to uncached pages causes the caching mechanism to create a replica and place it into the cache. The page becomes *cached and unmodified*. When a replica is modified, it is marked as dirty, the page goes into the *cached and dirty* state. This dirty mark is removed once the caching mechanism writes the modifications back. Writebacks, for example, happen by explicit writeback operations or as the consequence of a release fence. Replicas remain in the cache until they are invalidated,
either due to the cache management evicting them to make room for new replicas or by explicit
operations, such as an invalidate, flush, or acquire fence.

In SCS, the cache controller realizes these transitions and thus the whole caching mechanism.
The cache controller, therefore, interacts with the respective directory controllers. For example,
for creating a replica of a memory page, the cache controller first determines the directory
controller responsible for the page. Then it sends this directory controller a request for a replica.
The directory controller then records that the requesting tile holds a replica and sends the copy.

4.2.3 Consistency Mechanism

As we have seen above, the SCS realizes consistency through the release and acquire fences.
Similar to the cache operations, these fences are also realized by the cache controller.

As the page cache implements a write-back policy, modifications on the cached replicas will
only become globally visible when they are either evicted, flushed, or explicitly written back
from the page cache. Consequently, the release fence, therefore, can be realized by performing
a writeback and thus updating the original page with the modifications of a replica. On the
other hand, for the acquire fence, the cache controller has to ensure that any reads following
the fence do not see a stale entry in the page cache. Consequently, the cache controller has
to invalidate the cache entry so that any memory accesses following the fence result in cache
misses and, thus, new replicas.

In principle, the cache controller applies exactly these strategies to realize the acquire and
release fences: an acquire fence invalidates the memory region from the cache. In contrast,
a release fence writes back the corresponding cache contents. Later, in the implementation
section, we will take a closer look at how the cache controller takes advantage of the properties
of the weak memory consistency model to avoid the writeback and invalidation operations.

With our discussion of the consistency mechanism, we now reach the end of the system
design. In the following, we will now turn to the specific implementation details of the SCS
that we have described above.

4.3 Implementation of the Software Consistency System

In the design chapter, we learned that the SCS, conceptually, consists of two active components—
the cache controller and the directory controller. In the design, the cache controller manages
the local cache, whereas the directory controller manages memory.

In the implementation, more functional parts make up these components. Figure 4.6 presents
a refined view of the system's internals: this figure extends the architectural overview we
encountered earlier with additional parts upon which the implementation builds.

The following will discuss their purpose and how these components interact. We will also
revisit the components we discussed in the previous section. After that, we continue similarly
to the preceding section, with a discussion of the operations for the caching mechanism and
then with the consistency mechanism.
4.3 Implementation of the Software Consistency System

Figure 4.6: Refined overview of the components of the SCS as they are present in the implementation. The cache controller communicates with the directory controller via an RPC-based communication layer. The API is shown above the cache controller.

Page Directory and Directory Controller

The directory controller is the unit of the SCS that controls access to the individual units of memory installed in the architecture. The SCS, therefore, provides a directory controller per physical memory component. For its management task, the directory controller, on the one side, tracks the sharing and versioning information for the memory pages for which it is responsible. It stores this and further metadata in the page directory. This data structure is realized as a hash map that maps page addresses to their respective metadata entries. On the other side, this unit realizes the cache operations on the actual memory pages on behalf of the cache controllers.

Whenever the cache controller requests a memory page or performs an operation on a memory page, the directory controller responds to this request accordingly. To mitigate memory inconsistencies, the directory controller furthermore takes care to synchronize the operations on memory pages against each other.

Page Table Modifications

Whenever the SCS creates a replica or evicts a replica, it has to modify the page table. In the former case, it maps the remote address to the replica. In the latter case, it unmaps the evicted page. Evicting a page thereby commonly also involves a writeback operation. To prevent modifications from getting lost, the cache controller must ensure that all cores perceive the page as non-present so that no core can accidentally modify the page. This, however, requires care. As the process of translating a logical to a physical address is quite expensive, MMUs
commonly implement a Translation Lookaside Buffer (TLB). The TLB is a cache that speeds up address translations. Therefore, when a page mapping changes, one must take care to invalidate the corresponding TLB entry, as well. If the TLB otherwise still has the translation cached, the mapping remains accessible for the associated core.

When multiple cores share a page table, the corresponding TLB entry must be flushed for all of them. This operation is commonly referred to as a TLB shootdown. In the SCS, the TLB shootdown sends an Inter-Processor Interrupt (IPI) to all cores that share a page table to signal them that they should flush the TLB entry for the page. Finally, the TLB shootdown waits for the flush operations to complete on all cores. Then after the shootdown, all cores share the same memory view concerning the flushed page.

In the SCS, mappings either transition from non-present to either read-only or read-write when a read or a write operation happens, from read-only to read-write, when a read-only page is written to, or from read-only or read-write to non-present. The latter case happens when a page is evicted or invalidated. It is only the latter case where the new page state entails fewer access permissions (or a completely different mapping). It is thus this case where the SCS has to perform a TLB shootdown to prevent possible inconsistencies.

On the LEON3 platform, the TLB flush operation, however, is pretty expensive. While the SPARC v8 specification specifies flushes for single TLB entries, the Frontgrade Gaisler implementation realizes these as flushes of the whole TLB [AB23, Sec. 92.10.7]. Thus, whenever the operating system requests a flush for a TLB entry, the cache flushes the whole TLB. To make matters worse, the implementation of this flush operation also unconditionally flushes the whole data and instruction caches. This means that whenever the SCS maps a page as non-present, the whole TLBs, data, and instruction caches are flushed for every core on a tile.

Finally, while implementing the SCS we found a possible race condition in the MMU implementation. While the SPARC Architecture Manual states that dirty and accessed bits in the page table must be updated atomically [SPA92, p. 254], we experienced cases that lead us to suspect that the MMU does not modify these bits atomically. This bug manifested in an error where pages mapped as non-present remained present. The error happened when a write to a non-dirty page happened concurrently with the page being evicted. The problem is that the dirty, accessed, and present bits are stored in the same data word. We assume that the erroneous flow of actions thereby is as follows: the SCS attempts to map a page as non-present. In the meantime, another CPU tries to write to the page. This write access causes the MMU to read the flags from the table and adjust the dirty bit. In the meantime, the SCS maps the page with an atomic operation as non-present. Then the MMU writes the modified flags back and thus sets the page as present again.

The problem could be observed several times in this constellation. As a workaround, the SCS, therefore, implements page map operations in the form of a compare-and-swap loop: it performs the corresponding operation and then tests whether the page table entry is as expected. If this is not the case, it repeats the operation.
Remote Procedure Call System

The RPC system realizes the communication medium which facilitates the interaction between the cache controllers and the directory controllers.

OctoPOS provides multiple facilities for inter-tile communication. Firstly, the operating system kernel provides a mechanism to send i-lets across the NoC. In addition, these i-lets can be accompanied by a DMA. These i-lets have a maximum of two parameters and no return value. One direct consequence of this design is that significant effort is necessary to invoke functions with more than two parameters or functions with a return value. Applications, therefore, typically use multiple i-lets that are sent back and forth in order to transfer all arguments and return values. This induces significant overhead, as is evident in the example of an MPI implementation for OctoPOS in [Rhe+19b]. Furthermore, this also increases the overall code complexity.

Secondly, there are Sys-i-lets. These realize a more low-level communication mechanism that allows the execution of functions on other tiles as Interrupt Service Routines (ISRs). In contrast to i-lets, these Sys-i-lets support functions with arbitrary argument counts and even return values. However, they have the drawback that they execute as interrupts. This means that they can interrupt other i-let control flows and thus require additional care as i-lets have to be synchronized against them.

Due to these limitations of both communication mechanisms, the SCS instead implements a consistency RPC system using i-lets. This RPC mechanism supports arbitrary arguments and return values. This system leverages the capabilities of the SHARQ that we discussed earlier in Section 2.1.1.

Instead of relying on these mechanisms directly, the SCS therefore instead uses its own communication system. This communication system is realized as an extensible RPC mechanism. Upon invocation of a coherence RPC call, the coherence RPC system serializes the function arguments into an argument tuple. This tuple is then transferred alongside a function identifier to the receiving tile via the SHARQ. The SHARQ handler i-let then deserializes the argument tuple, identifies the corresponding function, and invokes this handler function in the form of an i-let. The invoking side receives a future object that it can then either use to explicitly wait for the completion of the remote call or register a completion i-let for execution. The future object optionally provides any return values after completion of the RPC.

The RPC mechanism thereby leverages C++ template programming. To define an RPC function, one has to define the function and specialize template functions and classes of the RPC framework. Furthermore, the serialization and deserialization of the argument tuple rely on C++ move semantics and thus involve only little copying overheads.

Synchronization and Processing of Cache Operations

The cache controller realizes the caching and consistency operations. These operations have the same structure: a single operation is responsible for precisely one memory page. When an application, for example, requests a cache writeback for the memory range from 0x101300 to
4 A Software Consistency System

0x103020, the cache controller would represent this request with three cache operations for the pages 0x101000, 0x102000, and 0x103000.

This design is motivated by the fact that cache operations for the same memory pages typically require synchronization. For example, invalidating the cache entry for the page 0x101000 in the above example while the writeback for this entry is still ongoing could lead to memory inconsistencies. Splitting cache operations at page boundaries thus enables fine-grained synchronization for cache operations. Conversely, the cache controller can execute operations for distinct memory pages in parallel.

Internally, the cache controller represents these operations with the abstract cache operation type. This type encapsulates the specific request associated with the operation, such as replica creation or flushing a cache entry. The cache controller then takes care of processing these requests.

In order to facilitate this synchronization scheme, the cache controller uses the active object design pattern [Sch+00, pp. 318–343]. This design pattern addresses the issue of performing a set of operations concurrently in a synchronized manner. The set of operations is expressed as the methods of the active object. Method invocations are therefore delegated to a separate worker thread belonging to the active object. The operation execution thus takes place in a control flow distinct from the calling side. Furthermore, the side invoking the active object’s methods receives a future object that can wait for the operation’s completion.

![Diagram of page synchronization object's mode of operation.](image)

Figure 4.7: Conceptual visualization of a page synchronization object’s mode of operation. The process on the left side has two control flows that perform release fences. When the first fence is executed ①, the page synchronization object creates two objects: a release request, which is stored into a pending requests list, and a future signalling the requests completion. As the release request is still outstanding, when the second release fence is executed ②, the page synchronization object does not create a second release request, but instead attaches the control flow to the existing future. Then the page synchronization object’s worker thread processes the release request ③. When it has completed the request, the worker thread signals the release future ④ and thus enables the waiting processes to continue ⑤.
4.3 Implementation of the Software Consistency System

The cache controller realizes this concept for the cache operations with the *page synchronization objects*, conceptually shown in Figure 4.7: a unique page synchronization object is associated with every memory page. When a cache operation for a specific page is invoked, the cache controller first checks whether a page synchronization object for this page exists. If no such object exists, it creates one and stores it so that later cache operations can synchronize on it. The cache controller then enqueues the invoked cache operation into the page synchronization object to delegate the operation for execution. The page synchronization object, in return, dequeues the request and processes it using a worker *i*-let. This mechanism guarantees the sequential execution of the cache operations concerning the same memory page. For distinct remote pages, different page synchronization objects are used, each of which with their worker *i*-lets.

When taking a closer look at the cache operations, another detail stands out. When multiple cores of a tile issue the same cache operations for a single page, it suffices to perform the operations only once. For example, if multiple cores access the same non-present page simultaneously, creating a single request for a replica creation suffices. Therefore, the page synchronization objects deduplicate operations. When a new cache operation is requested, and the same type is already outstanding, the cache controller will not enqueue the cache operation directly. Instead, it will register a future for the operation with the existing cache operation. Upon completion of the cache operation, the worker *i*-let signals all registered futures.

Page Cache

The SCS realizes the page cache with two data structures. The first of these is the *page frame pool*. This pool stores unused memory pages from the TLM. The SCS takes pages from this pool as empty page frames to create replicas. The second data structure is the *cached page list*. This data structure contains page frames currently occupied with replicas of remote memory pages, alongside their metadata and optionally a reference to the golden copies.

Whenever the cache controller creates a copy of a remote memory page, it first tries to allocate a page frame from the page frame buffer. If this allocation succeeds, it uses this page frame to fetch the page copy and then stores it in the used page list. On the other hand, if the cache controller finds the page frame pool to be empty, it instead evicts an entry from the cached page list and uses the associated page frame. Conversely, the cache controller replenishes the page frame pool when invalidating page entries from the cached page list. Therefore it first maps the address of the cached remote page as non-present and then moves the page frame to the page frame pool.

In order to further the exploitation of spatial and temporal locality, the cache controller manages the entries in the used page list with an Least Recently Used (LRU) policy. The cache controller, therefore, prioritizes unread and unwritten entries for eviction. Furthermore, to maintain the LRU property, the cache controller frequently checks the accessed and dirty bits of the cached replicas in the page table. To this end, the cache controller employs a periodically recurring service *i*-let. This *i*-let checks for each cache entry whether there have been recent
accesses to it. If this was the case for an entry, the service i-let promotes the entry to the front position in the caches list. By this strategy, cache entries without access will, over time, end at the end of the list, whereas “hottest” cache entries remain at the front. In order to evict unused cache entries, the cache controller thus can remove page entries from the tail of the list.

Cache Maintenance

Ideally, cache replacement strategies would evict those pages that will not be accessed in the near future. Thus, it would ensure that cache misses and evictions happen rarely. Unfortunately, there is no such thing as an optimal or clairvoyant cache replacement strategy. As such, at some point, most replacement strategies have weak points where they fail, and cache misses start to affect performance. The SCS, therefore, implements additional means to reduce the overheads of frequent cache evictions.

Evicting modified replicas entails that the cache controller first writes back the modifications before removing the replica from the cache. However, if the cache controller were to find an unmodified replica to evict instead, it can directly reuse it and does not have to perform a costly writeback first.

The software consistency mechanism attempts to maintain some unmodified replicas in the cache at any given time. Therefore, the system employs a services thread running on the system claim that writes back modified pages as soon as the number of unmodified pages falls below a certain threshold. This strategy is similar to the kswapd kernel thread used by Linux [Tor, Version 5.17.5, *mm/vmscan.c*, Line 4454].

While this feature should ensure cheaply evictable pages in the cache, we found in experiments that this feature was detrimental to the system’s performance. We suspect an interplay with the high costs of TLB flushes. As such, the feature was not used for further evaluation.

4.3.1 Caching Operations

As we have seen in the design description, the cache controller operations lie at the core of the caching mechanism. These procedures include operations for the creation of page cache entries, as well as invalidating and writing back cache entries. The following will shed light on how these respective operations work by illustrating how they are implemented and how they are presented to the application layer in the SCS’s API.

Note that the interface of these operations, thereby, follows a similar convention: each operation has a synchronous and asynchronous variant in the API. The asynchronous interface makes use of a future object which can be used to wait for the completion of the respective. Where possible and sensible, the cache operations operate on multiple memory ranges. Where applicable, the API additionally provides variants of these functions that operate on the whole cache, as well. For example, there are functions to invalidate parts of the page cache and functions that invalidate all cache entries.
Handling Cache Misses

The cache controller implements a cache operation that handles cache misses for the basic caching mechanism. This operation creates replicas of remote pages and stores them in the page cache. Depending on whether the application requests read or write access to the page, this operation creates either a read-only replica or a read-writeable replica. This distinction allows the SCS to reduce overheads: if a replica is never modified, no writeback operations will be necessary. So there is no need to create a golden copy. In this case, the SCS delays the creation of the golden copy until it is needed. It, therefore, implements a Copy on Write (CoW) mechanism.

![Activity diagram for the replica creation operation. The solid nodes show the operations performed by the cache controller, whereas the dashed nodes indicate operations with the help of the directory controller.](image)

The control flow of the replica creation operation is shown in Figure 4.8: the figure shows that the cache controller first allocates a page frame from the page frame pool for the operation. If no page frame is available, the cache controller instead evicts and thus frees an entry of the
cached page list, which it then uses. This page frame will store the primary copy of the remote memory page. With the free page frame, the procedure continues with sending a request for the page to the directory controller. The directory controller first records that the tile holds a copy in the requested page's sharers list. It then replies with a copy of the page via DMA to the page frame. If the replica was requested as read-writeable, the cache controller proceeds and initializes the golden copy of the page. For read-only copies, the cache controller omits this step. Finally, the cache controller adds the page frame to the cached page list, maps the remote address to the physical address of the page frame, and adjusts the access permissions to the local replica accordingly to be either read-only or read-writeable.

The caching API exposes this operation with the family of `fetch()` functions, as shown in Listing 4. In addition, as stated above, there exists a synchronous and asynchronous version of this interface, enabling applications to wait until a requested memory range is available explicitly.

```c
void fetch_page_async(void *address, int writeable, syscall_future *future);

void fetch_page(void *address, int writeable);
```

Listing 4: Syscall interface to create replicas of pages. Using the interface, a page, specified by its virtual address, can be fetched either as read-only, or as read-writeable. The system call interface provides asynchronous and blocking variants of this function.

Applications thereby can either use the `fetch()` function to load remote memory pages into their caches anticipatory, or they can rely on the SCS demand-paging mechanism. To this end, the SCS initially maps any remote memory pages as non-present. Once a non-present page is accessed, the MMU thus triggers a page fault. The handler for this page fault is provided by the SCS. This handler first checks whether the access was a read or a write and invokes the `fetch()` function accordingly.

Therefore, the SCS extends the OctoPOS kernel so that it supports context switches within interrupt contexts. Conceptually, when an interrupt occurs, the processor saves the current application state, disables interrupts, and switches to supervisor mode. For the context switch to another i-let from the page fault handler, the context switching mechanism thus must take care to reenable interrupts before performing the switch. Conversely, after switching back to the control flow in the interrupt handler, the previous processor status must be restored so that the ISR can return to the application control flow. Therefore, the page fault handler was extended to reenable interrupts and save the processor status register onto the stack. When continued later, the handler thus then disables interrupts again and restores the processor status word from the stack. The page fault handler, furthermore, has to take care to save and restore the set of floating point registers, as this set of registers typically is caller saved.
4.3 Implementation of the Software Consistency System

Granting Write Permissions for Cache Entries

As we have seen in the preceding section, the SCS distinguishes between read-only and read-writeable replicas. This distinction reduces the number of page copies the cache controller creates and maintains at runtime.

However, the type of initial memory access is not necessarily meaningful in determining whether a page will experience only read or also write accesses. The types of accesses might change over the replica's lifetime and the application's course. Consequently, the SCS provides a cache operation to promote read-only to read-writeable replicas. In this operation, the cache controller first allocates a new page to store the newly created golden copy. Then the cache controller populates this page by copying the contents of the primary copy. The page is read-only during this operation so that concurrent modifications can be ruled out. Finally, the cache controller maps the primary copy as read-writeable.

```c
void make_page_read_write_async(void *address, syscall_future *future);
void make_page_read_write(void *address);
```

Listing 5: Syscall interface to map a read-only page cache entry specified by its address as read-writeable. The system call interface provides asynchronous and blocking variants of the function.

The cache controller exposes this functionality with the `make_page_read_write()` family of interfaces presented in Listing 5. However, this interface is seldomly called from applications directly. As described above, this interface is instead invoked by a CoW mechanism. The cache controller, therefore, maps read-only copies accordingly as read-only in the page table. Hence, write accesses to these pages will result in page faults. When a write instruction triggers a page fault, the cache controller checks whether the page is cached. If this is the case, it performs the `make_page_read_write()` operation to map it read-writeable. If not, it performs the `fetch()` operation described above instead.

Writeback of Modified Cache Entries

The control flow of the writeback cache operation is shown in Figure 4.9: as the name suggests, this cache operation writes replicas back to their original memory pages. The cache controller exposes this functionality with the family of `writeback()` functions as presented in Listing 6.

For the writeback operation, the cache controller first maps the replica's address as non-present. This prevents further modifications while the writeback is in progress. The cache controller then checks whether the replica has been modified. It does so by first checking if the replica is a read-only copy. If so, the cache controller can rule out any modifications. Secondly, if the page is read-write, the cache controller checks whether the page has been marked as dirty in the page table. If the page is not dirty or a read-only copy, the replica is unmodified so that the cache controller can skip the writeback. Consequently, the cache controller maps the page as present again and signals completion.
Figure 4.9: Activity diagram of the writeback function. The solid nodes show the operations performed by the cache controller, whereas the dashed nodes indicate operations with the help of the directory controller.
4.3 Implementation of the Software Consistency System

```c
void writeback_ranges_async(void **address, size_t *size, size_t count,
syscall_future *future);
void writeback_ranges(void **address, size_t *size, size_t count);
void writeback_all_async(syscall_future *future);
void writeback_all();
```

Listing 6: Syscall interface to write back the contents of modified page cache entry. This interface can either write back one or multiple memory ranges, each specified by a start address and size of the range, or the contents of the whole cache. The system call interface provides asynchronous and blocking variants of this function.

However, if the page is modified, the cache controller has to update the original page. As described above, the cache controller, therefore, relies on diffs. Therefore, instead of directly writing back the full memory page, the cache controller creates a diff and applies this diff to the original page. This mechanism enables multiple tiles to modify replicas of the same page simultaneously without the risk of possibly overwriting the modifications of another tile. Creating and merging such a diff, however, is time-consuming. Therefore, the cache controller first attempts to perform a fast-writeback. The fast-writeback is a conditional writeback attempt without prior diff creation.

This concept is shown in Figure 4.10: suppose a tile is about to perform a writeback. Since it created its replica, there have not been any other writeback operations yet. In that case, there are no contents in the original memory page that the writeback would unwittingly overwrite. Therefore, the cache controller can directly write back the replica contents. Thus the cache controller spares the costs of creating and applying a diff.

![Figure 4.10: Visualization of the flow of events for fast- and diff-based writebacks from the perspective of a page P. Three tiles, shown from top to bottom, create copies of P and then try to writeback the page. The time in the visualization flows from left to right. The tile in the middle can perform a fast-writeback, whereas the bottom tile has to perform a diff-based writeback. Since top tile creates its copy after the writebacks, it can perform a fast-writeback, again.](image)

For the fast-writeback, directory controllers track version vectors in the metadata of their memory pages. This version vector thereby takes the form of a bitmask indicating whether a
tile may perform a fast-writeback; in other words, it indicates whether another tile has already performed a writeback. The directory controller maintains the version vectors: whenever a tile creates a replica, the directory controller marks the tile's replica as up-to-date (with respect to the original page) in the page's version vector entry. When later a tile performs any writeback to the page, the directory controller clears the version vector and thus the fast-writeback permissions for all tiles except the one that performed the writeback. Note that if the tile's bit in the version vector was already cleared, meaning that it only has permissions to perform diff-based writebacks, it remains this way.

When a tile's cache controller now performs a writeback operation, it consults the directory controller to see whether a fast-writeback is possible. The directory controller, in return, checks the version vector for the respective page. If a fast-writeback is possible, the cache controller performs a DMA transfer for the replica's contents. If, however, there were prior modifications to the original memory page by other tiles, the writeback operation falls back to the diff mechanism.

In order to reduce the number of failed fast-writeback attempts, the cache controller keeps track of whether prior fast-writeback attempts were successful. If a fast-writeback fails, future writeback operations will directly resort to diff-based writebacks.

For the diff-based writeback, the function proceeds by allocating a diff buffer. The function then creates the diff by performing word-wise XOR operations over the replica and its golden copy, thereby storing the results into the diff buffer. Additionally, the cache controller accumulates the results of the word-wise XOR with a binary OR instruction. By this, it can check whether the diff by chance was empty. If it finds an empty diff, there is no need to merge it into the far side. If, instead, it finds changes, it sends the diff to the directory controller so it can incorporate the changes.

If the replica should maintain in the cache after the writeback, the cache controller then updates the page's golden copy, overwriting it with the current contents of the cached page. If, instead, the replica is dropped from the cache after the writeback, as is the case with flush operations, the cache controller omits this step. Finally, the function frees the diff buffer and, if the page remains in the cache, clears the dirty bit and maps it as present again.

Invalidating Page Cache Entries

The cache controller exposes an interface to drop entries from the page cache with the family of invalidate() functions as presented in Listing 7.

The control flow of the operation that expresses this interface is depicted in Figure 4.11: this operation unconditionally removes page entries from the cache. The cache controller first maps the address of the cache entry as non-present. It then signals to the directory controller that the cache drops its page copy. The directory controller, in return, removes the respective tile from the page's sharers list and other metadata entries, such as the version vector we discussed.

---

1The case where the page is marked as dirty but the diff is empty can occur when, after a series of modifications, the page's contents are identical to that of the golden page.
4.3 Implementation of the Software Consistency System

```c
void invalidate_ranges_async(void **address, size_t *size, size_t count, syscall_future *future);

void invalidate_ranges(void **address, size_t *size, size_t count);

void invalidate_all_async(syscall_future *future);

void invalidate_all();
```

Listing 7: Syscall interface to invalidate contents of the cache. This interface can either invalidate one or multiple memory ranges, each specified by a start address and size of the range, or the contents of the whole cache. The system call interface provides asynchronous and blocking variants of this function.

earlier. Then the cache controller removes the page frame from the used page list and moves it into the page frame pool.

![Activity diagram of the invalidate function](image)

Figure 4.11: Activity diagram of the `invalidate` function. The solid nodes show the operations performed by the cache controller, whereas the dashed nodes indicate operations with the help of the directory controller.

Flushing Page Cache Entries

Finally, the cache controller provides operations that flush page cache entries. Again, these operations are also exposed to the application layer. The corresponding interface is shown in Listing 8.

The cache controller realizes the operation by performing a writeback operation on the requested page cache entry, followed by an invalidation operation.
void flush_range_async(void **address, size_t *size, size_t count, syscall_future *future);

void flush_range(void **address, size_t *size, size_t count);

void flush_all_async(syscall_future *future);

void flush_all();

Listing 8: Syscall interface to flush contents of the cache. This interface can either flush one or multiple memory ranges, each specified by a start address and the size of the range, or the contents of the whole cache. The system call interface provides asynchronous and blocking variants of this function.

4.3.2 Consistency Operations

When discussing the approach to consistency in the SCS earlier, we saw that the entry consistency model, in principle, relies on the acquire/release fence pair. As we have seen there, the cache operations are sufficient to implement the fences: the acquire fence can be realized by flushing the respective memory range from the cache. In contrast, we can use writeback operations for the release fence. However, flushing and writing back would make the memory model stricter than necessary.

Instead, while the SCS uses flush/writeback operations to realize the fence pair, it attempts to mitigate them when possible. In the following, we take a closer look at the strategies the SCS employs to realize the fences. We, therefore, describe the control flow and the respective API of the fences.

Acquire Fence

The cache controller exposes the acquire fences with the family of acquire() functions as shown in Listing 9.

The acquire fence thereby aims to ensure that references to the acquired regions will see the most current values. Generally, the fence achieves this by flushing the pages in the requested memory ranges from the cache. However, if the original memory page has not been modified since the replica creation, the replica’s contents correspond precisely to those of the original page. Hence, it is not necessary to flush the cache entry. Instead, an acquire only needs to flush cache entries if writebacks of other tiles have modified the corresponding memory pages.

The control flow of this operation is shown in Figure 4.12: the cache controller first notifies the directory controller about its intention to invalidate the page from the page cache and
Figure 4.12: Activity diagram of the acquire fence function. The solid nodes show the operations performed by the cache controller, whereas the dashed nodes indicate operations with the help of the directory controller.
Listing 9: Syscall interface of the acquire memory fence. This interface can either acquire one or multiple memory ranges, each specified by a start address and the size of the range, or the contents of the whole cache. The system call interface provides asynchronous and blocking variants of this function.

thereby checks or modifications to the page. If the directory controller holds a more recent page version, it acknowledges this intent and clears the tile from the sharer list in the page directory. The cache controller then proceeds and flushes the page from the page cache, as described above. However, if the cache page is up-to-date with the version noted in the directory, the directory controller signals the cache controller to keep the page in its cache.

Release Fence

The release fence is the counterpart to the acquire fence. This fence ensures that any modifications in the requested memory range become visible globally. The cache controller makes the fence available to the application layer via the `release()` interface. This interface is shown in Listing 10.

Listing 10: Syscall interface of the release memory fence. This interface operates either on one or multiple memory ranges, each specified by a start address and the size of the range, or the contents of the whole cache. The system call interface provides asynchronous and blocking variants of this function.

The cache controller realizes the release fence with a concept similar to lazy release consistency [KCZ92]: the principle idea of this strategy is that release and acquire fences constitute a producer-consumer relationship. Released modifications only need to become visible if another tile acquires them. Therefore, the cache controller can delay the writeback of the release until some other control flow requires the changes.
Figure 4.13: Activity diagram of the release fence function. The solid nodes show the operations performed by the cache controller, whereas the dashed nodes indicate operations with the help of the directory controller.
The control flow of these fences is illustrated in Figure 4.13: the cache controller first checks whether the page is modified on a release fence operation. If the page is not modified, the tile holds no changes that must be globally visible. Thus, the cache controller can skip the release, and the fence returns directly. However, if the replica is modified, the cache controller notifies the directory controller on a release fence about its intent to write back changes. The cache controller also notes in the replica’s metadata the release intent. This information allows the cache controller to skip the roundtrip to the directory controller on future release fences. The cache controller clears this information upon the next writeback of the replica.

The directory controller, in return, stores this information in an intent vector stored in the page directory. When a tile executes an acquire fence for a page for which the page directory stores one or multiple release intents, the directory controller instructs those tiles to write back their modifications. Once all tiles with release intents have written back their changes, the acquire operation continues as usual. Furthermore, the directory controller clears release intent once a tile has performed a writeback operation.

4.4 Résumé

This chapter described the design and implementation of a software consistency mechanism for non-coherence hardware architectures.

Advances in computer architectures, such as increases in parallelism and the diversification of memory technologies, lead to the increasing popularity of non-coherent systems. By abandoning coherence, computer architectures achieve better scalability. This increase in scalability, however, comes at the loss of the well-known and well-understood shared memory model. The introduced SCS proposes a software-based alternative to reestablish consistent shared memory in these systems. It does so by introducing a virtual memory abstraction that integrates the full shared memory and presents it consistently.

One obvious benefit of software-based consistency is its flexibility: it offers consistency as an on-demand feature. Users may use consistent shared memory if necessary, but they only pay the price for consistency if they use it. With hardware-based coherence mechanisms, on the other side, this is only partially true. As we have seen in Section 3.2.1, these systems have a significant footprint on logic and energy budgets. Due to the static nature of the hardware, this price is also paid when the shared memory is not used.

In Chapter 7, we will show the example of real-world application benchmarks that the SCS is a viable alternative to hardware-based coherence schemes. However, we will also see that software-based coherence is not a panacea to the coherence problem. Especially the cache line granularity of the proposed system can lead to significant runtime overheads, for example, if multiple tiles share pages extensively or if working sets exceed the cache capacity. This observation highlights that applications must be designed carefully according to the underlying caching mechanism.

A further limitation of the proposed design is that the shared memory abstraction is not a one-to-one copy of “normal” shared memory: the resulting shared memory has no support for atomic
operations. However, these instructions typically build the basis for synchronization mechanisms in shared memory applications. Some approaches address this issue. For example, Chapman and Heiser [CH09] proposes a hypervisor-based system that interprets atomic instructions. The SCS, however, treats this issue differently. Instead, it considers control flow synchronization an issue orthogonal to memory synchronization. This dissertation will discuss a possible shared memory programming system with means for synchronization in Chapter 6.

In the design for the SCS, we deliberately neglected the tiles’ last-level caches and instead introduced a software-managed page cache. The reasons for that were that the page cache allowed us to write back modified data selectively, and we thus could avoid any issues with data inconsistencies due to false sharing. However, in terms of using the available hardware capabilities, as stated in the problem statement, this approach is wasteful, as the last-level cache remains unused. Therefore, in the next chapter, we will see ways to benefit from the last-level cache while maintaining the same coherence guarantees.
In the preceding chapter, we saw the conception and implementation of the SCS, a software consistency system that presents an alternative to more traditional hardware coherence mechanisms. The system is motivated by two observations: firstly, as we have seen in Section 3.2.1, hardware coherence mechanisms progressively inhibit parallelism increases. Secondly, as is evident by the increasing prevalence of weak memory models, shared memory applications only sometimes require coherence for their shared data. In this light, relocating the coherence mechanism to software is beneficial, as it imposes the overheads inherent to maintaining coherence only when it is active and in use. However, this gain in flexibility comes at the price of having to emulate the cache operations in software.

In the following, we will explore ways to improve the performance of the SCS. As such, we identify the page cache as a worthwhile optimization target: while this unit provides the means for a consistent view of memory, it causes additional performance overheads for access to remote data. These overheads are particularly apparent when applications experience many capacity misses.

To alleviate these overheads, we present a novel strategy for the SCS that leverages the memory-sharing information that is available at runtime. This strategy is based on the idea that the coherence system only needs to maintain coherence for data that is actually shared. By leveraging sharing information, the consistency system can thus limit the memory range for which it provides consistency and, thus, the range accessed through the page cache. This approach differs from the earlier VSM research that we discussed in Section 3.2.3 insofar as this research mainly focused on improving coherence protocols to gain better performance.

The remainder of this chapter concerns the concept and the implementation of the extended consistency system. Therefore, we proceed as follows. First, we discuss a strategy to improve our software consistency mechanism to approach better performance. Next, we describe how we extend the design and implementation of the SCS to build the ASCS, a software consistency system that makes use of the last-level cache, whenever possible, to establish consistency, and benefit from the performance of the last-level cache. The chapter then concludes with a brief résumé.
5 An Adaptive Software Consistency System

5.1 An Adaptive Approach to Caching and Consistency

If we were to wish for an ideal coherence system, we would likely arrive at some system that combines the benefits of hardware and software solutions. On the one side, this coherence system would be flexible enough to require little to no dynamic and static resources when unused. Conversely, the underlying caching infrastructure would have comparable performance characteristics as hardware caches. Of course, this ideal coherence system will likely remain wishful thinking. As is typical in engineering, we must instead find a suitable trade-off between sought-after features and costs. Nonetheless, we can take such an imaginary, ideal coherence system as a starting point to derive feasible alternatives.

Following this line of thought, it pays to take a step back to reconsider the objective one is working on to see more clearly. First, let us recapitulate in simple terms the purpose of cache coherence mechanisms: coherence becomes necessary when more than one cache holds a replica of a specific area in memory so that their replicas do not get out of sync. The job of cache coherence mechanisms is precisely that. These units ensure that caches stay in sync and maintain thus a coherent view of shared memory.

However, this basic definition also implies that if two caches do not share any entries, they do not require a coherence mechanism. In other words, as we only need to maintain coherence for pages accessed by multiple tiles, we do not need to use the page cache for unshared pages. If we are capable of telling unshared from shared data at runtime, the SCS can exploit this information to switch between page cache and hardware cache adaptively. For example, the directory-based coherence mechanisms we discussed in Section 3.2.1 also use this fact. In these systems, invalidation messages are only sent to those caches holding copies of the to-be-invalidated lines.

The following presents the ASCS, an extension of the SCS that can determine the sharing information for memory pages and use this information for caching decisions. The main difference to the SCS thereby is that it uses hardware caches for unshared data and only falls back to the page cache for shared data. This approach provides several advantages: for one thing, the hardware cache does not experience the cache miss times that the page cache does. Furthermore, it reduces the active working set in the page cache. Thus, the overall memory consumption and the number of capacity misses decrease.

Therefore, the following will detail how the ASCS determines the number of sharers of a page. The following terminology will thereby be used throughout this chapter: we say that a tile has exclusive access to a page if it is the only tile that holds a replica of the page. If multiple tiles have replicas of the page, we say they have shared access. Note that the type of access does not state in what cache the tiles cache the page contents.

5.1.1 Tracking Sharing Information

How can the ASCS now determine whether a tile has exclusive access to a page or multiple tiles have shared access? In actuality, this information is already available in the respective page directories. As the directory controller already maintains a sharers list for every memory page,
the ASCS can determine how many tiles can access the page by looking into the corresponding page directory entry. This sharers list, therefore, directly reflects if a page is exclusive or shared.

From the cache controller’s perspective, the memory page’s life cycle thus results in a state machine. This state machine is visualized as a state diagram in Figure 5.1. A page’s life cycle starts with the no access state. When a tile requests access to the page, the directory controller grants exclusive access. If this tile invalidates its cached copy, the page transitions back to the no access state. If, however, another tile requests access, the page transitions to shared access. We refer to this state transition as a downgrade. This terminology is thereby borrowed from directory-based systems. Further access requests increase the number of sharers for the page. Finally, the page remains in the shared access state until only a single tile has access to it, resulting in the tile gaining exclusive access.

Figure 5.1: Overview of the different states of memory pages and their transitions, shown as a state machine. When the first tile caches a memory page, the ASCS provides exclusive access to the page, and the tile may access it through the hardware cache. However, when multiple tiles access the page, the ASCS switches to shared access. The page is then cached in the page caches only until the last sharer invalidates its copy.

Using the sharing information, the ASCS thus can trivially determine at page granularity whether data is shared. Furthermore, as this information is situated at the side of the directory controller, it is also this unit that decides whether the cache controllers use their hardware or page caches to access pages.

In the following, we will have a closer look into the cache management for shared and unshared data and, more importantly, how the strategies that the directory controller in the ASCS employs to determine the cache that is going to be used. We will now discuss two strategies to realize caching in the ASCS: exclusive caching and read-shared caching.

5.1.2 Exclusive Caching

The principle idea of the exclusive caching strategy is that the ASCS does not have to maintain consistency for exclusively held memory pages. Whenever a tile has exclusive access to a page, the cache controller can use the hardware caches for read and write access. This caching strategy is thus almost a direct realization of the state machine described in Figure 5.1. The task of the directory controller, thereby, is to handle the case where multiple tiles request access to a page.
To realize this approach, this caching strategy extends the directory controller with access permissions. The access permissions describe a given cache controller’s permissions to a specific page and are granted and revoked by the directory controller grants. The type of access permission thereby signals whether a page is accessed exclusively and, thus, whether a tile is eligible to use its hardware cache.

The exclusive caching scheme knows two types of page permissions: exclusive read-write access permissions $P^E_{R\cup W}$, and shared read-write access permissions $P^S_{R\cup W}$. Upon access to an uncached memory page, a directory controller initially grants $P^E_{R\cup W}$ permissions. The exclusive scheme asserts that only one tile at a time has $P^E_{R\cup W}$ access permissions. It thus permits only a single tile to use its hardware cache. When another tile requests access to a page with $P^E_{R\cup W}$ access permissions, the directory controller downgrades these permissions to $P^S_{R\cup W}$ access permissions, thus revoking the current exclusive page sharer’s entitlement to access the page via its hardware cache.

Figure 5.2: Overview of the page management scheme for exclusive caching, shown as a state machine.

This state machine is similar to Figure 5.1, the main difference being that there are no transitions for pages from $P^E_{R\cup W}$ access permissions to the $P^S_{R\cup W}$ access permissions. This is because destroying and recreating a page in the page cache is more expensive than keeping the page in the cache, even if only a single copy exists.

Figure 5.2 shows the implementation of the exclusive caching strategy. Initially, memory pages are in the uncached state. Once a tile reads from or writes to an uncached memory page, the page goes into the cached exclusive state. The directory controller grants $P^E_{R\cup W}$ access permissions to the tile and notes these permissions in the page’s page directory entry. The corresponding cache controller, in return, makes use of the exclusive permissions and identity-maps the page with read-write permissions. Thus, any further accesses from processing elements of this tile to the page are directly served by the backing memory or, for subsequent accesses, by the last-level cache.

When now a tile requests access to an exclusively cached page, the directory controller transitions this page to cached shared state and grants $P^S_{R\cup W}$ access permissions. The tile thus also becomes eligible to cache the page, although it must use its page cache to do so. The directory controller, therefore, first sends a downgrade notification to the tile that currently holds $P^E_{R\cup W}$ access permissions to prevent inconsistencies. This notification causes the respective cache controller to unmap the page and flush the page’s memory range from the hardware cache.
Next, the directory controller notes $P_{R|W}^S$ access permissions for the page in the page directory and finally grants these permissions to the requesting tile. The requesting cache controller, therefore, can proceed to create a replica of the page, similar to the strategy described in Section 4.3.1. Further access requests by other tiles for the page similarly result in share access permissions. The next time the tile that used to have the exclusive access permissions accesses the memory page, it will again trigger a page fault and request access permissions.

In addition to tracking access requests, the directory controller also keeps track of whenever a tile invalidates its cache entries. It does so for both hardware and page cache entries. When a page with $P_{R|W}^E$ access permission is invalidated, the directory controller clears the permissions for this page and drops the page entry from the page directory. The page transitions into *uncached*. Conversely, when a page with $P_{R|W}^S$ access permissions is invalidated, the directory controller withdraws the permissions for the tile that invalidated its copy. Note that the page never transitions to the *cached exclusive* state. This is because promoting the permissions would likely introduce overheads: when a tile already has a local replica of a memory page, promoting the access permissions would cause this replica to be flushed, and subsequent access to the page would then cause another page fault. Instead, once the last copy of the page is dropped, the page transitions into the *uncached* state.

### 5.1.3 Read-Shared Caching

The read-shared caching strategy extends the exclusive caching strategy with shared hardware cache access to pages. Thus, multiple tiles may access a single page simultaneously through their hardware caches, albeit only for reading.

The principle idea of read-sharing caching is further to refine the distinction of the types of memory access. When looking into accesses to shared memory, it is sensible to distinguish read from write accesses: read accesses will not modify page contents and consequently do not require consistency operations. Thus, when a page is only read, this page can be hardware cached by multiple tiles. Therefore, as we have done in the exclusive caching strategy, we exclude read-shared pages from the consistency mechanism.

Therefore, the read-shared caching strategy uses the $P_{R|W}^E$ and $P_{R|W}^S$ access permissions that we already know from the exclusive caching strategy. In addition to that, it introduces a third access permission type, the shared read-only permissions, denoted as $P_{R}^S$. The latter access permissions grant the ability to cache a page in the hardware cache, but only for read access. These permissions can thereby be granted to multiple tiles at the same time.

The state machine in Figure 5.3 illustrates how the ASCS leverages this concept for cache management. When the directory controller receives a request for an *uncached* page, it distinguishes whether it is a read or write access. If the tile requests write access, the directory controller grants $P_{R|W}^E$ access permissions and the page transitions into the *cached exclusive rw* state. On the other hand, if the tile requests read access, the directory controller grants $P_{R}^S$ access permissions and the page transitions to *cached shared ro*. Either way, the requesting cache controller identity maps the page with the granted access permissions so the tile can access
5 An Adaptive Software Consistency System

Figure 5.3: Overview of the page management scheme for read-shared caching, shown as a state machine.

This strategy distinguishes three types of access permissions: $P_{E}^{R}$, where a single tile has read-write permissions, $P_{R}^{S}$ where multiple tiles have read permissions, and $P_{R,W}^{S}$ where multiple tiles have read-write permissions. When granted either of the former permissions, the tiles may use their hardware caches. For the latter, they must resort to their page caches.

The remainder of this strategy, now in principle, functions similarly to the exclusive caching strategy. The main difference is the transitions from the cached shared ro state. For example, when a single tile has $P_{R}^{S}$ access permissions to a page and performs a write to said page, the directory controller promotes this tile to $P_{E}^{R, W}$ access permissions so that it can read from and write to the page. In this case the page transitions to cached exclusive rw. This transition, however, is only possible if only a single tile has access to the page. If, instead, multiple tiles have $P_{R}^{S}$ access permissions to a memory page and a tile attempts to write to the page, the directory controller instead performs a downgrade operation and transitions the page to cached shared rw. All sharers are then demoted to $P_{R,W}^{S}$ access permissions.

With the two extensions to the caching strategy explained, we can now proceed and discuss how the ASCS’s approach reflects in the system design.
5.2 An Adaptive Cache Design Incorporating Software and Hardware Caches

The adaptive consistency system we introduced in this chapter incorporates hardware caches to benefit from their performance and page caches to provide the necessary consistency. To this end, the ASCS leverages memory-sharing information that it retrieves runtime, as we have seen in the previous section. At its core, this consistency system, however, is comprised of the same building blocks as the design of the SCS that we discussed in Chapter 4.

![Diagram](image)

As can be seen in the Figure 5.4, the design of the ASCS thereby extends the SCS design in several places. Note that the API layer has been simplified for brevity reasons. What strikes the eye is that the design now distinguishes more than one cache backend at the side of the cache controller. Furthermore, the design now considers the access permissions discussed in the previous sections. The cache controller has been augmented with an access permission store, and the page directory entries additionally store access permissions.
5.2.1 Tracking of the Extended Sharing Information

In the SCS approach, the directory already tracks some pieces of information for each page. This information includes the number of sharers of the page, a version vector for fast-writeback, and whether the replicas recorded a release operation. The ASCS extends these metadata entries with access permissions. These access permissions can be either of $P_{R,W}^E$, $P_R^S$, or $P_{R,W}^S$. The directory interprets the access permissions and grants or revokes page access according to the caching strategy described in Section 5.1.

The transitions between the access permissions happen exclusively at the side of the directory controller in response to requests for replica creations, write accesses, or replica invalidations.

5.2.2 Access Permission Store

Each cache controller in the ASCS has an additional store to stow the access permissions that it currently holds. Using the information in this store, the cache controllers decide what actions are necessary upon cache operations. Although the access permissions that a tile currently holds are implicitly available from the page cache and the page table, storing them allows associating the permission entries alongside additional information. For example, the cache controller can track with this additional information when a page was last accessed.

In the following, we will overview how this concept manifests in the implementation.

5.3 Implementation of the Adaptive Software Consistency System

For the description of the ASCS implementation, we proceed in this section similarly as for the SCS in Chapter 4. We will first take a closer look at the system’s core functionalities and its cache and consistency operations. In the remainder of this section, we then have a closer look into the ASCS’s downgrade mechanism. Finally, we will discuss in more detail how the ASCS realizes the access permission management at the side of the cache controllers.

5.3.1 Caching Operations

The ASCS operates on the same principle caching operations as the SCS. However, it attempts to use the hardware instead of the page caches for remote memory access whenever possible. When requesting access to a page, the ASCS, therefore, consults the page directory to check the sharing information for pages. The cache controller chooses the suitable cache infrastructure, depending on the access permissions the page directory grants. In the following, we will describe how the access permission handling and multiple cache backends reflect in the caching operations.

Cache Miss Handling

The cache controller of the ASCS detects cache misses similarly as the SCS: it maps uncached pages and pages for which the tile lacks access permissions as non-present. Memory access to
these pages consequently causes the MMU to raise a page fault, which the cache controller then handles.

The further procedure for handling a cache miss in the ASCS is shown in the activity diagram in Figure 5.5. The cache controller first allocates a new page frame, regardless of whether the page or the hardware cache will be used. The motivation for performing the allocation first is that this allows skipping an additional round trip across the NoC, in the case that the directory controller grants $P^{S}_{RW}$ permissions. If this is the case, a page frame is already available. This operation is shown in the right path in the figure. The directory controller directly sends a replica for the requested page, following the procedure we discussed in Chapter 4.

![Activity Diagram for cache miss handling in the ASCS](image)

Figure 5.5: Activity diagram for cache miss handling in the ASCS. The solid nodes show the operations performed by the cache controller and dashed nodes indicate operations of the directory controller. Dotted notes show the page cache operations that are unchanged from the SCS.

When the directory controller, instead, grants $P^{E}_{RW}$ or $P^{S}_{R}$ access permissions, the directory controller first records the tile of the requesting cache controller and the granted permissions into the page directory. Then it signals these permissions back to the respective cache controller.
The cache controller now knows that it can use its hardware cache for access to the page and, in return, performs a rollback and frees the previously allocated page frame, as it no longer needs it. It then records the type of access permissions it gained for the page and, depending on the type of permissions, maps the page accordingly as read-only or read-write. Finally, it signals the completion of the operation.

Granting Write Permissions for Cache Entries

The SCS uses read-only mappings for page frames to initialize the golden pages on-demand via a CoW mechanism. This strategy is adopted for the ASCS for pages with $P^S_{R,W}$ access permissions.

In the ASCS, however, read-only mappings also find application for another purpose. We already established, that the ASCS distinguishes for the read-sharing strategy between $P^E_{R,W}$ and $P^S_R$ access permissions. The former access permissions signal that a single tile has exclusive access to a page. The tile thus can freely read from and write to the page’s data without a consistency mechanism. In return, the latter set of permissions can be granted to multiple tiles, allowing them to access a page through their hardware caches. The ASCS, therefore, monitors any modification attempts to the page by mapping pages with $P^S_R$ as read-only. If a tile writes to such a page, the MMU triggers a page fault. The cache controller then handles this page fault and requests $P^S_{R,W}$ access permissions for the page.

Taken together, this results in the control flow shown in Figure 5.6. If the cache controller holds $P^S_{R,W}$ access permissions, it populates the golden copy as described in Chapter 4. If, however, the tile currently has $P^S_R$ permissions, it sends a request for write permissions to the directory controller. The directory controller then first checks the sharing information for the page. Depending on whether the page is the sole sharer of the page or not, it performs one of two actions:

1. *The requesting tile is the only tile with access to the page*: the cache controller can upgrade to exclusive access permissions. Therefore, it records $P^E_{R,W}$ access permissions for the tile in the page directory. Then, it signals these access permissions change back to the cache controller. The cache controller furthermore updates its access permissions locally to $P^E_{R,W}$, maps the page as read-write, and signals the operation’s completion.

2. *Multiple tiles have read permissions for the page*: the directory controller must downgrade the page to $P^S_{R,W}$ access permissions. Consequently, it first sends a downgrade notification to all tiles holding access permissions. This causes them to invalidate their cache entries, unmap the page and drop their permissions. Then the directory controller records $P^S_{R,W}$ access permissions for the page in the page directory and signals these access permissions to the cache controller. The cache controller then, in return, allocates a free page and performs the steps to create a page replica. Finally, when the cache controller has created the page replica, it signals completion.
5.3 Implementation of the Adaptive Software Consistency System

Figure 5.6: Activity diagram for write fault handling in the ASCS. The solid nodes show the operations performed by the cache controller and dashed nodes indicate operations of the directory controller. Dotted notes show the page cache operations that are unchanged from the SCS. Write faults happen when a tile tries to write to a page with only $P_{R}^{S}$ access permissions or if the tile has $P_{R,U,W}^{S}$ but has yet to create its golden copy.

Writeback, Flush, and Invalidate Operations for Cache Entries

For pages with $P_{R,U,W}^{S}$ access permissions, the ASCS, again, employs the same strategies as the SCS to writeback, flush, and invalidate replicas. For memory pages with either $P_{R,U}^{E}$ or $P_{R}^{S}$ access permissions, the page cache instead resorts to performing the respective operations on its hardware cache.
5 An Adaptive Software Consistency System

5.3.2 Consistency Operations

The main objective of the ASCS remains to maintain a consistent view of memory. As such, it follows the example of the SCS and provides a weak memory consistency model that operates on special memory fence instructions to restore memory consistency, if necessary. As such, the ASCS uses the same strategy as the SCS for the pages with $P_{R\cup W}^S$ access permissions.

The remaining memory access permissions, however, are handled differently. Pages stored in the hardware caches, that is, pages with $P_{R\cup W}^E$ or $P_{R}^S$ access permissions, thereby do not require specialized fence instructions. Instead, the cache controllers rely on the downgrade mechanism to maintain consistency.

To explain this approach, let us first reconsider the purpose of the memory fences under the entry consistency model once more: an acquire fence for a memory object in this model ensures that any subsequent reads of this object return its most recent, globally visible state. In return, a release fence for a memory object ensures that any locally performed stores to this object become globally visible. In the ASCS, these memory objects are thereby of the granularity of memory pages. In the following, we consider the two cases in which the cache controller either has $P_{R\cup W}^E$ or $P_{R}^S$.

Exclusive Page Access Permissions

Only ever a single tile at a time in the system can have $P_{R\cup W}^E$ access permissions to a page. Furthermore, only the tile with these permissions can modify the page’s contents. When another tile tries to access the page, be it for a read or write access, the directory controller will revoke the $P_{R\cup W}^E$ access permissions and perform a downgrade. This downgrade flushes the page’s cache lines and inhibits further modifications. Any modifications that would have been written to the background storage by any release operation will thus automatically become visible globally once the downgrade has been completed.

Conversely, as stated above, the exclusive access permissions rule out that another tile may modify the page’s contents without the tile with $P_{R\cup W}^E$ access permissions being notified via the downgrade mechanism. Therefore, when a tile has exclusive access to a given page, this tile has the most recent view to memory concerning this page. Thus, no explicit acquire fences are necessary.

Shared Read-Only Access Permissions

When one or multiple tiles, instead, have $P_{R}^S$ access permissions to a memory page, the cache controller enforces that they only have read access to the page’s contents. Therefore any modification attempts to the page will cause the directory controller to invoke the downgrade mechanism. Thus, none of the tiles with $P_{R}^S$ access permissions will have any modified state of the page in their hardware caches. Therefore, no release fence is necessary.

\[^1\] If the tile does not cache the object, it obviously does not have to acquire or release it. Hence the fence can be skipped.
Conversely, as none of the tiles may modify their cached contents, they will always see the most current state of the memory concerning the page. Again, no acquire fence is necessary.

5.3.3 Downgrade Mechanism

The directory controller uses the downgrade mechanism to signal the revocation of $P^{E}_{R;W}$ or $P^{S}_{R}$ access permissions. For this mechanism, the directory controller first adjusts the access permission flag for the respective page in its page directory per the active caching strategy. Then, the directory controller invalidates the access permissions entries stored at the respective cache controllers. Therefore, it sends downgrade notifications to all tiles with access permissions.

Upon receiving a downgrade notification, each cache controller maps the page as non-present. It then flushes the cache entries for the page from its caches\(^2\). Finally, the cache controller removes the access permission entry from its permission store.

This mechanism thus has the effect that a tile that had $P^{E}_{R;W}$ or $P^{S}_{R}$ access permissions for a page will encounter a page fault upon subsequent access to the page. This page fault will then cause it to request access to the page anew. The corresponding directory controller will then respond with $P^{S}_{R;W}$ access permissions, thus causing the cache controller to resort to its page cache.

5.3.4 Access Permission Store

As we have seen in this chapter, the cache controllers in the ASCS have two different ways of organizing their cached data and the respective access permissions. Pages with $P^{S}_{R;W}$ access permissions are managed in the page caches. When a cache controller wants to test whether it has $P^{S}_{R;W}$ for a given page, it simply checks whether a corresponding entry exists in its page cache. Conversely, evicting an entry from the page cache also renounces the $P^{S}_{R;W}$ access permissions for the corresponding page.

For memory pages with $P^{E}_{R;W}$ or $P^{S}_{R}$ access permissions, however, the situation is different. If the hardware cache does not hold any cache lines for a page, this does not necessarily imply that the cache controller has no access permissions for the page. This situation arises as the hardware cache has no mechanism to notify the ASCS’s cache controller when it evicts the last cache line for a given page. Typically, the permissions are only invalidated once another tile tries to access the respective page, and the directory controller sends downgrade notifications.

As such, the cache controller may hold access permissions for pages not cached in the hardware cache. This can have an advisory effect on system performance, resulting in an increased memory footprint. On the side of the cache controller and the side of the directory controller, both units have to manage more and more permission entries.

The cache controller, therefore, applies a heuristic to limit the number of live access permissions. As we have seen in Chapter 4, the cache controller checks every so often for the cached\(^2\) For pages with $P^{S}_{R}$ permissions, we can rule out that the cache lines in question have been modified; thus, invalidating them would be sufficient. However, we can safely assume here that the hardware cache tracks a dirty bit for its cache lines and thus simply invalidates them without performing a write-back.

---

\(^2\)For pages with $P^{S}_{R}$ permissions, we can rule out that the cache lines in question have been modified; thus, invalidating them would be sufficient. However, we can safely assume here that the hardware cache tracks a dirty bit for its cache lines and thus simply invalidates them without performing a write-back.
page in the page map whether they have been accessed recently. It does so to maintain its LRU policy. In this process the ASCS additionally checks the pages with $P_{E}^{R,AW}$ or $P_{S}^{R}$ access permissions. When these pages were last accessed some time ago, the cache controller then relinquishes the access permissions for the page, flushes its caches, and signals the directory controller about the permission drop.

### 5.4 Résumé

In this chapter, we revisited the SCS and explored an alternative adaptive strategy for its caching infrastructure. This adaptive strategy thereby leverages the performance of hardware caches when possible and relies on page caches only when necessary. The principle insight underlying this approach is that application data can be distinguished as data exclusively accessed only by a single tile and data shared among multiple tiles. The SCS approach discussed in Chapter 4 eagerly uses its page cache for all remote memory accesses. While a valid strategy, this is excessive, as the consistency system, in actuality, must only maintain consistency for shared pages. Following the adaptive strategy, the consistency system can determine the sharing behavior and use this information to maintain consistency for shared data only. As such, the consistency system can adaptively choose different caching backends—hardware or page cache—as necessary. This approach has two advantages over the SCS: firstly, it relies on better-performing hardware caches, thus providing increased runtime performance. Secondly, this approach reduces the working set size the software cache has to hold. Depending on the working set, this reduces the overall number of evictions from the page cache, adversary effects of page thrashing, and the overall memory footprint of the consistency system. This idea thereby shares some similarities to [Cue+11], where the authors use a similar private/shared distinction to reduce the number of directory entries for a directory-based coherence mechanism for hardware caches.

The ASCS is an implementation of this adaptive consistency strategy. Whenever the ASCS determines data as accessed exclusively, it relies on the hardware cache. When, instead, it finds multiple tiles to access some data, it downgrades the exclusive access permissions to shared permissions and falls back to the software consistency mechanism.

The total effect of this approach, of course, is highly dependent on the application’s memory layout. From the ASCS perspective, a general rule of thumb for the memory layout of applications is to group those variables that are exclusively accessed by a single application. Thereby the programmer can reduce overall false sharing so that the ASCS can leverage the capabilities for hardware cache for more extensive parts of memory.
6  Runtime System for Shared Memory Programming

We established a shared memory abstraction with memory consistency with the software consistency mechanisms we discussed in Chapter 4 and Chapter 5. However, in addition to this shared memory abstraction, applications also need synchronization primitives to use this abstraction effectively. More precisely, these primitives must allow applications to synchronize the execution of their threads.

With the signal primitive, OctoPOS already provides an expressive mechanism for synchronization. However, not all applications are trivially adaptable to the programming model emerging from this type of synchronization primitive. Instead, in this chapter, we will therefore describe the design and implementation of a runtime system for OctoPOS that provides synchronization primitives more suitable for fork-join parallelism. The design of this runtime system is thereby geared towards the software consistency mechanisms and the overall system architecture provided by InvasIC platform.

In the remainder, we will introduce the design and implementation of this runtime system: first, we discuss the runtime system’s general design principles. Then we proceed with a more detailed description of the respective synchronization primitives that the runtime system provides and, more importantly, how these handle the matter of memory consistency.

6.1 Design of the Runtime System

To motivate the design of the runtime system, let us first reconsider the status quo of the programming interface that the InvasIC system and, in particular, OctoPOS provides. By default, the InvasIC system has only limited shared memory programming support: memory in this system appears only coherent for cores on the same tiles. Cores on different tiles do not perceive the shared memory as coherent.

A similar picture emerges for the coordination of threads: OctoPOS provides the means to coordinate threads; however, these are only meaningful within the context of the respective tiles. Instead, applications that want to share data or coordinate threads across tile boundaries have to resort to message passing.

The software consistency mechanisms introduced in this dissertation address the former of these restrictions: they provide a shared memory abstraction with memory consistency guarantees across all tiles. The runtime system introduced in this chapter complements the software consistency mechanism with an API that facilitates the means to coordinate and
synchronize the execution of threads in shared memory parallel applications across multiple tiles.

Therefore, the runtime system for our software consistency systems realizes fork-join parallelism. The main thread of an application using the runtime system initially forks into several worker threads and then waits for them to finish and join after they complete their work. During the execution of the parallel part, the number of worker threads is fixed and determined by the preset degree of parallelism. As a result, the applications create no new threads, and none of the threads terminates preliminarily. Furthermore, as the threads primarily perform computational tasks, there is no gained benefit, for example, through latency hiding, from creating more threads than computational resources in the system.

The runtime system realizes the application threads with the OctoPOS \( i \)-let primitive. In order to benefit from the available compute resources, the runtime system thereby pins each \( i \)-let to an individual core. The runtime system realizes this pinning by reserving the system’s cores into individual claims, each comprising a single core. It then maps the individual \( i \)-lets to the claims, in a one-to-one fashion. In the course of their runtime, the \( i \)-lets never migrate between claims or tiles.

For the assignment of the \( i \)-lets, the runtime system thereby implements two strategies, a tile-wise strategy and a round-robin strategy. The tile-wise strategy attempts to group as many \( i \)-lets on a tile as possible. The strategy, thereby, assigns \( i \)-lets to the same tile as long as there are free cores available. Only if it finds no more free cores for a tile, the strategy moves on to the next tile and repeats this process until no more \( i \)-lets are available. The round-robin strategy attempts to do the converse of the tile-wise strategy. It distributes \( i \)-lets across tiles as much as possible. Therefore, it assigns one \( i \)-let to one tile at a time until no more \( i \)-lets are available. Both assignment strategies thereby assume that there are at most as many \( i \)-lets as there are compute cores. That is, cores are not shared for \( i \)-let execution.

In terms of memory management, the runtime system presents the entirety of the SHM as coherent shared memory. The individual TLMs, on the other side, are used as private memories for the tiles. Dynamic allocations within the benchmark applications are thus served from the shared memory region. Similarly, other shared data, such as global variables, is mapped to a dedicated section for global data in the SHM.

Having explained the principle setup of the runtime system with the core allocations and principle memory management, we will now discuss in more detail how the runtime system realizes its synchronization primitives.

### 6.2 Synchronization Primitives

The SPLASH-3 benchmark applications rely on a subset of the ANL PARMACs macros specified in [LO85] to express their parallelism. The runtime system follows this principle and implements the primitives in this subset. These are barriers, locks, condition variables, and events.

Internally, the functionality of the respective primitives is realized with the signal synchronization primitive provided by the OctoPOS kernel. These signals allow a single control flow
to wait for the occurrence of one or multiple events. Appendix A provides a more detailed discussion of these synchronization primitives. The signals, however, come with a caveat: their implementation relies on atomic test-and-set operations, which work only locally in the tile’s TLM—a pragmatic decision stemming from the tile-based design of the InvasIC architecture. As such, these primitives cannot be used directly to signal across tile boundaries. Cross-tile synchronization instead additionally requires the use of RPCs. This must also be considered for the design of the synchronization primitives of the runtime system.

However, as we have seen in the preceding section, the runtime system supposes global variables to be accessible via the SHM. In the runtime system, the synchronization primitives, therefore, consist of two parts, as conceptually visualized in Figure 6.1. First, there is the local data. This data is used for the primitive’s actual synchronization mechanism. As the name suggests, this data resides in the private TLM area of one—or multiple, as in the case of the barrier primitive—tiles. As such, the runtime system can realize them on top of the OctoPOS synchronization mechanisms.

Secondly, for each synchronization primitive, there is a global reference. This reference points to the primitive’s local data and is used as a handle for the respective primitives API. This global reference can safely be stored in the SHM and is thus accessible by any application thread, irrespective of its tile.

In order to now use a synchronization primitive, a thread first determines where the primitive’s local data resides. After this step, it then sends an RPC to the tile containing the synchronization primitive’s local data to perform the according operation there.

Figure 6.1: Conceptual realization of a synchronization primitive in the shared memory programming runtime system. The handles to the synchronization primitives point to places in the shared memory. This place then references the actual implementation of the primitive, which is realized as one or more tile-local data structures.

Besides their functionality of coordinating the execution order of control flows, the synchronization primitives serve another vital purpose: they implicitly ensure that the threads perceive the shared memory according to the memory consistency model. Before we take a closer look
at the function and implementation of the respective synchronization primitives, we will first examine which strategy they use to implement the memory consistency guarantees.

6.2.1 Entry Consistency for the Synchronization Primitives

Besides ordering the execution of parallel flows of execution, it is also the job of synchronization primitives to provide a consistent view of memory. In the case of the software consistency mechanisms presented in this dissertation, the primitives must do so following the entry consistency model.

An example best illustrates this necessity: suppose a critical section in which shared data is read and written. Access to this section is thereby guarded by a lock. When a thread acquires the lock and thus enters the critical section, it expects to see the most current state of the variables it accesses there. For the entry consistency model, the lock operation thus must perform an acquire fence when it grants the thread access to the section. Similarly, this thread expects any modifications it performs within the critical section to become visible to all the threads once it leaves the section and unlocks the lock. Therefore, the unlock operation thus must perform a release fence just before releasing the lock. We can further generalize this example for the other synchronization primitives and derive the necessary memory fences for their operations.

As we have discussed Chapter 2, in the entry consistency model, memory accesses appear sequentially consistent if the code is free of data races and accesses to shared memory is guarded properly by memory fences. This means that if, in an application, synchronization primitives guard the accesses to shared data and the synchronization primitives perform the memory fences accordingly, the memory appears sequentially consistent.

This means we must devise a strategy so that the barriers perform the correct fences under entry consistency. Let us, therefore, suppose a synchronization primitive that guards read access to some shared data. From a memory consistency perspective, the primitive must ensure that a thread gaining access to said data sees its current state. So for this, the synchronization primitive must execute an acquire fence just before it unblocks the thread. Conversely, suppose a thread performs an operation that unblocks other threads. In that case, this operation must ensure that any modifications by this thread become visible to the unblocked threads. Therefore, this operation must perform a release fence just unblocking the other threads.

A straightforward strategy to implement this would be to perform a full acquire fence before unblocking a thread on the entrance to a critical section and full release fences before unblocking other threads upon leaving a critical section. The qualifier “full” refers to all page entries and L2 caches. As we have seen when we discussed consistency models in Section 2.2, this approach is functionally correct. However, it comes at the disadvantage that it also removes—possibly hot—cache entries unrelated to the respective critical section. For example, consider a situation where multiple threads attempt to increment a single counter variable. Applying the above

---

1In actuality, the memory fences do not necessarily have to be performed by the synchronization primitives; the applications themselves could also perform them. However, firstly, this would be inconvenient for the applications, and secondly, this would waste some optimization potential, as explained below.
6.2 Synchronization Primitives

strategy would lead to them invalidating the entirety of their caches, whereas the threads only require a consistent view of the counter variable.

Instead, the synchronization primitive interface in the runtime system takes advantage of the entry consistency model that the SCS and ASCS implement. Therefore, each primitive’s interfaces provide additional parameters to specify the memory regions where the respective operations should perform the memory fences. In addition, each primitive has a second interface that, for convenience, performs full memory fences.

There is yet another performance problem that underlies the synchronization primitives. This issue lies within the system architecture. Recall that the InvasIC architecture organizes its cores in tiles that share L2 and page caches. While the synchronization primitives coordinate the threads of the individual cores of a tile, the memory fences operate on the shared caches. Consequently, memory fences may be executed redundantly if multiple cores on the same tile perform fences for the same memory regions. This can, for example, lead to memory being invalidated from the caches multiple times and thus negatively affect the overall performance. Consider the example of a critical section guarded by a lock. When a thread leaves the critical section and grants access to the lock directly to another thread running on the same tile, writing back and invalidating the cache contents—as would be the case with the obligatory release and acquire in between the unlock and lock operations—is entirely unnecessary. This is because the modifications of the first thread are already in the cache. Since no other thread was executed in between, they are thus already visible by the second thread.

In order to reduce this redundancy, the runtime system does not directly perform the memory fences in the synchronization primitives. Instead, it keeps track of the requested memory fences and performs them only when obligatory. The synchronization primitives, therefore, have two separate lists: a list of requested acquire fences and a list of requested release fences. We will detail this strategy for the individual synchronization primitives in the following when discussing the implementation of these primitives.

6.2.2 Barrier

The barrier realizes a synchronization primitive that allows a fixed number of threads to wait for one another. Typically, it is used to wait for all threads to reach a certain point in code. The interface for this primitive is presented in Listing 11. The barrier instance has to be initialized once for a given group size, determining the total number of threads that can synchronize upon it. When now a thread invokes barrier_reach(barrier, n) on an instance barrier of the synchronization primitive, the operation blocks the thread until \( n - 1 \) other threads also have invoked barrier_read() on the same barrier instance.

Implementation

The runtime system realizes the barrier primitive as sense-reversing [HS08, pp. 399–400] barriers. The barriers furthermore follow a hierarchical design that minimizes cross-tile communication.
for synchronization. A barrier thereby consists of tile-local barrier instances that synchronize threads on the same tile and a global barrier instance that then synchronizes the tiles.

The tile-local barrier effectively consists of a counter variable and a barrier queue. When arriving at the barrier, the threads atomically decrement the counter variable. If the resulting value is greater than zero, the thread must wait for other threads to arrive. The thread, therefore, enqueues itself into the barrier queue and blocks. Conversely, if the counter is zero, the thread was the last to reach the tile-local barrier. All threads on the tile have reached the local barrier, and the thread must check whether the threads on other tiles have reached their local barriers. Therefore, the thread signals the global barrier and waits for the other tiles to reach the global barrier.

The global barrier functions by sending RPCs between tiles. It implements the butterfly barrier algorithm to reduce the number of sent messages [Bro86]. The conceptual idea of this barrier algorithm is shown in Figure 6.2. In this algorithm, the tiles notify one another pair-wise when they reach the barrier. The algorithm performs this notification step thereby over multiple rounds—more precisely \( r = \log_2(n) \) rounds, where \( n \) is the number of threads—whereby the notification partners vary in every step. Thus, this algorithm mitigates the necessity of a single centralized synchronization point while keeping the number of required notifications low.

Once a thread was notified in the last round of the tile-global barrier, it proceeds by inverting the local sense variable to reinitialize the barrier. Then it traverses the barrier queue and signals the locally waiting threads.

Memory Consistency

Concerning memory consistency, barrier synchronization has to ensure that the modifications performed by the synchronizing threads become visible to one another. More precisely, they must observe the new state once they continue after the barrier. For entry consistency, this means that after the control flows have reached the barrier, they must first perform a release fence. Once all control flows have released their modifications, and before continuing their execution, they must then perform an acquire fence.
6.2 Synchronization Primitives

As threads on the same tile share the last-level and page caches, it suffices when exactly one thread per tile performs these memory fences. More precisely, it must be the last thread to reach the tile-local barrier; otherwise, some modifications may remain unreleased. In addition, this thread must account for the memory ranges requested by the individual threads that might deviate from each other. The flow of operations, therefore, is as follows. When a thread reaches the barrier, it first enqueues the memory regions that it wants the acquire and release fences to be performed for into the barrier’s requested acquire and requested release lists. After this, it proceeds as described above and blocks at the barrier. When the last thread reaches the tile-local barrier, it traverses the requested release list and performs a release fence for each entry. Then it proceeds to notify the global barrier as described above. Once all tiles have reached the global barrier and the tile is unblocked, one thread per tile first traverses the requested acquire list and performs acquire fences for each entry. After this, the thread proceeds to signal the local waiting control flows.

6.2.3 Lock

The lock primitive is a means to realize mutual exclusion, such as guarding critical sections. The interface for this primitive is shown in Listing 12. A thread can request exclusive access to a lock instance via lock_lock(lock). When it finds the lock occupied, the thread blocks until the lock becomes free. When the thread successfully acquires the lock, the lock guarantees exclusive access until the thread releases the lock via lock_unlock(lock).

Implementation

Similar to the barrier, the lock follows a hierarchical design. Each lock consists of multiple tile-local lock instances and one global lock instance. This design, thereby, on the one hand, reduces the number of RPC calls involved with performing the lock_lock() and lock_unlock() operations. On the other hand, it allows the reduction of the overall necessary consistency.
6 Runtime System for Shared Memory Programming

void lock_init(lock_t *lock);

void lock_lock(lock_t *lock);

void lock_unlock(lock_t *lock);

void lock_lock(lock_t *lock, void **addr, size_t *size, size_t count);

void lock_unlock(lock_t *lock, void **addr, size_t *size, size_t count);

Listing 12: API of the family of lock functions. The listing shows two types of lock functions: the first operations implement release consistency semantics for the operation and flush/invalidate the whole caches. The second group implements entry consistency semantics. Therefore, they have additional parameters that specify the memory ranges so that the operation can enforce consistency selectively.

operations. The design achieves both goals by trying to keep the execution of critical sections local to tiles as long as possible. Thus, the threads can take advantage of the fact that the accessed memory is likely to be already cached. Furthermore, if two or more cores on the same tile attempt to access the same critical section, the number of necessary acquire and release fences can be reduced, as, from the tile’s perspective, memory is always consistent.

Conceptually, synchronization at the lock primitive proceeds as follows: a thread performing lock_lock() first attempts to lock the local lock instance. When it finds this local lock occupied, the tile (more precisely, another thread on the tile) either already has access to the global lock or another thread of the same tile has requested access to the global lock instance. Consequently, the thread blocks and waits for access to the local lock. If the thread gains access to the local lock instead, the tile still needs access to the global lock instance. The thread, therefore, sends an RPC to the global lock’s home tile. There, it attempts to acquire the global lock instance, blocking if the global lock is currently occupied. Once the thread has gained access to the global lock, it proceeds to the critical section.

When a thread releases a lock instance with lock_unlock(), it first checks whether other local threads have requested access to the lock. If this is the case, it unblocks one of them. The unblocked thread then directly gains access to the lock. If no other local threads are waiting, the thread instead sends an RPC to the global lock instance and unblocks this lock so that other tiles may access the lock.

In order to mitigate starvation issues with the scheme described above, the number of times local threads are favored is limited by the local lock with a starvation counter. The lock_lock() function initializes this starvation counter to a threshold upon gaining access to the global lock. Whenever a local waiting thread is woken directly, this counter is decremented. If this threshold reaches zero, the global lock is released, regardless of whether control flows are waiting locally.

Consistency

The principle idea behind the consistency strategy for locks is that a control flow that enters a critical section guarded by a lock wants to see the most recent version of the shared memory it
will access in this section. As such, the lock must perform an acquire fence to ensure that cached copies do not shadow the global state of the shared memory. Conversely, suppose the control flow modifies the shared state in the critical section. In that case, the lock implementation must ensure that the modifications become visible as soon as the control flow leaves the critical section. Hence, the unlock operation must perform a release fence upon freeing the lock.

The lock implementation realizes this scheme and, in addition, attempts to optimize the case where it passes access to the local lock from one thread to another. It does so by minimizing the number of memory fence operations. As established earlier, threads on the same tile share a coherent view of memory. Thus when passing the local lock between threads without freeing the global lock instance between, no memory fences are required—that is, as long as these fences concern the same memory regions.

The implementation uses this insight as follows: when a thread gains access to a lock, it performs an acquire fence for the specified memory regions—or the whole cache, depending on the operation—and then registers this memory range into the requested acquire list. Therefore, it first checks the requested acquire list and skips those regions that it already finds in the list. Then, for all remaining, it performs the fence and adds them to the list, thereby merging entries if they overlap.

After releasing the lock, it adds the memory ranges to release into the requested release list, similarly merging and skipping entries already found in the list. Finally, if a thread releases the global lock, either because no more local threads are waiting for access or because the threshold was reached, it performs the release fences for the entries in the requested release list.

6.2.4 Condition Variable

The condition variable operates in combination with the lock primitive. The typical use case of this primitive is that a thread takes a lock, checks for a given condition, and enqueues itself into a wait queue if this condition is not yet met. The thread, thereby, must release the lock before enqueuing itself into the wait queue to allow other control flows to modify the condition. The catch is that enqueuing in the wait queue and releasing the lock must logically appear atomically.

The condition variable primitive facilitates this concept with the interface shown in Listing 13. One or multiple threads wait on the condition variable \( c_{\text{var}} \) with \( \text{cond\_wait}(c_{\text{var}}, \text{lock}) \), whereas the lock \( \text{lock} \) guards the actual condition. Another thread can then modify the condition and finally signal one (with \( \text{cond\_signal}(c_{\text{var}}) \)) or all (with \( \text{cond\_broadcast}(c_{\text{var}}) \)) waiting threads. When a thread leaves \( \text{cond\_wait}() \), it holds access to \( \text{lock} \).

Implementation

The runtime system realizes condition variables with a single global instance. This instance lives in the private memory of the tile that initialized the condition variable with the \( \text{cond\_init}() \) function. In order to wait on a condition variable, a thread creates a local OctoPOS signal and sends an RPC with the address of this signal to the tile where the global instance resides.
6 Runtime System for Shared Memory Programming

```c
void cond_init(condition_variable_t *c_var);
void cond_wait(condition_variable_t *c_var, lock_t *lock);
void cond_signal(condition_variable_t *c_var);
void cond_broadcast(condition_variable_t *c_var);
void cond_wait(condition_variable_t *c_var, lock_t *lock,
    void **addr, size_t *size, size_t count);
void cond_signal(condition_variable_t *c_var,
    void **addr, size_t *size, size_t count);
void cond_broadcast(condition_variable_t *c_var,
    void **addr, size_t *size, size_t count);
```

Listing 13: API of the family of condition variable functions. The listing shows two types of condition variable functions: the first operations implement release consistency semantics for the operation and flush/invalidate the whole caches. The second group implements entry consistency semantics. Therefore, they have additional parameters that specify the memory ranges so that the operation can enforce consistency selectively.

Then it waits on this signal. The RPC, on the other side, stores the address of the signal in the condition variable’s wait queue. In order to signal a condition variable, a thread sends an RPC to the condition variables home tile, where the RPC then traverses the wait queue and wakes one or all waiting threads—depending on whether the cond_signal or the cond_broadcast function was invoked—by sending them RPCs that signal their respective signals.

Memory Consistency

Regarding memory consistency, the condition variable must ensure that after cond_wait(), the most current state of memory is visible. Furthermore, it must ensure that, before waking threads with cond_signal() or cond_broadcast(), any local memory modifications become globally visible. Therefore, cond_wait() performs an acquire fence when the executing thread is unblocked. Conversely, before unblocking blocked threads, cond_signal() or cond_broadcast() perform release fences.

6.2.5 Event

With the event synchronization primitive, one or multiple threads can wait for a specific event condition. Its interface is shown in Listing 14. An event consists of a flag variable which can be set with event_signal(event). A thread can wait for the flag of an instance event to be set with event_wait(event). If the flag of event is already set, the thread continues execution. If the flag is not set, it blocks and waits until it becomes set. When another thread signals the event instance with event_signal(event), it sets the flag and unblocks all currently waiting threads. In order to reuse event, it must be reinitialized with the event_clear(event) function.
void event_init(event_t *event);
void event_wait(event_t *event);
void event_signal(event_t *event);
void event_clear(event_t *event);
void event_wait(event_t *event, void **addr, size_t *size, size_t count);
void event_signal(event_t *event, void **addr, size_t *size, size_t count);

Listing 14: API of the family of event functions. The listing shows two types of event functions: the first
operations implement release consistency semantics for the operation and flush/invalidate the
whole caches. The second group implements entry consistency semantics. Therefore, they
have additional parameters that specify the memory ranges so that the operation can enforce
consistency selectively.

Implementation

Like the condition variable, the event primitive consists of a single global instance. This instance
lives in the private memory of the tile that initialized it via event_init(). Internally, an event
consists of a flag indicating whether it has already been triggered and a wait queue. Similar
to the condition variable, the wait queue thereby again stores the addresses of instances of
OctoPOS signals.

When a thread calls event_wait(), it determines the tile where the event instance resides.
Then it creates an OctoPOS signal and sends an RPC with the address of this signal to this tile.
Finally, it waits on the signal. The RPC, on the other side, first checks the flag to see whether
the event has already been signaled. If so, it directly signals this information back and wakes
the waiting thread. Otherwise, it enqueues the address of the signal into the event’s wait queue.
Upon event_signal(), in return, an RPC is sent to the tile of the event instance. The RPC
then first atomically sets the event flag. If there are waiting control flows, these are then woken
one by one by sending RPCs. The event_clear() function finally clears the event flag so the
event instance can be used again.

Memory Consistency

Concerning memory consistency, the event works not unlike the condition variable. A control
flow waiting on a signal instance expects that it sees the most current state of memory after
being signaled. Conversely, when signaling an event, the signaling thread expects its most recent
changes to become visible to all the control flows currently waiting for the event. Consequently,
the event implementation does just that: before returning the control to a control flow, the
event_wait operation performs an acquire fence for the specified memory region. Conversely,
before setting unblocking the waiting control flows, the event_signal operation performs a
release fence.
6.2.6 Conclusion

In this chapter, we introduced the design and implementation of a runtime system for shared memory programming for the SCS and ASCS. While OctoPOS provides its own set of synchronization primitives, these signals impose a parallel programming model for which not all applications can easily be adapted. The runtime system, therefore, builds upon the OctoPOS primitives to provide an alternative fork-join parallelism programming interface.

This runtime system comprises a mechanism to statically distribute threads across the tiles in the InvasIC prototype system. Furthermore, it provides barriers, locks, condition variables, and events as synchronization primitives. The primitives thereby have especially been designed for the tile-based nature of the prototype system and the entry consistency model of the SCS and ASCS: on the one side, they try to reduce cross-tile communication as much as possible, for example with the butterfly-algorithm of the barrier, or by favoring lock-handoff to threads on the same tile. Conversely, the primitives have been designed to optimize calls to the consistency system’s memory fences. Programmers can specify memory range hints with every synchronization primitive, which are then merged and delayed until absolutely necessary.
"As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications."
— Dave Parnas

7 Evaluation and Discussion

This thesis describes software consistency mechanisms that rely on strategies originating in VSM systems. These mechanisms are thereby adapted for the particularities of tiled MPSoC systems. In the following, we will test the different caching strategies of the software consistency system. By this, we aim to understand better the performance potential of software-based coherence.

We will, therefore, first establish the properties of the software consistency mechanisms that we want to investigate and how to approach them. Then, we discuss the evaluation environment we will use to assess these properties. Concerning the evaluation, we will then first examine the SCS and the ASCS with micro-benchmarks. Finally, we take a closer look into the scalability in terms of parallelism for different caching strategies of both the SCS and the ASCS at the example of selected benchmark applications from the SPLASH-3 benchmark suite.

7.1 Preliminary Considerations

Before we can delve into the evaluation, we must consider what interests us. Thereby two things come to mind: firstly, what aspects and characteristics of the consistency system are relevant, and secondly, how can we evaluate these characteristics? When put in the context of the dissertation, these questions become: in which properties of the proposed software consistency mechanisms are we interested? How can we derive meaningful statements about these properties?

In the following, we first answer the what. Based on this, we then establish the how. We will use these considerations as the guiding motif for the remainder of this chapter.

7.1.1 What is of Interest

The overarching research question of this dissertation is to explore the viability of software-based coherence for coherence-less systems. The possibility of realizing memory consistency, and thus coherence, purely in software thereby is evident, as the VSM systems of the past have demonstrated. Therefore, it is relevant for us to address the qualifier “viable” underlying the research question.

The main reason for a coherent shared memory is to provide the means to realize a parallel programming interface. The necessity for parallelism emerges from the need to perform larger working packages or speed up the execution of existing working packages. As such, the software
7 Evaluation and Discussion

consistency mechanisms must not only provide a coherent and consistent view of said memory; they must do so in a performant and scalable way.

With this consideration in mind, we can conclude that we must evaluate the software consistency mechanism concerning its scalability and performance in order for us to establish whether we can consider it a viable approach.

7.1.2 How to approach the topic of interest?

Next, we must establish a suitable evaluation strategy for the abovementioned properties. We will approach these characteristics in two steps: firstly, we will build a better understanding of the baseline execution times of the building blocks that constitute the SCS and the ASCS variants. We do so through micro-benchmarks. As a second step, we then put the software consistency mechanisms to the test through benchmarking applications—in our case, with applications from the SPLASH-3 suite [Sak+16].

7.2 Experimental Setup

The experiments were performed on the InvasIC prototype system as described in Section 2.1.1: a tile-based architecture emulated on an FPGA system, clocked at 50 MHz. Each tile thereby hosts five LEON3 cores and a TLM of 8 MiB realized as SRAM. These components use an AMBA bus as interconnect. Furthermore, each tile has an L2 cache of 4 MiB that caches memory accesses to memory on other tiles. This L2 cache is direct-mapped and has a cache line size of 32 B. The L2 cache is thereby configured to operate with write-back policy.

The architecture of the whole system is a prototype configuration of 16 tiles arranged in a 4 × 4 grid. This configuration provides a total of 80 processor cores. One of the tiles houses the memory adapter, which connects additional 2 GiB DDR memory to the prototype. These 2 GiB DDR memory act as the system’s SHM memory. Table 7.1 summarizes the system’s properties for better clarity.

<table>
<thead>
<tr>
<th>Section</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tiles</td>
<td>16 (4 × 4 layout)</td>
</tr>
<tr>
<td>ISA</td>
<td>SPARC v8 (LEON3)</td>
</tr>
<tr>
<td>Cores</td>
<td>80 (usable 48)</td>
</tr>
<tr>
<td>Clock</td>
<td>50 MHz</td>
</tr>
<tr>
<td>TLM</td>
<td>8 MiB (SRAM)</td>
</tr>
<tr>
<td>SHM</td>
<td>2 GiB (DRAM)</td>
</tr>
</tbody>
</table>

Table 7.1: Overview of the used architecture configuration.

For the evaluation, the most recent version\(^1\) of the OctoPOS kernel, including the modifications for the SCS and ASCS was used. The kernel and the benchmark applications were

\(^1\)The git commit short hash of the iRTSS repository at this version is d3da88329.
compiled with the GNU Compiler Collection (GCC) for the SPARC v8 architecture in version 8.2.0. The OctoPOS kernel was thereby compiled with the -Os compile flag to optimize the resulting image size for the limited TLM size. The application code was compiled with the optimization level -O3.

Another important detail of the used OctoPOS configuration is that it allocates two cores per tile exclusively for system tasks. The first core is solely concerned with interrupt handling, and the second core is used to construct the system claim. As such, these cores are thus not available for applications, and the evaluation system has a total of $16 \times 3 = 48$ cores for the execution of application code.

The experiments were thereby performed for the SCS and both strategies of the ASCS, exclusive and read-shared. Each of these configurations thereby used a page cache with a capacity of 256 page frames, amounting to a total of 1 MiB of page cache memory. A comprehensive overview of the characteristics of the page cache and the L2 cache is given in Table 7.2.

<table>
<thead>
<tr>
<th>Cache</th>
<th>Line Size</th>
<th>Line Count</th>
<th>Mapping</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Cache</td>
<td>4096 B</td>
<td>256</td>
<td>Fully Associative</td>
<td>1 MiB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>32 B</td>
<td>133120</td>
<td>Direct Mapped</td>
<td>4 MiB</td>
</tr>
</tbody>
</table>

Table 7.2: Overview of the L2 and page cache configurations used for the evaluation.

### 7.3 Fundamental System Characteristics

In the following, we will examine the performance characteristics of the software consistency mechanism variants. The objective is to understand better how these caches compare to one another. Furthermore, we will examine the additional costs of the consistency mechanisms, such as handling page faults.

Unless stated otherwise, every experiment was performed for a total of 1000 runs. Furthermore, for every experiment, additional 100 warm-up runs were executed upfront, for which the measured values were discarded to warm up caches and reduce a possible bias. In addition, each benchmark was performed in a non-utilized system to exclude biases of other applications. The timer used for the measurements provided timestamps at nano-second resolution. Finally, the results account for bias by measurement overheads. The average overhead for measuring time durations was determined and accounted for by subtracting the average measurement bias from the observed timing values. We will discuss the measurement overheads in the next section.

The statistical evaluation of experiments does not assume a specific distribution of the results. For this reason, the experiment results are described by median ($\tilde{\mu}$), the 0.25% and 0.75% percentiles (denoted as $Q_{0.25}$ and $Q_{0.75}$, respectively), and the minimum and maximum ($\min$ and $\max$, respectively).
7 Evaluation and Discussion

7.3.1 Accounting for Measurement Overheads

As stated above, when performing time measurements, one must also consider that our measurement code influences the results. The micro-benchmark outlined in Listing 15 determines this bias. This micro-benchmark calculates the measurement bias by taking two successive time stamps.

```c
auto start_time = timer_start();
auto end_time = timer_stop();
auto result = ticks_to_nanoseconds(end_time - start_time);
```

Listing 15: Micro-benchmark kernel that was used to establish timestamp overheads.

<table>
<thead>
<tr>
<th>( \bar{x} ) (ns)</th>
<th>( Q_{0.25} ) (ns)</th>
<th>( Q_{0.75} ) (ns)</th>
<th>min (ns)</th>
<th>max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>600</td>
<td>600</td>
<td>600</td>
<td>600</td>
<td>620</td>
</tr>
</tbody>
</table>

Table 7.3: Overview of the runtime overhead statistics for the execution of an empty loop body.

The results of the measurement overheads evaluation are presented in Table 7.3. As both \( Q_{0.25}, Q_{0.75} \), and the observed minimum lie at the same value as \( \bar{x} \), we can assume the measured 600 ns to be a reliable estimate for the average measurement bias.

7.3.2 Page Fault Handling

As SCS and ASCS rely on the MMU to realize the caching mechanism, page faults can be considered a substantial source of time overheads. Therefore, this micro-benchmark establishes their execution time by measuring the time it takes from a memory access to a non-present page to the handler routine. Conversely, to establish the overheads of leaving the page fault handler, the benchmark measures the time from leaving the page fault handler to the point where the application can continue. These times are presented in Table 7.4.

<table>
<thead>
<tr>
<th>Operation</th>
<th>( \bar{x} ) (( \mu )s)</th>
<th>( Q_{0.25} ) (( \mu )s)</th>
<th>( Q_{0.75} ) (( \mu )s)</th>
<th>min (( \mu )s)</th>
<th>max (( \mu )s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter Handler</td>
<td>10.52</td>
<td>10.52</td>
<td>10.52</td>
<td>10.52</td>
<td>15.20</td>
</tr>
<tr>
<td>Leave Handler</td>
<td>4.94</td>
<td>4.94</td>
<td>4.94</td>
<td>4.94</td>
<td>6.40</td>
</tr>
</tbody>
</table>

Table 7.4: Overview of the runtime overhead statistics for entering and leaving the page fault handler.

We can see there that the median for entering the page fault handler lies at \( \bar{x} = 10.52 \mu \)s, with the \( Q_{0.25}, Q_{0.75} \), and minimum also lying at this value. The high maximum outlier for entering thereby can likely be explained by interference from other interrupt handlers. Moreover, we can observe a similar distribution for the measurements for leaving the page fault handler,
whereas we here have \( \bar{x} = 4.94 \mu s \). Therefore, we assume in the following that the average time for entering and leaving trap handlers equals their observed median values.

### 7.3.3 Syscall Overheads

Another factor of the software consistency mechanisms we must consider is memory fences. OctoPOS exposes these fences at its system call interface. While OctoPOS is a library operating system, and the system call mechanism does not involve expensive transitions between privilege levels, invoking them still induces considerable costs.

The micro-benchmark measures the execution time from the call to a memory fence until the corresponding function in the software consistency mechanism has been entered. Similarly, the micro-benchmark determines the time from completing a memory fence until the control flow returns to the application code. Both paths are similar for the acquire and release fences in all configurations of the software consistency mechanism, so it is sufficient to consider only one of the fences for the evaluation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>( \bar{x} ) (( \mu s ))</th>
<th>( Q_{0.25} ) (( \mu s ))</th>
<th>( Q_{0.75} ) (( \mu s ))</th>
<th>min (( \mu s ))</th>
<th>max (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syscall Enter</td>
<td>98.60</td>
<td>98.60</td>
<td>98.60</td>
<td>98.60</td>
<td>122.02</td>
</tr>
<tr>
<td>Syscall Leave</td>
<td>104.82</td>
<td>104.82</td>
<td>104.82</td>
<td>104.82</td>
<td>119.10</td>
</tr>
</tbody>
</table>

Table 7.5: Overview of the runtime overhead statistics for entering and leaving the memory fence syscalls.

As can be seen in Table 7.5, entering a memory fence syscall on average takes \( \bar{x} = 98.60 \mu s \), with \( Q_{0.25} = 98.60 \mu s \) and \( Q_{0.75} = 98.60 \mu s \) being tight around this median. The numbers for leaving the system call paint a similar picture with \( \bar{x} = 104.82 \mu s \), and the quantiles \( Q_{0.25} = 104.82 \mu s \), \( Q_{0.75} = 104.82 \mu s \). Like for the previous experiments, we thus consider the median a suitable metric to estimate the average duration for the memory fence syscalls.

### 7.3.4 DMA Transfers

The DMA unit of the InvasIC system is used extensively in all variants of the software consistency mechanisms to copy memory pages between tiles. As such, almost all coherence operation involving the page cache also involves a DMA in one form or another.

To determine the time that a DMA takes to transfer a memory page (4096 B on the LEON3 system), this benchmark times the transfer of a local memory page in TLM to the SHM. Therefore, it takes the start time when initiating the DMA operation and the completion time once the completion i-let is scheduled.

Table 7.6 shows the results for the DMA benchmark. We can see here that, on average, the transfer of a memory page takes around \( \bar{x} = 68.70 \mu s \) the overall completion time for this operation thereby varies considerably. We explain these variations with the fact that the DMA transfer involves several communication mediums, the local and the remote Advanced High-performance Bus (AHB) buses, the NoC, and the DDR controller. Note that these numbers
### 7 Evaluation and Discussion

<table>
<thead>
<tr>
<th>Variant</th>
<th>( \bar{x} ) (ns)</th>
<th>( Q_{0.25} ) (ns)</th>
<th>( Q_{0.75} ) (ns)</th>
<th>min (ns)</th>
<th>max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1140</td>
</tr>
<tr>
<td>SCS</td>
<td>820</td>
<td>820</td>
<td>820</td>
<td>820</td>
<td>840</td>
</tr>
<tr>
<td>ASCS Exclusive</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
</tr>
<tr>
<td>ASCS Read-Shared</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1100</td>
<td>1140</td>
</tr>
</tbody>
</table>

Table 7.6: Overview of the runtime statistics for DMA copy operations for buffers of the size of 4096 B.

Table 7.7: Overview of the runtime statistics for serving cache hits in the different caching architectures.

The timing characteristics for the different cache implementations are shown in Table 7.7. An eye-catching result of this benchmark is that the access time to cached data is 1.34 times faster for the page cache (\( \bar{x} = 8.20 \times 10^2 \) ns) than for the L2 cache (\( \bar{x} = 1.10 \times 10^3 \) ns).

This result is astonishing insofar as both the L2 cache and the TLM are realized as SRAM. One would thus expect roughly the same access performance. However, comparing the respective benchmark loop bodies in assembly showed no differences that would explain the difference in their execution times. One possible explanation would be additional tag comparison logic in the L2 cache, which could amount to additional latencies.

only reflect the transfer duration in a non-utilized system. They must therefore be understood as a lower bound for DMA transfers. Nevertheless, these figures allow us to estimate the approximate costs for DMA operations in the cache operations.

#### 7.3.5 Cache Operations

Having established the timing characteristics of the operations that the software consistency mechanisms build upon, we can now continue with a close examination of the actual cache operations. As the ASCS strategies rely on the L2 cache, the measurements include the L2 cache operations.

**Cache Hit Latency**

The cache hit latency was evaluated by measuring the time that the cache takes to serve a hit. Therefore, the cache hit latency benchmark first accesses a data word in remote memory. This access causes the L2 cache to cache the respective cache line. Then the benchmark flushes the L1 cache to ensure the L2 cache serves subsequent access. After this preparation, the benchmark measures the time for a read access on the remote data word.
Both adaptive caching strategies, in return, show the same performance characteristics as the L2 cache. This result is evident as both use the L2 cache in the benchmark.

Compulsory Miss Latency

A compulsory miss refers to cache misses that occur when the cache is cold: the cache does not yet contain the requested cache line but has sufficient capacity to store it. In order to determine the timing characteristics for this type of miss, a benchmark is applied that first invalidates the respective cache and the L1 cache. After this, the benchmark measures the time to read a remote data word.

<table>
<thead>
<tr>
<th>Variant</th>
<th>$\bar{x}$ (μs)</th>
<th>$Q_{0.25}$ (μs)</th>
<th>$Q_{0.75}$ (μs)</th>
<th>min (μs)</th>
<th>max (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache</td>
<td>4.36</td>
<td>4.34</td>
<td>4.38</td>
<td>4.32</td>
<td>4.64</td>
</tr>
<tr>
<td>SCS</td>
<td>888.68</td>
<td>888.26</td>
<td>889.21</td>
<td>887.26</td>
<td>975.32</td>
</tr>
<tr>
<td>ASCS Exclusive</td>
<td>790.94</td>
<td>790.56</td>
<td>791.76</td>
<td>789.58</td>
<td>827.34</td>
</tr>
<tr>
<td>ASCS Read-Shared</td>
<td>789.68</td>
<td>789.46</td>
<td>789.80</td>
<td>789.24</td>
<td>824.32</td>
</tr>
</tbody>
</table>

Table 7.8: Overview of the runtime statistics for serving compulsory misses for the different caches.

The results for the benchmark are presented in Table 7.8. Interestingly, the ASCS strategies perform only slightly better than the SCS. We can see that it takes the SCS roughly 100 μs longer to serve a capacity miss than the ASCS. When we additionally consider the results of the earlier DMA benchmark, we find, that this approximately corresponds to the time to perform a DMA and possibly perform some cache management tasks.

We conclude that a significant fraction of the compulsory miss handling in the SCS and ASCS, therefore, must be spent for maintaining cache management operations and inter-tile communication.

Capacity Miss Latency

A capacity miss occurs when a requested value is not found in the cache, and the capacity of the cache is exhausted. That means the requested cache line will not just fit into the cache. Therefore, the cache controller must first free a cache line before it can then continue to load the requested line. The benchmark that measures the time for handling a capacity miss first performs as many memory accesses as are necessary to fill the cache. It then measures the time to access an uncached data word. As the ASCS strategies do not track the capacity of the hardware cache directly, this benchmark is only meaningful for the L2 cache and the SCS variant. Thus, these variants were excluded from the experiment. Table 7.9 shows the results of this benchmark.

When comparing the duration for the SCS with the results for the compulsory miss in Table 7.8, we see that handling this miss takes almost twice as long. This result is natural, as
7 Evaluation and Discussion

Table 7.9: Overview of the runtime statistics for serving capacity misses for the different caches.

<table>
<thead>
<tr>
<th>Variant</th>
<th>$\bar{x}$ (µs)</th>
<th>$Q_{0.25}$ (µs)</th>
<th>$Q_{0.75}$ (µs)</th>
<th>min (µs)</th>
<th>max (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache</td>
<td>4.90</td>
<td>4.88</td>
<td>4.96</td>
<td>4.84</td>
<td>5.12</td>
</tr>
<tr>
<td>SCS</td>
<td>1811.94</td>
<td>1811.54</td>
<td>1812.32</td>
<td>1807.48</td>
<td>1981.34</td>
</tr>
<tr>
<td>Exclusive</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Read-Shared</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7.10: Overview of the runtime statistics for writing back cache lines for the different caches.

The cache controller for a capacity miss must make room for the to-be-fetched page by first writing an occupied cache entry back.

Furthermore, we see, that capacity misses in the L2 cache take almost as long as compulsory misses. This indicates that the L2 cache uses a store buffer or similar technique to wait for acknowledgments for evicted cache lines. This is insofar interesting, as the ASCS variants will ultimately run into L2 cache capacity limits. As such, this timing will also affect the overall ASCS performance.

Writeback

In order to measure the writeback operation, the micro-benchmark measures the time to write back modified cache lines. The micro-benchmark, therefore, modifies some remote data and then invokes the writeback operation of the underlying cache mechanism.

The results of this experiment are shown in Table 7.10. For the evaluation results of the SCS, we thereby additionally distinguish the fast-writeback mechanism we discussed in Chapter 4 and the diff-based writeback mechanism. In return, the results for the ASCS show the writeback for the cases where the strategy can use the hardware caches.

<table>
<thead>
<tr>
<th>Variant</th>
<th>$\bar{x}$ (µs)</th>
<th>$Q_{0.25}$ (µs)</th>
<th>$Q_{0.75}$ (µs)</th>
<th>min (µs)</th>
<th>max (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache</td>
<td>5.28</td>
<td>5.26</td>
<td>5.32</td>
<td>5.20</td>
<td>5.60</td>
</tr>
<tr>
<td>Page-Cache-Only</td>
<td>1044.74</td>
<td>1044.56</td>
<td>1044.98</td>
<td>1040.52</td>
<td>1198.18</td>
</tr>
<tr>
<td>Fast</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page-Cache-Only</td>
<td>1415.56</td>
<td>1415.30</td>
<td>1416.06</td>
<td>1411.14</td>
<td>1544.94</td>
</tr>
<tr>
<td>Only Diff-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASCS Exclusive</td>
<td>604.60</td>
<td>604.60</td>
<td>604.62</td>
<td>601.24</td>
<td>636.24</td>
</tr>
<tr>
<td>ASCS Read-Shared</td>
<td>608.96</td>
<td>608.92</td>
<td>608.98</td>
<td>605.16</td>
<td>638.66</td>
</tr>
</tbody>
</table>

Table 7.10: Overview of the runtime statistics for writing back cache lines for the different caches.
For the SCS, the performance differences between the fast- and diff-based writeback are noticeable. The fast-writeback method is 35% faster than the diff-based mechanism. Recall that both writeback mechanisms do not differ too much: the main difference is that the diff-based mechanism additionally generates and applies the diff of the page. Thus, it is safe to assume that both writeback mechanisms spend most of their execution time on administration activities and inter-tile communication. Still, this implies that applications profit when they limit their sharing.

Finally, both ASCS strategies show roughly the same runtime for performing the writeback operation, which is logical, as both approaches perform similar code paths. However, in general, the ASCS must only flush the L2 cache to perform a writeback, as compared to the SCS which has to writeback the entire page’s contents. This difference shows as the ASCS is faster than the fast-writeback path of the SCS.

**Invalidation**

The invalidation benchmark determines the runtime of the invalidate operation. Therefore, it measures the time it takes the caches to mark their cached data as invalid. The micro-benchmark, therefore, first reads some data from remote memory to ensure that valid data is stored in the cache. It then measures the time to perform an invalidation operation on the just-read data. The results for this operation are presented in Table 7.11.

<table>
<thead>
<tr>
<th>Variant</th>
<th>( \mu s )</th>
<th>( Q_{0.25} (\mu s) )</th>
<th>( Q_{0.75} (\mu s) )</th>
<th>min (( \mu s ))</th>
<th>max (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2 Cache</td>
<td>7.66</td>
<td>7.66</td>
<td>7.66</td>
<td>7.66</td>
<td>7.72</td>
</tr>
<tr>
<td>SCS</td>
<td>774.35</td>
<td>773.70</td>
<td>774.46</td>
<td>770.46</td>
<td>888.62</td>
</tr>
<tr>
<td>ASCS Exclusive</td>
<td>822.86</td>
<td>822.58</td>
<td>823.21</td>
<td>818.20</td>
<td>854.18</td>
</tr>
<tr>
<td>ASCS Read-Shared</td>
<td>827.41</td>
<td>827.08</td>
<td>827.72</td>
<td>823.42</td>
<td>838.22</td>
</tr>
</tbody>
</table>

Table 7.11: Overview of the runtime statistics for invalidating cache lines for the different caches.

There are two notable details about the results for this benchmark: firstly, one would consider invalidating cache contents a relatively cheap operation. However, the long duration this operation takes in the SCS and the ASCS can be explained due to the involved inter-tile communication for bookkeeping in the directory controller.

The second interesting detail about these results is that invalidating a page from the cache takes longer for the ASCS variants than for the SCS. This is because the ASCS variants must invalidate all L2 cache lines in the page range.

As we have discussed in Chapter 5, the ASCS implement additional logic for their permission management. In the following, we take a closer look into the permission transitions of these strategies.
Handling Shared Cache Lines in the ASCS Exclusive Strategy

As described in Chapter 5, the directory controller in the ASCS exclusive strategy ensures that only one tile can cache pages in its L2 cache at a time. Therefore, when a tile requests access to an L2 cached page, the directory controller first sends an invalidate to the page’s current exclusive owner. The benchmark that estimates the timing characteristics for cases where such an invalidate is necessary operates as follows. First, it accesses a remote data word from another tile, thus ensuring that the containing page is cached on this tile. Then the micro-benchmark measures the time to access said data word locally.

<table>
<thead>
<tr>
<th>Variant</th>
<th>(\bar{x}) ((\mu s))</th>
<th>(Q_{0.25}) ((\mu s))</th>
<th>(Q_{0.75}) ((\mu s))</th>
<th>min ((\mu s))</th>
<th>max ((\mu s))</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCS Exclusive</td>
<td>1571.90</td>
<td>1571.46</td>
<td>1572.50</td>
<td>1567.02</td>
<td>1662.72</td>
</tr>
</tbody>
</table>

Table 7.12: Overview of the runtime statistics for serving a miss on an exclusively cached page cache lines for the different caches.

The timing characteristics for this operation are shown in Table 7.12. When comparing these numbers to the timing properties shown in Table 7.8, we can see that serving a miss for a page exclusively held by another tile takes almost twice as long as a cache miss for an unshared page and 1.7 as long as granting access to a page in the SCS. This stands to reason, as this operation involves the costs of invalidating the cache entries on another tile and the costs of creating a page copy.

Handling Shared Cache Lines in the ASCS Read-Shared Strategy

Finally, we have to account for the state transitions of the ASCS read-shared strategy. The directory controller performs different operations depending on the access permissions the directory grants for a memory page. These transitions are:

- Reading from a page with \(P^S_R\) access permissions: a tile reads from a tile for which one or multiple other tiles have read-only access. The directory controller adds the tile to the sharers and grants \(P^S_R\) access permissions.

- Write to a page with \(P^S_R\) access permissions: a tile performs a write to a page for which other tiles have read-only access permissions. Before the directory controller can grant write access, it revokes the \(P^S_R\) access permissions and invalidates the corresponding cached.

- Reading from/writing to a page with \(P^{E}_{R:AW}\) access permissions: a tile performs a read from or a write to a page to which another tile has exclusive write permissions. The directory controller invalidates the exclusive copy. The tile with \(P^{E}_{R:AW}\) access permissions thus invalidates its copy and flushes the page from the L2 cache.

The latter two cases, thereby, finally involve the creation of a page copy.
In order to determine the timing characteristics for all these cases, the benchmarks operate conceptually similarly. First, they cause another tile to perform a remote memory access to ensure it is cached—depending on the case, either by reading from (to grant $P_R^S$ access permissions) or writing to it (to ensure $P_{R,W}^E$ access permissions). Locally, the benchmark then measures the time to read from or write to this data word. The results for this micro-benchmark are shown in Table 7.13.

<table>
<thead>
<tr>
<th>Operation</th>
<th>$\bar{x}$ (\mu s)</th>
<th>$Q_{0.25}$ (\mu s)</th>
<th>$Q_{0.75}$ (\mu s)</th>
<th>min (\mu s)</th>
<th>max (\mu s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read to Read-Shared</td>
<td>812.04</td>
<td>811.74</td>
<td>812.30</td>
<td>811.62</td>
<td>850.30</td>
</tr>
<tr>
<td>Write to Read-Shared</td>
<td>986.40</td>
<td>985.88</td>
<td>987.10</td>
<td>984.44</td>
<td>1089.68</td>
</tr>
<tr>
<td>Write to Exclusive</td>
<td>1640.02</td>
<td>1639.64</td>
<td>1640.32</td>
<td>1636.16</td>
<td>1741.62</td>
</tr>
</tbody>
</table>

Table 7.13: Overview of the runtime statistics for serving the different miss types on a shared page in the adaptive-read-shared strategy.

As expected, reading from a $P_R^S$ page is the fastest operation. For this operation, the directory controller notes the reading tile as another sharer and then grants access. The tile then maps the page as present with $P_R^S$ access permissions.

Interestingly, writing to a $P_R^S$ page is also relatively inexpensive compared to reading access. This is interesting insofar as the operation invalidates the L2 cache entries on the sharing tiles and creates a page copy. However, the directory controller here takes advantage of the fact that it does not have to wait for the invalidation of L2 entries. Instead, it starts directly with creating the page copy. In contrast, writing to the page for which another tile has $P_{R,W}^E$ access permissions takes significantly longer. As the exclusive copy holder can modify its copy, the directory controller must first ensure that all cache lines for said page in the L2 cache have been flushed.

7.3.6 Memory Access Latencies

The measurements above primarily focused on operations to single memory pages. However, we can expect working set sizes to exceed this limit in actual applications. Therefore, the following investigates how the SCS and ASCS perform for larger working set sizes. We also include the L2 cache as an additional reference to understand the adaptive approach’s performance better.

This benchmark measures the duration for successive word-wise memory accesses to a buffer in remote memory. The measurements were performed for every caching strategy for buffers of increasing size, starting with 64 B, up to 32768 B in 64 B increments like the other micro-benchmarks, 1000 samples were measured for each buffer size with additional 100 warm-up runs upfront.

Using the results of the previous benchmarks, we already gathered some knowledge that allows us to predict the outcome: firstly, we know that the hits in the L2 cache take longer than those in the page cache. Secondly, as the L2 cache line size is 32 B, the L2 cache benchmark will experience additional cache misses for every eighth access. This contrasts the page cache
where every 1024\textsuperscript{th} access leads to a page miss. We know that the cache misses in the software consistency mechanisms will take significantly longer than those of the L2 cache. The ASCS strategies, finally, will experience both the cache misses of the L2 cache and cache misses when crossing page boundaries.

![Graph showing the median execution time for word-wise successive store operations to remote memory ranges of increasing sizes. As both ASCS strategies handle cache misses similarly, both show similar access delays and have been grouped as the green graph. The vertical dotted lines show where the buffer size exceeds page boundaries.](image)

Figure 7.1 shows the results of this benchmark and supports these predictions. This figure shows the median time duration for word-wise stores concerning the working set sizes. As both strategies of the ASCS apply the same logic to handle cache misses, they show the same results in the evaluation. As such, we chose to show only the graph for the exclusive strategy.

As we can see in the figure, both the graph for the L2 cache and the ASCS have the same slope—at least for the accesses within page boundaries. This observation tells us that these approaches have the same memory access times. This result is expected, as in both cases, cache hits are served by the L2 cache. Furthermore, this slope is higher than that of SCS, consistent with the results presented in Table 7.7. Finally, we see characteristic steep increases in access times for the ASCS strategies and the SCS whenever their working sets cross multiples of the page size.

Interestingly, the adaptive caching strategies generally perform worse than the SCS. While handling a cache miss in the adaptive strategies, on average, takes slightly shorter than for the SCS, the overall worse cache hit time compensates for that. As the working set size increases, the runtime difference between these strategies increases further.
7.3.7 Summarizing Thoughts on the Micro-Benchmark Results

Above, we explored the characteristics of the SCS, the ASCS strategies, and, for reference, the L2 cache. In general, we could thereby make three interesting observations:

1. The ASCS variants do not show significant differences in the micro-benchmarks. This stands to reason, as both strategies, in principle, only vary in how they handle page sharing, a property that most, except for two, of the micro-benchmarks did not consider.

2. Considering the latency to serve cache hits, the L2 cache, in general, performs worse than the TLM. The L2 cache serves hits on average 25% slower than the SCS. This, of course, also affects the adaptive strategies. However, we consider this to be an implementation-specific property, that is, dependent on the used SRAM and L2 cache components and the FPGA system. Nevertheless, as is obvious for the larger problem sets, this will affect the overall system performance. As such, we must remember this fact when interpreting the application benchmarks in the following section.

3. The DMA transfer times are not the dominating factor for the execution times when handling cache misses. Instead, the code for management in the cache and directory controller appears to take a far larger fraction of these times. We have seen that the actual transfer of page data only plays an insignificant part in the overall execution times for the cache operations. This becomes evident when comparing the costs for DMA transfers, as shown in Table 7.6, to the overall costs for handling a compulsory cache miss, for the SCS, shown in Table 7.8: the DMA transfer makes only up 7.7% of the operations total time. This is also visible in the observation that handling compulsory misses in the ASCS takes slightly less time than handling the miss for the SCS.

The graphs in Figure 7.1 also underline these observations: the ASCS strategies perform worse than the SCS. This result is logical, as for the ASCS strategies, the memory access times involve costly compulsory miss handling, the slower access times of the L2 cache, and frequent L2 cache misses. In contrast, under the SCS strategy, there is only compulsory miss handling every 4096 B. Moreover, this effect worsens as the working set size increases.

However, this impression is only accurate at first glance, as the benchmark only shows compulsory misses. Once the capacity of the cache is reached, however, the following misses will be capacity misses. As we have seen in Table 7.9, these are significantly more expensive in the SCS than in the ASCS. Once the cache capacity is reached, the ASCS strategies likely outperform the SCS. This effect should still be visible even once the capacity for the L2 cache is reached. This is because serving compulsory misses is still relatively cheap—compared to the operation of the SCS—and thus, this effect will maintain.

In the following, we will now have a closer look at caching strategies in the setting of real-world applications.
7 Evaluation and Discussion

7.4 Evaluation with Real World Applications

In the previous sections, we established the fundamental properties of the software consistency mechanisms. While we could draw some initial conclusions from these experiments, micro-benchmarks generally only have limited expressiveness. To complete the picture, we must also consider scenarios closer to the real world.

As such, the strategies were additionally evaluated with selected applications from the SPLASH-3 benchmark suite [Sak+16]. The SPLASH-3 benchmark suite contains an array of parallel application scenarios and kernels of scientific computational workloads. These workloads demonstrate the effectiveness of shared memory interfaces for different memory access scenarios. The SPLASH-3 suite thereby builds upon the SPLASH-2 suite [Woo+95] and integrates fixes for data races and other performance issues.

7.4.1 Benchmark Setup

The following evaluation explores the scalability of the SCS and the ASCS strategies, by these three evaluation parameters:

1. **Degree of Parallelism:** 1, 2, 4, 8, 16, and 32 threads
2. **Caching Mechanism:** SCS, ASCS exclusive, and ASCS read-write
3. **Thread Distribution Strategy:** round-robin and tile-wise

The latter configuration parameter requires explanation: for the execution of the benchmark applications, the application threads are pinned to individual cores. The thread distribution strategy thereby describes how and in what order threads are pinned to cores in the system. The tile-wise strategy assigns threads to cores of the same tile until no more free cores are available before moving to the next tile. The round-robin strategy, in return, distributes the threads in a round-robin fashion, assigning threads to the cores of different tiles in turns.

In theory, the former strategy thus should exploit locality, as it puts more threads together. In contrast, the latter strategy is beneficial without much sharing, as the threads for lower core counts will not have to compete for cache resources.

7.4.2 Overview of the Benchmark Applications

The selection of benchmark applications was based on their compatibility with the evaluation system. Unfortunately, some applications had proven to be incompatible as they either had too large memory requirements or required specific unsupported system calls that OctoPOS does not provide. For example, the Cholesky benchmark application requires much stack memory and repeatedly causes call-stack overruns. The Volrend benchmark, on the contrary, required file I/O not provided by OctoPOS.

Table 7.14 shows an overview of the selected applications and kernels in the benchmark suite and a brief description of their functionality. A more in-depth description of the functionality of the respective benchmark applications and kernels is given in [Woo+95].
The benchmark suite suggests for each benchmark a specific problem set size. Unless stated differently in Table 7.14, all benchmarks were performed with the suggested problem set sizes. Where necessary, due to the benchmarks exceeding execution time bounds, the problem size was reduced. The threshold for the execution time was thereby chosen at five minutes.

Every benchmark application ran for every of the configuration options described above for a total of ten runs. The completion time reported by the applications was then taken for evaluation. The execution times were thereby measured with the OctoPOS wall clock timer, which provides timestamps with microsecond accuracy.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FFT</td>
<td>1D FFT on Complex Numbers</td>
<td></td>
</tr>
<tr>
<td>LU (contiguous blocks)</td>
<td>LU Matrix Factorization</td>
<td></td>
</tr>
<tr>
<td>LU (non-contiguous blocks)</td>
<td>LU Matrix Factorization</td>
<td>Reduced Problem Set Size (-n256)</td>
</tr>
<tr>
<td>Radix</td>
<td>Integer Radix Sort</td>
<td>Reduced Problem Set Size (-n262144)</td>
</tr>
<tr>
<td>applications</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ocean (contiguous partitions)</td>
<td>Simulation of Ocean Movement</td>
<td>Reduced Problem Set Size (-n130)</td>
</tr>
<tr>
<td>Water (spatial)</td>
<td>Simulation of Water Molecules</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.14: Overview of the used kernels and applications in the SPLASH-3 benchmark suite. The note column shows where a problem size deviating from the suggested configuration was used.

7.4.3 Notes on the Execution of the Experiments

In order to make the SPLASH-3 benchmark suite compatible with OctoPOS, some modifications to the benchmarks and to OctoPOS were necessary:

1. Shared memory runtime system: the SPLASH-3 applications implement fork-join parallelism and use the ANL PARMACS macros [LO85] to realize synchronization. The macros were adapted for the runtime environment for shared memory parallel applications described in Chapter 6. This system thereby realizes thread-to-core mapping, synchronization primitives, and memory consistency. Note that while the SCS and ASCS
realize entry consistency, the SPLASH-3 benchmarks use the stricter release consistency model. This is to limit the necessary modifications to the benchmark source base.

2. Mapping of shared data to the SHM: as every tile in the InvasIC system runs its instance of the OctoPOS kernel, each tile comes with its data segment. The SPLASH-3 benchmarks, however, suppose that all threads share a single address space with (unique) instances of the globally defined variables. Therefore, globally defined variables are mapped once to the system's SHM. Thus, all threads, independently of the tile they are running on, see and operate the same instances for these variables.

3. File input emulation for input parameters: the benchmark programs expect to read their inputs with the I/O primitives of the C standard library, either by opening the appropriate file or reading from stdin. As OctoPOS does not provide file-stream or file-system abstractions, the runtime system emulates this input mechanism with a custom C library that provides the necessary interfaces. Behind the curtains, the contents of the problem set input files are compiled statically into the binary as null-terminated strings.

Finally, the fast-writeback mechanism we discussed in Section 4.3.1 was found to lead to corrupted memory consistently. This is apparently an unresolved issue in the DMA engine that prevents it from writing correctly to the DDR memory. We come to this conclusion as the fast-writeback is the only operation that transfers memory from the TLM directly to the SHM. Unfortunately, as of the writing of this dissertation, this bug could not be fixed. The benchmarks, therefore, were performed without the fast-writeback mechanism.

7.4.4 FFT

This benchmark kernel implements a complex 1-D FFT algorithm. Figure 7.2 shows box plots for the execution time for \( n = 10 \) runs over an increasing degree of parallelism. The subfigures are organized as follows: the rows show the strategy of the runtime system to distribute the threads in the system. The first row thereby shows the results for the round-robin distribution, whereas the second row shows the tile-wise distribution. The columns, in return, show the used caching strategy. The leftmost column is the SCS, the middle column is ASCS exclusive, and the rightmost column is the ASCS read-shared strategy. The following benchmark result figures follow the same organization.

We can make some observations from these benchmark results. Firstly, the ASCS strategies generally perform better than the SCS. This is especially true for lower degrees of parallelism but is still noticeable when more threads are used. Furthermore, the ASCS read-shared approach performs slightly better than the ASCS exclusive strategy.

Secondly, we see that all configurations initially profit from increased parallelism; however, they do so to a different degree. Most notably, however, as the parallelism surpasses certain bounds, we can observe that the runtime increases again considerably. This bound lies at eight threads for the round-robin variants: the SCS and ASCS exclusive strategies already take
7.4 Evaluation with Real World Applications

Figure 7.2: Box plots of the runtimes for the FFT kernel for different configurations of the evaluation system. The top row shows the thread distribution in round-robin distribution, whereas the bottom row shows the tile-wise distribution. The columns show the respective caching strategies. Each plot shows the varying execution times (shown on the vertical axis) over a varying degree of parallelism (on the horizontal axis) with box plots performed for \( n = 10 \) runs.

slightly longer than for four threads, and the ASCS read-shared variant is only slightly faster than with four threads.

The tile-wise variants show a similar pattern concerning their scalability. However, the turning point lies at 16 threads. For the SCS, 16 threads are slower than eight threads. The ASCS strategies, in return, show an additional performance gain at 16 threads.

Overall, the tile-wise distribution performs worse than the round-robin distribution for lower degrees of parallelism. However, they achieve better scalability for higher thread counts. At these thread counts, they even outperform the round-robin distribution.

We interpret the observed results as follows: the FFT kernel does not benefit directly from data locality in the caches. Consequently, we can observe that the tile-wise strategy performs worse than the round-robin strategy. However, we see an indirect benefit in better scalability for the tile-wise configuration. As the degree of parallelism increases, the number of requests that the directory controller on the SHM tile has to serve increases, as well. This growth in requests also causes the time to serve requests to grow. Effectively, the directory controller thus becomes a bottleneck. This is the case for both distribution strategies, round-robin and tile-wise. However, in the tile-wise configuration, more cores share the same tiles. Thus threads on the same tile, per the distribution strategy of the benchmark, share cache lines and thus can produce fewer requests to the directory controller.
7 Evaluation and Discussion

7.4.5 LU (Contiguous Blocks)

The LU kernel performs a blockwise LU decomposition of a square matrix. The contiguous blocks variant constructs the blocks so that their elements lie contiguous in memory for better spatial locality [Woo+95]. The runtime results for this benchmark kernel are presented in Figure 7.3.

![LU Contiguous Blocks Execution Times](image)

**Figure 7.3:** Box plots of the runtimes for the LU (Contiguous Blocks) kernel for different configurations of the evaluation system. The top row shows the thread distribution in round-robin distribution, whereas the bottom row shows the tile-wise distribution. The columns show the respective caching strategies. Each plot shows the varying execution times (shown on the vertical axis) over a varying degree of parallelism (on the horizontal axis) with box plots performed for 10 runs.

The benchmark results show that the LU kernel scales well with increasing parallelism for all configurations. Furthermore, we can make additional observations when comparing the configuration. Firstly, for lower degrees of parallelism (<16 threads), all strategies appear to perform better for the round-robin distribution than for the tile-wise distribution. After this point, the tile-wise distribution outperforms the round-robin distribution. Our explanation for this observation is that each thread runs on a separate tile in the round-robin distribution with less than 16 threads. Therefore, each thread can exploit far more of the respective tile’s cache (page cache and L2 cache alike).

In configurations that map multiple threads to a single tile, two possible outcomes are imaginable, depending on the application’s memory access model. Either the threads profit from locality by sharing cache lines, or they suffer as they evict each other’s cache lines. We suspect the latter scenario for this benchmark. As the system uses 15 compute tiles, we see
this in effect for the round-robin strategy for 16 and more cores, whereas the tile-wise strategy suffers from the effect right from the start.

A second observation is that the ASCS strategies outperform the SCS. We explain this by the fact that, as the degree of parallelism increases, so does the number of requests on the page directory. As the ASCS involve less DMA operations, the page directory and the DMA unit on this tile are less of a bottleneck.

Finally, the ASCS read-shared strategy performs worse than the ASCS exclusive strategy. This observation is especially evident for smaller numbers of threads. Our interpretation is that in the ASCS exclusive strategy, the directory controller directly grants write access to a page upon first access, independent of the type of access. In contrast, a read access leads to read-only permissions under the ASCS read-shared strategy. Consequently, the ASCS read-shared strategy leads to more requests on the page directory, depending on the types of memory accesses an application performs.

7.4.6 LU (Non-Contiguous Blocks)

This application is a variant of the LU matrix decomposition described in the previous section. As the name suggests, this variant does not use continuous blocks. It does so for a simplified programming model under the tradeoff of less spatial locality. The execution times for this benchmark are shown in Figure 7.4.

This benchmark overall does not scale well, especially for the round-robin distribution. We observe a similar pattern for all caching strategies: the runtime initially decreases from one to two cores, only to increase as the degree of parallelism increases. Interestingly, however, we see the shortest runtime for 32 threads.

Conversely, the tile-wise distribution also does not show good scalability. However, the overall pattern is better than that of the round-robin distribution. Here, for all caching strategies, we see that the benchmark initially profits from increased parallelism but then plateaus from 2 to 16 threads. Again, we can observe the minimal runtime for 32 threads.

We explain this pattern as follows: as initially described, the memory of the blocks in this benchmark is not contiguous. Therefore the threads perform more access to different pages, as they would for the contiguous variant of this application. For the round-robin distribution, this means that the directory controller on the SHM tile must serve more requests as the number of threads and, thus, tiles increase. Again, with more than 15 threads, multiple threads are allocated to a tile; thus, these threads share their respective caches. The decrease in runtime for the 32 threads suggests that these threads then profit from the locality in the shared caches. This also explains the runtime pattern we see for the tile-wise strategy.

Concerning the caching strategies, we see that the ASCS exclusive caching strategy performs best. On the other hand, the ASCS read-shared strategy performs equally or, for the round-robin case, even worse than the SCS. We interpret this as follows: the ASCS read-shared strategy requires more requests per memory access when a write access follows an initial read to a page.
7 Evaluation and Discussion

7.4.7 Radix

The radix benchmark sorts an array of integers using a parallel radix sorting algorithm. The results for this benchmark are presented in Figure 7.5.

The benchmark scales under the round-robin distribution for up to four threads, whereas the runtime increases starting at eight threads. For the tile-wise distribution, the benchmark scales for up to eight threads and similarly increases execution time for higher degrees of parallelism.

Following the argument of the previous benchmarks, we explain the increase in runtime for 16 and 32 threads by the directory controller becoming a bottleneck as the number of requests increases.

Notably, for the tile-wise distribution, the execution time for two and four threads is higher than that for a single thread. Our interpretation of this result is that this pattern is due to the many non-consecutive memory accesses. In the tile-wise distribution with two and four threads, many threads share the same tile, so it is likely for these threads to fill the cache quickly and then repeatedly cause capacity misses. In other words, threads compete for cache entries, causing cache thrashing.
7.4 Evaluation with Real World Applications

Figure 7.5: Box plots of the runtimes for the Radix kernel for different configurations of the evaluation system. The top row shows the thread distribution in round-robin distribution, whereas the bottom row shows the tile-wise distribution. The columns show the respective caching strategies. Each plot shows the varying execution times (shown on the vertical axis) over a varying degree of parallelism (on the horizontal axis) with box plots performed for $n = 10$ runs.

Interestingly, the ASCS strategies perform significantly better for this benchmark than the SCS. This observation suggests that the benchmark experiences many capacity misses. As we have seen in Table 7.9, these are especially expensive to serve with the SCS.

7.4.8 Ocean (Contiguous Partitions)

This benchmarking application realizes a simulation of ocean movements. The runtime data for this benchmark are shown in Figure 7.6.

This benchmark does not show good scalability for the round-robin distribution under all caching strategies. We can see a similar pattern for all of them: after an initial steep decrease in the runtime from one to two cores, the overall runtime increases steadily with the degree of parallelism. In fact, the benchmark shows the best runtime when only two threads are used.

On the other side, we can observe a steady decrease in the runtime until eight threads for the tile-wise distribution under all caching strategies. Again, this pattern is similar to all caching strategies. However, the overall runtime remains worse than the round-robin strategy except for eight threads.

These observations suggest that the benchmark accesses a large working set. When the threads are spread evenly over multiple tiles, as is the case of the round-robin distribution, the benchmark performance is bound by the performance of the directory controller and the DMA.
7 Evaluation and Discussion

Figure 7.6: Box plots of the runtimes for the Ocean (Contiguous Partitions) application for different configurations of the evaluation system. The top row shows the thread distribution in round-robin distribution, whereas the bottom row shows the tile-wise distribution. The columns show the respective caching strategies. Each plot shows the varying execution times (shown on the vertical axis) over a varying degree of parallelism (on the horizontal axis) with box plots performed for $n = 10$ runs.

unit on the SHM tile. On the contrary, the limiting factor for the tile-wise distribution is the cache size.

Furthermore, we can observe here that the ASCS exclusive strategy initially performs better than the SCS and the ASCS read-shared strategy. However, with increasing parallelism, the ASCS read-shared strategy performs better than the SCS and the ASCS exclusive strategies. We conclude that the application performs many read accesses to shared data that is seldom modified.

7.4.9 Water Spatial

This application performs a molecular dynamics N-body problem simulation. The results for this benchmark are shown in Figure 7.7.

The results show that the benchmark experiences good scalability for all variants. However, the variants with tile-wise distribution perform better than those with round-robin distribution. We explain this by the benchmark profiting from cache locality. Furthermore, the ASCS strategies do not provide any benefit over the SCS. This indicates that there is either minimal sharing between the application threads or that the benchmark is not cache bound.
7.4 Evaluation with Real World Applications

Figure 7.7: Box plots of the runtimes for the Water Spatial application for different configurations of the evaluation system. The top row shows the thread distribution in round-robin distribution, whereas the bottom row shows the tile-wise distribution. The columns show the respective caching strategies. Each plot shows the varying execution times (shown on the vertical axis) over a varying degree of parallelism (on the horizontal axis) with box plots performed for \( n = 10 \) runs.

7.4.10 Comparison of the Application Runtimes

Until now, we have considered the three caching strategies concerning different degrees of parallelism and distribution strategies. One question that remains, however, is whether there are general trends when comparing the strategies with one another. With Figure 7.8, we approach this question by comparing the relative execution times for all benchmark configurations. We, therefore, use the SCS as the baseline for our comparison and present the runtimes of the ASCS strategies relative to it.

The ASCS strategies, in general, outperform the SCS considerably. For example, in the case of the Radix application, we see a speedup of up to 2.5 for 32 threads. The exception is the Water Spatial benchmark, for which, in the tile-wise configuration with 16 threads, the ASCS strategies perform worse than the SCS. Recall that this benchmark overall showed for all configurations good scalability and that we concluded that it only had limited shared memory accesses, which would explain why this benchmark can profit from the performance of the SCS. Overall, the degree to which the ASCS strategies improve over the SCS is application dependent.
7 Evaluation and Discussion

Figure 7.8: Charts of the relative execution times under varying software consistency mechanisms. The charts for round-robin and tile-wise distributions for 16 and 32 threads are given. The runtimes are presented relative to the page-cache-only strategy.

Moreover, we can see that it is application dependent on whether the ASCS exclusive or the ASCS read-shared strategy performs better. This stands to reason, as the ASCS read-shared strategy benefits from read-sharing, which may only sometimes be given for an application.

7.4.11 Discussion of the Application Benchmarks

We can now take together the results of the benchmarking applications to draw conclusions regarding the performance of these strategies and their scalability in general. Finally, we also discuss some limitations that apply to the results.

Cross-tile Sharing Determines the Scalability

This observation stands to reason, as this property is inherent to any cache coherency system independent of the implementation: the more parallel units operate on shared data, the more synchronization of cache contents is necessary. The synchronization overheads, thereby, are, on the one hand, determined by the implemented memory consistency model. More relaxed memory models generally allow applications to scale more gracefully. On the other hand,
7.4 Evaluation with Real World Applications

however, these overheads are also determined by the size of the cached objects. Keeping small cache lines in the two-to-three-digit byte ranges in sync is more efficient than doing so for memory pages, as in the SCS.

This observation becomes evident when comparing the results for the SCS variants of the respective benchmark applications. The round-robin strategy outperforms the tile-wise strategy in the LU Contiguous Blocks and Radix benchmarks. In these setups, the tile-wise strategy effectively suffers twice: firstly, it must perform more consistency operations to maintain consistency between the tiles. Secondly, the threads in these configurations have less opportunity to benefit from cache locality. Note that the applied distribution strategies did not attempt to find thread-to-tile mappings that optimize cache locality.

Considering coherence systems, it is advisable to structure applications for tile-based architectures so they hierarchically exploit parallelism along tile boundaries: threads on the same tile thus can share data more frequently, whereas shared data accesses across tile boundaries should happen less often. This way, NoC traffic and coherence overheads could be reduced. Furthermore, such a structure would be additionally beneficial for synchronization, as threads could rely on tile-local synchronization more often instead of the more expensive global synchronization.

Directory Controller Determines Scalability

In the setup for the application benchmarks, the shared memory exclusively resides in the SHM and is consequently handled by a single directory controller. Unfortunately, this single directory controller becomes a bottleneck when the number of tiles that request access to the shared memory rises. We observed this for the FFT, Radix, and Ocean Contiguous Partitions benchmarks. As a result, all these benchmarks initially scale with increasing parallelism but then increase in their runtimes.

Ultimately, the severity at which this effect takes place, of course, depends on the application and the way that it interacts with its working set. For example, the described effect does not emerge for the LU Continuous Blocks and the Water Spatial benchmarks. Nevertheless, some system parameters to tweak could decrease the effect of this bottleneck.

As of now, the benchmarks suppose a release consistency memory model. The SCS, however, provides the more relaxed entry consistency model. When the applications use this consistency model, they would reduce the total number of consistency messages and, thus, possibly, the total number that the directory controller has to handle.

Alternatively, we can reconsider the memory layout in the MPSoC architecture. Currently, a single tile integrates the majority of the system’s DRAM\(^2\). When the tiles share this memory for parallel cross-tile computations, it inevitably becomes a bottleneck. An alternative here is to split the DRAM and locate it on the individual tiles, quite similar to a more classical NUMA approach. This approach has the advantage of allowing the workload distribution to the tiles

\(^2\)A system configuration with two DDR controllers on distinct tiles exists. However, ultimately, the outlined problem persists in this configuration.
following the compute resources. Consequently, remote accesses would be distributed among
multiple tiles, lessening the stress on a single directory controller.

Application Memory Access Patterns Determine Scalability and Performance

This observation is probably the most obvious one. We covered it partially with the previous
two observations: the number of operations to maintain a consistent view of the shared memory
impacts the performance. However, as we have seen with the micro-benchmarks, there are
further costs to the software consistency mechanisms besides those for maintaining consistency.
These are the costs that the caching mechanism induces. For example, as we have seen, capacity
misses can induce substantial runtime costs. However, again, this effect mainly affects the SCS.

Especially applications that repeatedly experience capacity misses can severely affect perfor-
mane. This is particularly evident for the Radix benchmark. As we can see in Figure 7.8, we
can observe almost a speedup of 5 for this benchmark for 16 threads in the tile-wise configura-
tion. This speedup can be ascribed to the cache size of the L2 cache compared to the SCS,
which the benchmark can use in the ASCS strategies, as well as to significantly smaller costs
for capacity misses in the L2 cache.

ASCS Strategies Outperform the SCS

These considerations bring us to the following observation. In Figure 7.1, we compared the
performance of memory accesses for small working set sizes. As a reminder, these results
suggested that the SCS will perform better than the ASCS strategies, as memory accesses to
the TLM are served faster than accesses to the L2 cache. Contrary to these results, however,
we can see in Figure 7.8 that the adaptive caching schemes outperform the SCS for almost all
configurations.

This performance advantage is probably due to two mutually dependent factors: firstly,
the adaptive strategies have more cache memory due to the additional use of the L2 cache.
Although the L2 cache memory does not simply extend the available memory of the page
cache, and data can only be kept in the L2 cache under certain conditions (for example, if
they are accessed exclusively or are read-only), the applications utilizing the ASCS strategies
will therefore experience fewer capacity misses than the SCS strategies. The second factor in
the performance profile is that the L2 cache can handle capacity misses much more efficiently
than the page cache. Furthermore, evicting data from the L2 cache in the ASCS strategies
does not cause costly changes in the memory map. Therefore, even if all cache lines of a page
are evicted from the L2 cache, subsequent access to the page does not necessarily lead to an
expensive page fault.

Limitations and Further Considerations

Finally, there are some additional limitations regarding the implementation and the hardware
platform that should also be considered when interpreting the results.
Firstly, as stated above, unfortunately, only a subset of the benchmarks in the SPLASH-3 benchmark suite was compatible with the evaluation system. Ideally, the software consistency mechanisms should be tested on an even more comprehensive array of applications to tap possible corner cases in memory access patterns and further solidify our findings.

Secondly, we encountered some issues with the hardware. One example is the DMA unit, which affected the fast-writeback mechanism. We have seen in Table 7.10 that the fast-writeback can provide a significant benefit over the diff-based writebacks. This mechanism comes mainly in the SCS into play and can, as such, tip the scales in favor of the SCS.

Moreover, it must be considered that the evaluation system has some characteristics that differ significantly from current computer systems. On the one hand, it is not an example of a modern computer architecture, given the fact that it is based upon SPARC Leon v8 architecture, with a relatively modest 8 MiB of TLM and a shared system clock of 50 MHz. Earlier, in Chapter 4 saw how properties of the platform could affect the overall system performance: there are no selective TLB flushes on the Leon implementation of the SPARC v8. A TLB flush on this architecture flushes all TLB entries. To make matters worse, it additionally also flushes the whole L1 cache. As such, modifications in the page table, which are relatively frequent in the software consistency mechanisms, severely impact the overall system performance. More modern architectures thus provide selective access to these structures. For example, the Intel x86 family provide more fine-grained access to the TLB and caches [18, Sec. 4.10.4.1].

Conversely, however, the evaluation system also provides a unique set of features that are not typically found in current computer platforms, such as the SHARQ (or more generally, the DMA unit), or the hardware scheduling support. The SCS and ASCS make good use of these features.

7.5 Conclusion

In this chapter, we put the SCS and ASCS to test to determine their characteristics. In Section 7.1.1, we started out by determining an acceptance criterion for coherence systems. As we elaborated, any coherence mechanism must eventually be able to provide scalability for parallel shared memory applications.

We, therefore, presented the evaluation results for runtime benchmarks for an array of shared memory parallel applications from the SPLASH-3 benchmark suite. We saw that some of the applications provided good scalability, whereas others did not scale as well. However, we argued that reasonable architectural changes would improve the results for the latter class. In conclusion, we argue that the software consistency mechanism and its caching strategies indeed provide the ability to develop scalable shared memory applications.

However, we also saw that the overall performance depends on the respective application’s workload—an insight likely inherent to all caching and coherence-related topics. We also saw that the directory controller is prone to be a performance bottleneck in the used evaluation setup. As outlined above, however, this issue can likely be addressed by redistributing memory across the tiles’ memory.
A further insight of the evaluation is that the adaptive caching strategies especially provide promising performance results. Even though the L2 cache provides worse access times than the TLM does, these caching strategies benefit from the larger size of the L2 cache and, in addition to that, the lower costs for capacity misses. Furthermore, these strategies likely allow reducing the overall memory footprint of the software consistency mechanisms as, in them, the page cache is only used for shared pages and can thus be decreased in size.
8

Conclusion

This thesis makes a case for an alternative, software-based approach to the problem of cache coherence. Therefore, it proposed the SCS and ASCS software consistency mechanisms realized as operating system components. With their design, these software consistency mechanisms starkly contrast traditional approaches to coherence: firstly, they provide memory consistency, a more expressive concept than merely cache coherence. Secondly, traditional coherence systems are typically realized as hardware components.

The motivation to break with this tradition stems from the increased functional and non-functional demands on today's embedded systems. Especially the requirement for increasing performance drives the further increase in parallelism in these systems, making traditional coherence mechanisms challenging to adapt. Shifting responsibilities for coherent shared memory to the system software layer in this context is advantageous for several reasons:

1. It relieves the hardware layer from the burden of providing coherence. This is especially beneficial in terms of chip area that becomes increasingly sparse.

2. A coherent shared memory mechanism realized in software is more flexible and thus adaptive to application requirements.

3. A coherence mechanism that is conceptually closer to the application can profit from application context knowledge to implement further optimizations.

This chapter will summarize this dissertation's results and provide an outlook to further interesting research perspectives.

8.1 Summary

This dissertation is motivated by the conflicting developments regarding cache coherence in the hardware and software domains: in light of the increasingly parallel processor architectures, hardware developers strive to find cache coherence strategies that are sufficiently fast, chip-area- and energy-efficient to keep up with requirements of modern embedded platforms. Conversely, in the realm of (system-)software, however, coherence mechanisms are regarded as a source of overheads. This observation is especially evident in modern runtime and operating systems that integrate many clever allocation, synchronization, and scheduling strategies that attempt to utilize the coherence system as little as possible. This development becomes especially evident
in today’s NUMA systems. However, realizing a coherent shared memory interface that is only seldom in use in hardware is wasteful in terms of chip area and energy resources. Hence we can polemically summarize this situation as follows: hardware manufacturers strive to provide a feature that software developers avoid as much as possible.

Tile-based architectures such as that envisioned in the InvasIC project on which this dissertation is based, therefore, forgo global coherence. Each tile sees the memory in the system (local and remote) coherently, but several tiles taken together do not see the memory coherently anymore. Consequently, applications in these systems can no longer rely on shared memory to communicate. Instead, they must communicate via message passing. Message passing, however, is not ideal and convenient for all use cases.

This thesis, therefore, presented the SCS as an extension to OctoPOS, the operating system of the InvasIC project. The SCS is a software mechanism that compensates for the lack of a hardware coherence system by providing a memory-consistent shared memory abstraction.

The SCS is realized as a VSM system and follows, therefore, the principle mode of operation that most VSM systems apply: it implements a paging-based virtual memory that provides a unified abstraction over local and remote pages. The system thus introduces a virtual memory abstraction across the memory in the system, which provides each memory with the same consistency guarantees. Internally, the SCS, therefore, uses MMUs to track accesses to remote memory. Upon such an access, it then replicates the remote memory page into a page cache. Thus, maintaining a consistent shared memory becomes the task of maintaining consistency for the memory in the page caches. Accordingly, the SCS provides memory fences and tracks per page sharing information in a cache directory. When encountering a memory fence, the SCS uses this information to write back cached pages back or to invalidate them from the cache.

In order to reduce the number of costly cache operations, the SCS thereby realizes the entry consistency memory model. This consistency model allows the SCS furthermore to realize several optimization strategies, such as a diff mechanism to allow concurrent write access to shared pages and thus reduce false sharing. Additionally, the SCS performs writeback operations lazily. A fast writeback mechanism furthermore attempts to avoid the diff calculations when possible.

In addition to the purely paging-based technique, this thesis furthermore explored a novel adaptive caching approach for the SCS. This technique is distinctive of both traditional hardware coherence mechanisms and VSM systems, as it uses both hardware and software caches and switches adaptively between them. Therefore, the adaptive caching strategy uses the already available sharing information to switch between page and hardware cache, as necessary adaptively. More precisely, the ASCS attempts to make use of the hardware cache for read-only data and data that is exclusively accessed by a single tile. The page cache thus becomes a fall-back option whenever pages are shared between multiple tiles. This approach is advantageous as it reduces overall memory consumption and the total number of computationally expensive page cache operations.

Furthermore, this thesis demonstrates with the runtime system for shared memory programming how an integration of the SCS and ASCS mechanisms into parallel programming interfaces...
can look. This runtime system provides the means to create parallel fork-join applications for the InvasIC platform and implicitly takes care to maintain memory consistency at synchronization points. The programmer can parameterize the synchronization interfaces via memory hints to ensure memory consistency and only execute the necessary memory fences.

Finally, this thesis presented the experimental evaluation of the SCS and ASCS under micro-benchmarks and application benchmarks. For the latter, benchmarks from the SPLASH-3 benchmark suite were used. The main focus of this evaluation thereby was whether the shared memory interface of the software consistency mechanisms provides scalability for parallel applications. One result of the evaluation thereby is that applications using software consistency mechanisms in the InvasIC platform indeed can exploit parallelism across multiple tiles. However, limitations apply: the InvasIC prototype realizes its shared memory as off-chip DDR that is connected to a single tile. As a result, some of the experiments experienced significant slowdowns as the directory and DMA engine on this tile became a bottleneck. An alternative memory organization for the off-chip memory would likely remedy the problem, for example, with multiple per-tile DRAM modules as a supplement for the TLM instead of a single shared memory. Another outcome of the evaluation is that the adaptive caching strategies outperform the approach that utilizes only the page cache considerably. Especially working sets that experience many compulsory cache misses can benefit from these adaptive caching strategies. We could demonstrate that the adaptive approach of the ASCS resulted in an overall speedup in all but one test case, as compared to the non-adaptive variant of the SCS.

8.2 Outlook

With the main matter of this thesis summed up, what now remains is to look ahead. On the one side, there are some topics that this thesis did not cover. These topics had to be excluded for practical reasons and reasons of feasibility. Conversely, as the research world keeps turning, new fields and topics of interest arise, which would be interesting to incorporate with this thesis’s results. The following gives an overview of these topics and thereby sketches possible open research directions building upon the SCS and ASCS.

Although we expect hardware-based coherence mechanisms to generally outperform software coherence mechanisms like the one presented in this thesis, it would be interesting to understand better how far the divide concerning performance and resource consumption between these approaches is. We, therefore, identified three topics of interest: firstly, the issue of the comparability of performance and scalability numbers between hardware coherence mechanisms and software consistency mechanisms such as the SCS or ASCS. Secondly, the topic of how the software consistency mechanisms compare in terms of energy consumption. Finally, the impact of not having a coherence system in hardware on the complexity of the chip logic. The first two questions can likely be approached for a first estimate utilizing hardware simulation, such as gem5 [Bin+11]. The challenging problem is finding a suitable architecture that allows a fair comparison of coherence mechanisms. The third topic is best to be answered by chip
8 Conclusion

manufacturers themselves. Alternatively, this issue can best be approximated by comparing the results of FPGA synthesis.

Another aspect is the evaluation of the software consistency mechanisms on more modern computer architectures. While the InvasIC prototype, in many regards, provides a suitable reference that integrates several unique and novel architectural aspects, some of its components are dated. One example of this is the SPARC LEON v8 core and its expensive TLB shootdown mechanism that we discussed in Section 4.3. Therefore, it would be interesting to see whether the observed characteristics can still be observed. Conversely, both SCS and ASCS make use of unique features of the InvasIC prototype, and as such, it would be interesting how they performed in the absence of these features.

A modern hardware architecture would probably also allow investigating of how the mechanisms could benefit from different types of hardware accelerators. Both, SCS and ASCS, already make use of the DMA engine in the InvasIC prototype to copy memory pages. Still, more tasks in the mechanism could be offloaded to dedicated hardware. One example where such an accelerator would be helpful is the diff operations. These operations happen frequently and are, by their underlying data processing pattern, ideally suited for offloading to dedicated hardware. For example, Intel recently announced its data streaming accelerator [Int21]. Among others, this accelerator implements operations such as nulling and copying memory blocks and performing diff calculations. It thus provides all the basic building blocks to offload the diff mechanism to hardware.

Finally, the memory abstraction layer can be extended featurewise. By providing different memory consistency models depending on the application’s needs, the practicability of the suggested software consistency models would significantly improve. The concept of memory abstraction is thereby not limited to unifying consistency alone. With the increasing diversification of the memory hierarchy, for example, by varying degrees of size, memory access speed, and even memory volatility, a generalized virtual memory concept could help unify these concepts and make them better accessible. Ideally, the resulting interface should be flexible enough to dynamically adapt to the application requirements.

8.3 Closing Remarks

Software and hardware are changing rapidly. These changes are, on the one side, driven by novel, exciting workloads and fields of application, such as artificial intelligence. These workloads require tremendous computational power. On the other side, these changes are driven by sheer necessity as manufacturers approach the limits of what is physically possible.

At the core of this thesis lies the thought of how hardware and system software together can adapt to these changes. In the case of this thesis, we considered the topic of coherence: instead of maintaining coherence at the hardware level, it is worthwhile to move it to the software layer and make it thus more flexible. In many regards, it is worthwhile to reconsider the problems that are classically resolved at the software or the hardware layer, especially in light of current technical developments.
The Invasive Execution Model

When refraining from time-sharing processing resources, the resource management strategy in an InvasIC system lends itself to an execution model that supports microparallelism. In this model, applications can use parallelism even for relatively short code paths, such as loop statements. As mentioned in Section 2.1.1 in the InvasIC system, a specialized hardware unit, the CiC, takes the place of the scheduler. The only task left for the OctoPOS kernel is to execute the i-lets.

The operating system, therefore, realizes an execution model in which it executes control flows run-to-completion, meaning that once an i-let is dispatched, it relinquishes control over the processor only explicitly to wait for external events or, upon completion. Doing so allows for reducing the number of context switches significantly. Moreover, it allows the kernel to manage even large numbers of i-lets in a relatively cost-efficient way: for non-blocking i-lets, it suffices to provide a single execution context per processing element, as multiple i-lets can share a single context. The implementation of i-let execution thus comes down to calling the function embedded in the i-let. Once the i-let execution is finished, the execution context can be repurposed for the execution of the following i-lets. An overview of this execution model is shown in Figure A.1.

So far, we have considered the execution of i-lets in isolation. However, applications typically consist of multiple i-lets that additionally require means to synchronize with one another. Typically, control flow synchronization entails blocking one or multiple of them until an event occurs. Operating systems commonly implement synchronization primitives by switching from the blocking control flow to another control flow ready for execution. The blocking control flow is then stored away until its blocking condition has been resolved. OctoPOS realizes synchronization with the signal primitive. This primitive allows waiting for one or multiple signalizations.

In the OctoPOS execution model, multiple i-lets share a single context. We can, therefore, not simply switch to another i-let, as the context cannot be reused. To execute further i-lets, a new execution context must be allocated and initialized. This call to the allocator adds runtime overhead. In addition to the runtime costs, keeping multiple blocked contexts is expensive memory-wise.

OctoPOS therefore provides an alternative synchronization primitive, the infecting signal. Instead of explicitly blocking a control flow, it is equivalent to shifting the parts of the control flow after the signal into a separate i-let. By doing so, dispatching the new i-let can be delayed up
until the point when the infecting signal is triggered, and the need to block execution contexts is mitigated. To this end, an infecting signal instance consists of a configurable signalization counter and a handler $\ell$-let. Other $\ell$-lets may signal on an infecting signal instance, thereby reducing the signalization counter. Once the counter reaches zero, the handler $\ell$-let is spawned.

This synchronization style requires a programming style that structures the parallel control flows along signalizations instead of explicitly blocking for events. Both programming patterns are equally expressive. However, the signaling style requires the variables that are used across different control flows to be explicitly passed via the heap. These variables would typically be passed implicitly via the stack in the blocking style. While the signaling pattern is more verbose than its blocking counterpart, it reduces the number of context switches. It thus allows the operating system to optimize the total memory usage, as the operating system must only provide call stacks for those control flows currently in execution and control flows that are blocking.

It should be mentioned that OctoPOS also provides blocking signals. Blocking signals have the same semantics as infecting signals. However, instead of spawning a new $\ell$-let, these signals block the calling $\ell$-let until triggered. The blocking signals should be understood as a means of convenience and are better avoided. As described above, blocking the current execution context means that OctoPOS must allocate, initialize and switch to a new context. These operations entail additional runtime overhead and memory costs. Figure A.1 shows on the right-hand side how the context of a blocking $\ell$-let is replaced with a free context for the execution of the following $\ell$-let.
Bibliography


136
Bibliography


Bibliography


Bibliography


# Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iRTSS</td>
<td>Invasive Runtime Support System</td>
</tr>
<tr>
<td>AHB</td>
<td>Advanced High-performance Bus</td>
</tr>
<tr>
<td>AMBA</td>
<td>Advanced Microcontroller Bus Architecture</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ASCS</td>
<td>Adaptive Software Consistency System</td>
</tr>
<tr>
<td>CiC</td>
<td>Core i-corelet Controller</td>
</tr>
<tr>
<td>CoW</td>
<td>Copy on Write</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DDR</td>
<td>Double Data Rate</td>
</tr>
<tr>
<td>diff</td>
<td>Difference Set</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DRAM</td>
<td>Dynamic RAM</td>
</tr>
<tr>
<td>DSM</td>
<td>Distributed Shared Memory</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GCC</td>
<td>GNU Compiler Collection</td>
</tr>
<tr>
<td>HBM</td>
<td>High Bandwidth Memory</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>InvasIC</td>
<td>Invasive Computing</td>
</tr>
<tr>
<td>IO</td>
<td>Input/Output</td>
</tr>
<tr>
<td>IPI</td>
<td>Inter-Processor Interrupt</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>LRU</td>
<td>Least Recently Used</td>
</tr>
<tr>
<td>MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>MOESI</td>
<td>Modified Owned Exclusive Shared Invalid</td>
</tr>
</tbody>
</table>
### Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
</tr>
<tr>
<td>MPSoC</td>
<td>Multiprocessor System-on-Chip</td>
</tr>
<tr>
<td>NA</td>
<td>Network Adapter</td>
</tr>
<tr>
<td>NoC</td>
<td>Network-on-Chip</td>
</tr>
<tr>
<td>NUCA</td>
<td>Non-Uniform Cache Access</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-Uniform Memory Access</td>
</tr>
<tr>
<td>PGAS</td>
<td>Partitioned Global Address Space</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SCS</td>
<td>Software Consistency System</td>
</tr>
<tr>
<td>SHARQ</td>
<td>Software-Defined Hardware-Managed Queues</td>
</tr>
<tr>
<td>SHM</td>
<td>Shared Memory</td>
</tr>
<tr>
<td>SMP</td>
<td>Symmetric Multiprocessing</td>
</tr>
<tr>
<td>SRAM</td>
<td>Static RAM</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>TLM</td>
<td>Tile Local Memory</td>
</tr>
<tr>
<td>TSO</td>
<td>Total Store Order</td>
</tr>
<tr>
<td>VSM</td>
<td>Virtual Shared Memory</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive Or</td>
</tr>
</tbody>
</table>