Aspects of Code Generation and Data Transfer Techniques for Modern Parallel Architectures

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von

Manuel Mohr

aus Heilbronn

Tag der mündlichen Prüfung:	4. Juli 2018
Erster Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweiter Gutachter:	Prof. Dr.-Ing. Jürgen Teich
Dritter Gutachter:	Prof. Dr. rer. nat. Sebastian Hack

Contents

Prose is architecture and the Baroque age is over.

Ernest Hemingway

Abstract

The focus of hardware architecture development has shifted from striving for ever higher clock frequencies towards incorporating an ever increasing number of cores on a single chip. A high number of cores makes it possible to offer a mixture of weak and strong cores, and even specialized cores with completely different instruction sets. This makes development for such a heterogeneous platform challenging and requires adequate support by tools, such as compilers. Besides their core structure, there is a second dimension to these architectures: memory. A major obstacle to scalability regarding the memory hierarchy of many-core platforms is maintaining global cache coherence. Hardware coherence protocols either scale poorly, or are complex and often suffer from performance and power overheads. Abandoning global cache coherence is a radical solution to this problem. However, efficiently mapping programming models to hardware with relaxed guarantees is challenging. In this dissertation, we make contributions to compilation techniques targeting both dimensions of modern parallel architectures: memory and core structure.

The first part of this dissertation concerns data transfer techniques for non-cache-coherent architectures. Such non-cache-coherent shared-memory

architectures provide a shared physical address space, but do not implement hardware-based coherence between all caches of the system. Logically partitioning the shared memory offers a safe way of programming such a platform. In general, this creates the need to copy data between memory partitions.

We study the compilation to invasive architectures, a family of non-cache-coherent many-core architectures. We investigate the efficient implementation of data transfers for both simple and complex data structures on these architectures. Specifically, we propose a novel approach to copy complex pointered data structures without the need for serialization. To this end, we generalize object cloning to work in the presence of non-coherent caches by extending object cloning with compiler-directed automatic software-managed coherence. We present implementations of multiple data transfer techniques in an existing compiler and runtime system. We extensively evaluate these implementations on an FPGA-based prototype of an invasive architecture. Finally, we propose adding hardware support for range-based cache operations, and describe and evaluate possible implementations and overheads.

The second part of this dissertation concerns code generation techniques to accelerate shuffle code by using permutation instructions. Shuffle code arises during register allocation, where the compiler maps program variables to machine registers. The compiler may introduce shuffle code, consisting of copy and swap operations, to transfer data between registers. Depending on the quality of register allocation and the number of available registers, a large amount of shuffle code may be generated.

We propose to speed up the execution of shuffle code by using permutation instructions that arbitrarily permute the contents of small sets of registers in one clock cycle. To show the feasibility of this idea we first present an extension of an existing RISC instruction set with permutation instructions. We then describe how to implement the proposed permutation instructions in an existing RISC architecture. Subsequently, we develop two code generation schemes that exploit permutation instructions to implement shuffle code: a fast heuristic and a dynamic-programming-based approach. We formally prove quality and correctness properties of both approaches and show the latter approach to be optimal. In the following, we implement both code generation algorithms in a compiler and extensively evaluate

and compare their code quality using a standardized benchmark suite. We first measure precise dynamic instruction counts, which we then validate by measuring running times on an FPGA-based prototype implementation of the proposed RISC architecture with permutation instructions. Finally, we argue that permutation instructions are cheap to implement on modern out-of-order architectures that already support register renaming.

Optimierung ist, wenn es
manchmal nicht schlechter wird.

Lehrstuhlweisheit, nach Rubino Geiß

Zusammenfassung

Im Bereich der Prozessorarchitekturen hat sich der Fokus neuer Entwicklungen von immer höheren Taktfrequenzen hin zu immer mehr Kernen auf einem Chip verschoben. Eine hohe Kernanzahl ermöglicht es unterschiedlich leistungsfähige Kerne anzubieten, und sogar dedizierte Kerne mit speziellen Befehlssätzen. Die Entwicklung für solch heterogene Plattformen ist herausfordernd und benötigt entsprechende Unterstützung von Entwicklungswerkzeugen, wie beispielsweise Übersetzern. Neben ihrer heterogenen Kernstruktur gibt es eine zweite Dimension, die die Entwicklung für solche Architekturen anspruchsvoll macht: ihre Speicherstruktur. Die Aufrechterhaltung von globaler Cache-Kohärenz erschwert das Erreichen hoher Kernzahlen. Hardwarebasierte Cache-Kohärenz-Protokolle skalieren entweder schlecht, oder sind kompliziert und führen zu Problemen bei Ausführungszeit und Energieeffizienz. Eine radikale Lösung dieses Problems stellt die Abschaffung der globalen Cache-Kohärenz dar. Jedoch ist es schwierig, bestehende Programmiermodelle effizient auf solch eine Hardware-Architektur mit schwachen Garantien abzubilden.

Der erste Teil dieser Dissertation beschäftigt sich Datentransfertechniken für nicht-cache-kohärente Architekturen mit gemeinsamem Speicher.

Diese Architekturen bieten einen gemeinsamen physikalischen Adressraum, implementieren aber keine hardwarebasierte Kohärenz zwischen allen Caches des Systems. Die logische Partitionierung des gemeinsamen Speichers ermöglicht die sichere Programmierung einer solchen Plattform. Im Allgemeinen erzeugt dies die Notwendigkeit Daten zwischen Speicherpartitionen zu kopieren.

Wir untersuchen die Übersetzung für invasive Architekturen, einer Familie von nicht-cache-kohärenten Vielkernarchitekturen. Wir betrachten die effiziente Implementierung von Datentransfers sowohl einfacher als auch komplexer Datenstrukturen auf invasiven Architekturen. Insbesondere schlagen wir eine neuartige Technik zum Kopieren komplexer verzeigerter Datenstrukturen vor, die ohne Serialisierung auskommt. Hierzu verallgemeinern wir den Objekt-Klon-Ansatz mit übersetzergesteuerter automatischer software-basierter Kohärenz, sodass er auch im Kontext nicht-kohärenter Caches funktioniert. Wir präsentieren Implementierungen mehrerer Datentransfertechniken im Rahmen eines existierenden Übersetzers und seines Laufzeitsystems. Wir führen eine ausführliche Auswertung dieser Implementierungen auf einem FPGA-basierten Prototypen einer invasiven Architektur durch. Schließlich schlagen wir vor, Hardwareunterstützung für bereichsbasierte Cache-Operationen hinzuzufügen und beschreiben und bewerten mögliche Implementierungen und deren Kosten.

Der zweite Teil dieser Dissertation befasst sich mit der Beschleunigung von Shuffle-Code, der bei der Registerzuteilung auftritt, durch die Verwendung von Permutationsbefehlen. Die Aufgabe der Registerzuteilung während der Programmübersetzung ist die Abbildung von Programmvariablen auf Maschinenregister. Während der Registerzuteilung erzeugt der Übersetzer Shuffle-Code, der aus Kopier- und Tauschbefehlen besteht, um Werte zwischen Registern zu transferieren. Abhängig von der Qualität der Registerzuteilung und der Zahl der verfügbaren Register kann eine große Menge an Shuffle-Code erzeugt werden.

Wir schlagen vor, die Ausführung von Shuffle-Code mit Hilfe von neuartigen Permutationsbefehlen zu beschleunigen, die die Inhalte von einigen Registern in einem Taktzyklus beliebig permutieren. Um die Machbarkeit dieser Idee zu demonstrieren, erweitern wir zunächst ein bestehendes RISC-Befehlsformat um Permutationsbefehle. Anschließend beschreiben wir, wie die vorgeschlagenen Permutationsbefehle in einer bestehenden RISC-Architektur implementiert werden können. Dann entwickeln wir zwei Verfahren zur Codeerzeugung, die die Permutationsbefehle ausnutzen, um Shuffle-Code zu beschleunigen: eine schnelle Heuristik und einen auf dynamischer Programmierung basierenden optimalen Ansatz. Wir beweisen Qualitäts- und Korrektheitseingeschaften beider Ansätze und zeigen die Optimalität des zweiten Ansatzes. Im Folgenden implementieren wir beide Codeerzeugungsverfahren in einem Übersetzer und untersuchen sowie vergleichen deren Codequalität ausführlich mit Hilfe standardisierter Benchmarks. Zunächst messen wir die genaue Zahl der dynamisch ausgeführten Befehle, welche wir folgend validieren, indem wir Programmlaufzeiten auf einer FPGA-basierten Prototypimplementierung der um Permutationsbefehle erweiterten RISC-Architektur messen. Schließlich argumentieren wir, dass Permutationsbefehle auf modernen Out-Of-Order-Prozessorarchitekturen, die bereits Registerumbenennung unterstützen, mit wenig Aufwand implementierbar sind.

Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.

Douglas Hofstadter

Acknowledgments

First, I wish to thank my advisor Prof. Gregor Snelting for his support and the opportunity to pursue my own interests without pressure. I also thank him for shielding me and his whole group from the many adversities of academic life, such as the need to secure a steady stream of money. He provided an environment where it was possible to concentrate on research, on building efficient and robust software, as well as on excellence in teaching, which is a luxury one becomes accustomed to far too easily. Next, I would like to thank Prof. Jürgen Teich for reviewing this dissertation. I also thank him for founding the research project Invasive Computing, which taught me a great deal about hardware, software, the many things that can go wrong between them—and how great it is when they finally work together. The first part of this dissertation would not have been possible without this research project. Moreover, I want to thank Prof. Sebastian Hack for serving as the third reviewer of this dissertation. I also thank him for leaving an inconspicuous footnote in his dissertation, which ultimately gave rise to the second part of this work.

Next, I have to thank the (former and current) machine code connoisseurs from the compiler group in Karlsruhe, namely Matthias Braun, Sebastian

Buchwald, Andreas Fried, and Andreas Zwinkau. I especially thank my former office inmate Matthias Braun for being a walking encyclopedia of Firm and x86 peculiarities and for sharing his knowledge with me. Furthermore, I thank Sebastian Buchwald for his tireless commitment to correctness and clarity of expression in both code and written text. And also for removing all trailing whitespace[1]. I thank Andreas Zwinkau for joining me in the quest to bring the invasive prototype system to life while at the same time keeping me up to date on every development in the world of programming languages. Lastly, I thank Andreas Fried for being a very knowledgable office mate, and for creating the nerdiest and most difficult crossword puzzle I ever failed to solve. All compiler constructors were always available for help and technical discussions. Without them, countless hours of staring at Firm graphs and assembly dumps would have been much more boring.

However, our group consisted of more than the pointer arithmeticians in the compiler group. Thus, I also thank all my context-sensitive colleagues from the JOANA group, namely Simon Bischof, Jürgen Graf, Martin Hecker, and Martin Mohr. In particular, I thank Simon Bischof for finding an embarrassing number of bugs in our compiler lab reference compiler. I thank Jürgen Graf for annual barbecues on his panorama terrace and being a close (pun intended) friend. I thank Martin Hecker for his dedication to improving the quality of our teaching material and for regularly destroying half-baked or unfair exam question proposals. And I thank Martin Mohr for his quirky humorous remarks and his love for everything at the bottom of the movie barrel. May all your wishes happen in parallel.

Moreover, I thank all side-effect-free purists from our automated theorem proving group, namely Joachim Breitner, Andreas Lochbihler, Denis Lohner, Sebastian Ullrich, and Maximilian Wagner. I thank Joachim Breitner for producing new ideas faster than I could follow the previous ones, and for proving that days do have more than 24 hours for some people. I thank "Altgesell" Andreas Lochbihler for taking me under his wing back when I started as a doctoral researcher, and also for letting all our dissertations seem short in comparison. I thank Denis Lohner for his outstanding organizational skills and for his arcane knowledge of AFS and other technological oddities in our infrastructure. Lastly, I thank Sebastian

[1] I broke one ligature in this section on purpose, did you spot it?

Ullrich and Maximilian Wagner for humiliating us and our compiler lab reference compiler in front of everyone, and for convincing me that it is possible to write beautiful Scala code.

Technically, I ought to mention everyone I met via the Invasive Computing project, but this list would be too long. Hence, I have to pick some subset and apologize in advance to everyone I do not mention. Thanks to the operating systems group in Erlangen, namely Gabor Drescher, Christoph Erhardt, Sebastian Maier, Benjamin Oechslein, Jens Schedel, and Florian Schmaus for joining us in valiantly defending the system-software layer against evil feature-request deniers from the hardware down below and against blissfully ignorant application developers from above. Project C1 set the bar high for system software quality, just like it should be. I also thank Lars Bauer and Artjom Grudnitsky for realizing the slightly crazy register permutation idea and answering all my stupid questions about hardware. And I thank Stephanie Friederich, Jan Heißwolf, Sven Rheindt, and Aurang Zaib for bringing our hardware platform to life. Special thanks go to Sven Rheindt, who exhumed an ancient hardware design from its grave in a decommissioned Subversion repository and added some new functionality to it, just so I could improve my evaluation. Without people like the ones I just mentioned, who went the extra mile and sometimes traded fewer publications for more fixed bugs, the Invasive Computing project would not have come as far as it did. They all made the countless hours of fighting Scheinzwerge and Heisenbugs much more worthwhile.

Furthermore, I thank Carsten Tradowsky for throwing student after student at our project of extending hardware with range-based cache operations. He mastered the art of delegating tasks.

I thank Ignaz Rutter for his tremendous help with formalizing and investigating the theoretical aspects of shuffle-code generation. Back in May 2012, I innocently walked into his office with my little problem of generating shuffle code, expecting him to point me to some existing paper or book chapter. Somehow, a few months later, I had pages and pages of lemmas and proofs, and also quite some trouble still recognizing my own problem. It was a pleasure witnessing him working his magic while being able to contribute a bit of my own. I also thank him for initiating weekly Ricochet Robots rounds years ago.

Of course, I must mention all students who contributed to software or hardware projects that I used. One of the luxuries of working at a university is the large pool of talented and highly motivated students I could draw from. Hence, I thank Eduard Frank, Jonas Haag, Christoph Jost, Tobias Kahlert, Tobias Modschiedler, Julian Oppermann, Tobias Rapp, Bernhard Scheirle, Martin Seidel, and Philipp Serrer for their contributions.

I also thank all hard-working proof readers, who ploughed through hundreds of pages and found issues both small and large. Namely my helpers were Sebastian Buchwald, Christoph Erhardt, Andreas Fried, Marina Mohr, Martin Mohr, Maximilian Wagner, and Andreas Zwinkau.

Moreover, I thank my parents as well as my sister Marina for their unconditional support of whatever decision I made and whatever task I set my mind on. They always encouraged me to pursue my interests and not to be afraid of taking on challenges. I especially thank my father for denying me my wish for a VTech learning computer and instead putting a real PC into my room; something I assume very few nine year olds had at that time. This sparked my interest in computers and programming and I benefit from this decision to this day.

Finally, I thank Eva for supporting and enduring me during the past years. In her, I have always found both an attentive listener as well as a keen observer. I highly value her advice, as she is right more often than I like to admit. While working on this pamphlet, I have read my share of dissertations in search of inspiration and almost everyone acknowledges the many ups and downs that working in solitude on a document of such size entails. Little did I know how high the ups can be—and how deep the downs. However, I could always count on her support, for which I was and am extremely grateful.

After such an introduction, I can hardly wait to hear what I'm going to say.

Evelyn Anderson

1 Introduction

During the last decade, the computer architecture landscape has changed dramatically. Up until circa 2005, processor designers focused on improving single-thread performance. Moore's Law [Mac11] provided an ongoing miniaturization of transistors, enabling more logic per chip area, while at the same time Dennard scaling [Den+74] allowed to operate these transistors at decreasing voltages and currents.

These advances in chip manufacturing enabled the three main drivers behind faster execution of sequential code: (i) higher clock speeds, i.e., finishing more clock cycles in the same amount of time, (ii) larger caches, i.e., the ability to keep more data close to the core for fast access, and (iii) architectural improvements, i.e., doing more work per clock cycle. The architectural improvements mainly aimed at exploiting instruction-level parallelism (ILP) [HP11, chapter 2]. This included techniques such as prediction of branches in the control flow; dynamic scheduling of instruction streams (also known as out-of-order execution); and speculative execution. Overall, this led to increasingly complex processors.

Then, around 2005, Dennard scaling started to break down. Now it was no longer possible to lower transistor voltages and currents to compensate for increased power usage due to higher frequencies. Hence, clock frequencies started to stagnate while Moore's Law still continued to supply processor designers with higher transistor densities and therefore chip area for additional logic. As instruction-level parallelism was already well exploited, computer architects started putting multiple cores onto a single chip.

The resulting homogeneous multicore architectures included multiple copies of the same complex core that had before powered a single-core processor. However, programs could not exploit the added computational resources of such multicore processors as easily as before. It now became necessary to write parallel programs that distribute their workload across multiple cores.

Once an application splits its work into separate tasks, it quickly becomes clear that not every task requires the same hardware capabilities. For example, for some tasks, the speedup obtained by exploiting instruction-level parallelism on the hardware level is not worth the added hardware complexity. Here, it can be more beneficial to spend the chip area to provide multiple simple cores instead of a single complex core. These simpler cores are still able to run the same code (i.e., they support the same instruction set), but trade sequential execution speed for a smaller area footprint, enabling more parallelism. Hence, such heterogeneous architectures offer different types of cores suitable for different types of tasks.

There can be different degrees of heterogeneity in an architecture. Offering cores with the same instruction set able to execute the same programs is the lowest degree of heterogeneity. Taking this idea further, some architectures provide completely different and specialized cores. These specialized cores may use different instruction sets and may not even be able to run general-purpose programs. However, they can provide superior throughput or energy efficiency for certain parallel tasks.

Hence, one dimension to modern parallel architectures is their core diversification: they not only incorporate many cores, but may also provide cores with different performance characteristics or even instruction sets. Some cores are small and highly specialized, but excel at energy efficiency or throughput for parallel workloads. Other cores are big and complex, but execute sequential program parts with high speed. The resulting heterogeneous multicore architectures provide vast computational resources in principle.

At the same time, there is a second dimension to the developments in the context of modern hardware architectures: the memory hierarchy. Single-core processors had a simple memory structure, where a single memory supplied data to the single core. To hide memory access latency and exploit spatial as well as temporal locality of memory accesses, these architectures included one or multiple levels of caches between core and memory.

Early multicore systems continued to use a single memory. Here, the hardware provides a shared physical address space. All cores can load and store values to that address space, which is backed by the single memory. To reduce access frequency to the main memory, architectures often also include per-core private caches. However, giving each of the cores in such a multicore system its own cache created a new problem: the possibility of accessing stale data due to outdated data copies in caches. If core c_1 has a copy of some data item in its cache and another core c_2 changes that data item in the main memory, core c_1 now has a stale copy of that item in its cache. If c_1 is not notified in some way, it will operate on an out-of-date copy; we say that the situation has become *incoherent*. In order to prevent such incoherent situations, multicore designs settled on implementing hardware-based protocols to keep caches coherent. These protocols thus make caches as functionally invisible as caches in a single-core system [SHW11].

In multicore systems with a single memory, memory access is uniform because distance, and therefore latency, to the memory is the same for every core in the system. As the number of cores further increased, soon a single memory was not able to satisfy the bandwidth requirements by the higher number of cores any more. Therefore, computer architects introduced physically distributed memory, i.e., multiple memories, while still providing a shared address space. This added a notion of locality: from the view of a particular core, there was now a notion of "local" and "remote" memory, with local memory being physically closer and offering, in general, lower access latency and higher bandwidth. Therefore, these systems are also known as non-uniform memory architectures. Their non-uniformity created new challenges. Suddenly, it mattered where data is placed in the memory and it can even be beneficial to copy data to more local memory to avoid frequent more expensive remote accesses.

However, the increasing number of cores and the existence of distributed memory made it more difficult to keep caches coherent. Distributed memory is often used in conjunction with more complicated interconnects between cores, making the implementation of hardware-based coherence protocols considerably more complex. Additionally, overhead related to coherence often grows superlinearly with the number of cores [Kum+11]. This "coherence wall" [Kum+11] has led to the design of non-cache-coherent shared-memory architectures. These architectures still provide a shared address space for all cores in the system; however, they do not guarantee coherent caches on a hardware level. Thus, they remove a major factor that may limit scalability to higher core counts. Yet, caches are now not functionally invisible any more. Therefore, the software, on some level, needs to be aware of the caches and may have to manage coherence itself.

Alternatively, instead of offering a shared physical address space without hardware-based cache coherence, it is also possible to give up the shared address space altogether. Such architectures offer separate physical address spaces, i.e., there are memory locations that are only accessible by a subset of all cores in the system. Hence, such architectures require copying data between memories in order to make it accessible to cores associated with distinct address spaces.

In summary, we have identified two important dimensions of heterogeneous parallel architectures: cores and memory. Figure 1.1 shows the design space spanned by these two dimensions with the characteristics we identified for each. We see a variety of cores, ranging from complex cores, well suited to execute sequential parts of a program by exploiting instruction-level parallelism, to simpler and highly specialized cores that provide higher integration density. Regarding the memory dimension, as we move to the right, we see that the hardware gradually relaxes guarantees to improve scalability. In general, moving up and right in this design space offers higher energy efficiency and more parallelism.

For both dimensions, the compiler plays a key role in the efficient usage of such heterogeneous multicore platforms. Regarding the core diversity, the compiler needs to generate code tailored to the respective core's capabilities. While complex cores extract some parallelism automatically on the hardware level, code generation is still challenging. Due to their

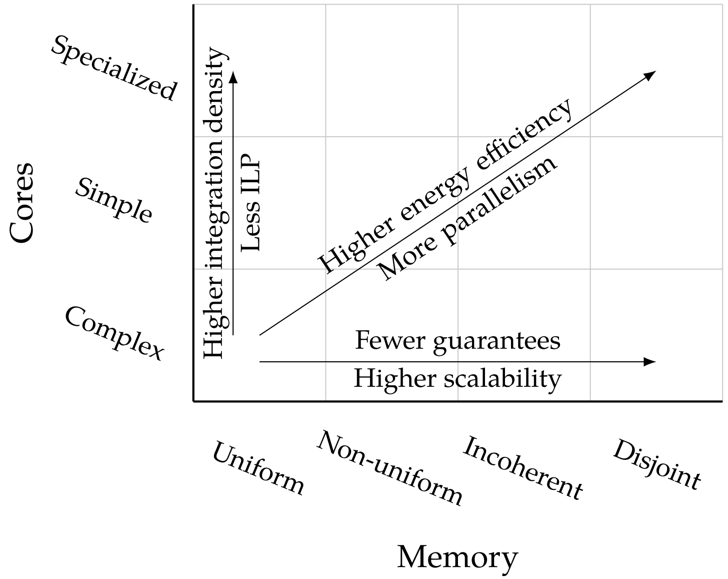


Figure 1.1: A possible design space of modern parallel hardware architectures. Depiction based on Sutter [Sut12].

complicated microarchitecture and execution behavior, deriving cost models to guide compiler code generation is difficult. Furthermore, large instruction sets significantly increase the number of possible encodings for constructs in the source program. In contrast, simpler cores have more predictable performance characteristics. However, they are more dependent on the compiler generating good code in the first place.

Regarding the memory architecture, the compiler must efficiently map the parallel programming model used by an application to the hardware. Due to relaxed hardware guarantees, the compiler may need to do additional work to bridge the gap between guarantees expected by the programmer and those actually provided by the hardware. Moreover, a more complicated memory structure may lead to the usage of different programming models, creating new optimization challenges.

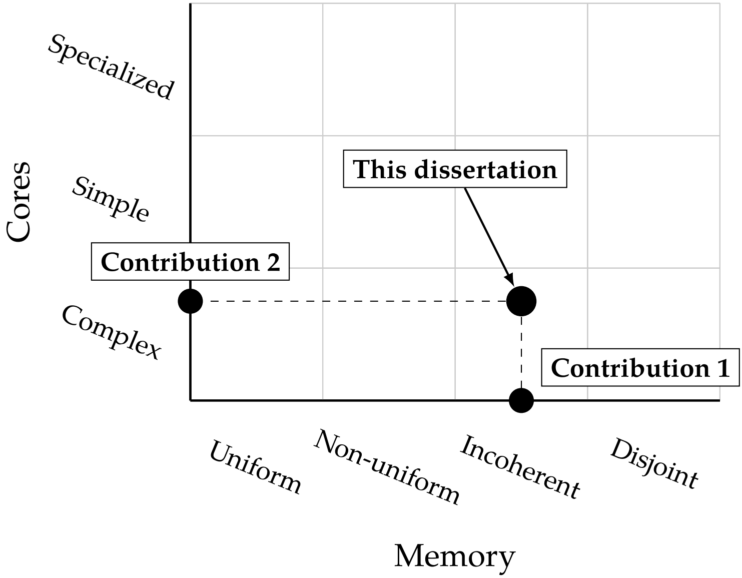


Figure 1.2: The point in the design space of modern parallel architectures targeted by compilation techniques presented in this dissertation. We make contributions in both dimensions.

1.1. Contributions

This dissertation investigates compilation and code-generation techniques for modern parallel architectures. Figure 1.2 shows the point targeted by this dissertation in the architecture design space that we identified in the previous section. This dissertation makes contributions along both axes. More specifically, we investigate

1. compilation to invasive architectures, a familiy of non-cache-coherent shared-memory architectures; and

2. code generation in the context of out-of-order processors.

We give a brief introduction to each topic, before we state our technical contributions.

Non-cache-coherent shared memory. Shared-memory architectures offer a single shared address space. Here, cores communicate by reading from and writing to a shared address space. These systems usually add caches to hide memory latencies and reduce memory traffic by exploiting temporal and spatial locality of data. However, caches create the potential for incoherent situations, i.e., the possibility of accessing stale data.

The standard solution to prevent incoherence is to implement a hardware cache coherence protocol to keep caches coherent. Simple coherence protocols do not scale well with increasing core count. While more complex protocols scale better, they may cause complexity and power issues. This scalability problem is known as the "coherence wall" [Kum+11].

Non-cache-coherent architectures represent a radical solution to circumvent the coherence wall. These architectures do not provide hardware-based cache coherence between all caches of the system. This raises the question of how to program such machines.

One possibility is to logically partition the address space. Thus, every coherence domain only accesses (and caches) addresses in its own partition, which sidesteps the issues caused by missing hardware-based coherence. However, this requires different programming models, such as the Partitioned Global Address Space (PGAS) model or the message-passing model. In both models, efficient data transfers between coherence domains are essential for program performance.

In this dissertation, we make the following technical contributions:

- We study the compilation of X10, a PGAS language, to invasive architectures, a family of non-cache-coherent architectures.
- We describe how we map X10's language features to invasive software and hardware.
- We study in detail data transfers on invasive architectures.
- We present a novel data-transfer technique that avoids serialization of pointered data structures.
- We propose hardware support for range-based cache operations and consider possible implementations.
- We extensively evaluate our data-transfer techniques on an FPGA prototype of an invasive architecture using an existing testsuite.
- We evaluate the hardware overhead of our range operations with an FPGA-based prototype implementation.

Code generation with permutation instructions. Modern parallel architectures exploit parallelism also on the instruction level. Such out-of-order processors dynamically rearrange instruction streams to the extent permitted by the data dependencies between instructions. Hence, instructions are not necessarily executed in the order specified in the program.

In order to implement this technique, such processors employ register renaming. Here, the processor has more physical registers than logical registers exposed in its instruction set. This enables the processor to eliminate certain dependencies between instructions, which would otherwise prevent their independent execution.

In a common implementation of register renaming, the processor contains a so-called register alias table. This table maps logical to physical registers. The table is purely controlled by hardware and inaccessible to software. We can express some value transfers between registers solely by modifying this indirection table, without touching any register contents.

There are many occasions during code generation where it would be beneficial for the compiler to have access to this mapping. However, current instruction sets only offer indirect access in the form of copy and swap instructions on registers.

In this dissertation, we make the following technical contributions:

- We propose the concept of permutation instructions that allow permuting the contents of a small set of registers. This can be viewed as allowing software to more directly manipulate an indirection table similar to a register alias table.
- We extend an existing instruction set with permutation instructions to arbitrarily permute up to five registers in one clock cycle.
- We describe an FPGA-based prototype implementation of this extended architecture with permutation instructions.
- We develop two code-generation schemes that allow compilers to exploit permutation instructions: a fast heuristic and an optimal dynamic programming-based approach.
- We formally prove the latter to be optimal.
- We implement both code-generation schemes in an existing compiler and conduct an extensive evaluation using an adapted processor emulator as well as our hardware prototype.

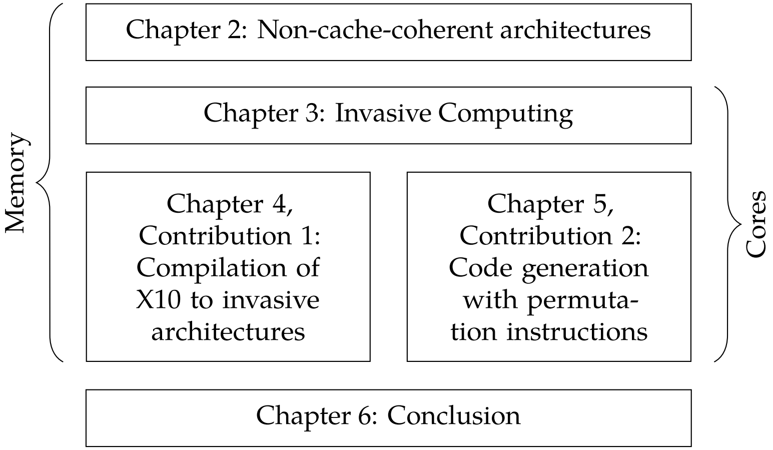

Chapter 2: Non-cache-coherent architectures
Chapter 3: Invasive Computing
Chapter 4, Contribution 1: Compilation of X10 to invasive architectures
Chapter 5, Contribution 2: Code generation with permutation instructions
Chapter 6: Conclusion
Memory
Cores

Figure 1.3: The structure of this dissertation.

1.2. Structure

In Chapter 2, we first give an overview of non-cache-coherent architectures and discuss their impact on programming models and compilers. This allows us to proceed to Chapter 3, which introduces the research project Invasive Computing, as part of which we carried out the work described in this dissertation. In particular, we present the hardware platform developed in the context of this project as an instance of a heterogeneous non-cache-coherent shared-memory architecture using our groundwork from Chapter 2. This hardware platform serves as the basis for both our contributions.

Then, in Chapter 4, we discuss compilation of X10 to invasive architectures. This includes our contribution regarding the efficient copying of pointered data structures.

Subsequently, we turn towards code generation aspects in Chapter 5. Here, we present our contribution concerning the use of permutation instructions to speed up program execution. To increase locality, we introduce the necessary hardware basics at the beginning of this chapter.

Chapter 6 summarizes our results and presents ideas for future research.

1.3. Notation and Conventions

As the results presented in this dissertation are intimately connected with several research projects that have many contributors, this dissertation uses “we” everywhere (except for the acknowledgment section). For the sake of completeness, we include some material that is not the contribution of the author. We explicitly state this fact at the beginning of such sections and switch to “Contributor et al.” and “they” if necessary.

We finish our definitions and theorems with a non-filled square □ and our proofs with a black square ■. We typeset `code like this`, with keywords such as **`if`** and **`else`** highlighted bold. We add hyphens to compound words if it avoids ambiguities. We use a comma after both “e.g.” and “i.e.”, as proposed by the majority of style guides we consulted. We differentiate between running time (the wall clock time of benchmark runs), run-time (the point in time when a program runs, in contrast to, e.g., compilation time), and runtime (as a shorthand for runtime library). We use the units and prefixes defined by the standard IEEE 1541-2002.

In printed versions of this dissertation, we provide a DVD with all software artifacts produced as part of this dissertation. We also make all artifacts available for download. See Appendix B for an overview. All specific software revisions we mention are relative to the projects listed there.

1.4. List of Publications

In this section, we give an overview of the author’s publications. We differentiate between publications that contribute to the dissertation at hand and those which do not.

The following publications contribute to material presented in this dissertation. All mentioned talks were given by the author.

Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, and Jörg Henkel. “Hardware Acceleration for Programs in SSA Form”. In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES’13. Piscataway, NJ, USA: IEEE Press, 2013, 14:1–14:10. DOI: 10.1109/CASES.2013.6662518

Presented on October 1, 2013 in Montréal, Canada.

Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. *Dynamic X10: Resource-Aware Programming for Higher Efficiency*. Tech. rep. 8. X10 '14. Karlsruhe Institute of Technology, 2014. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000041061`

Presented on June 12, 2014 in Edinburgh, Scotland.

Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. "Cutting out the Middleman: OS-Level Support for X10 Activities". In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10'15. Portland, OR, USA: ACM, 2015, pp. 13–18. ISBN: 978-1-4503-3586-7. DOI: `10.1145/2771774.2771775`

Presented on June 14, 2015 in Portland, USA.

Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter. "Optimal Shuffle Code with Permutation Instructions". In: *Algorithms and Data Structures*. Ed. by Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege. Vol. 9214. WADS'15. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 528–541. DOI: `10.1007/978-3-319-21840-3_44`
Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter. "Optimal Shuffle Code with Permutation Instructions". In: *CoRR* abs/1504.07073 (2015). URL: `http://arxiv.org/abs/1504.07073`

Presented on August 5, 2015 in Victoria, Canada.

Manuel Mohr and Carsten Tradowsky. "Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores". In: *Proceedings of Design, Automation and Test in Europe Conference Exhibition*. DATE'17. IEEE, Mar. 2017, pp. 1781–1786. DOI: `10.23919/DATE.2017.7927281`

Presented on March 30, 2017 in Lausanne, Switzerland.

Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. *An X10 Compiler for Invasive Architectures*. Tech. rep. 9. Karlsruhe Institute of Technology, 2012. URL: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112`

The following publications do not contribute to material presented in this dissertation.

Jonathan Aldrich, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. "Permission-Based Programming Languages (NIER track)". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: ACM, 2011, pp. 828–831. DOI: `10.1145/1985793.1985915`

Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. "AEminium: A Permission Based Concurrent-by-Default Programming Language Approach". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 36.1 (Mar. 2014), 2:1–2:42. DOI: `10.1145/2543920`

Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. "Malleable Invasive Applications". In: *Proceedings of the 8th Working Conference on Programming Languages*. ATPS'15. Springer Berlin Heidelberg, 2015, pp. 123–126

Alexander Pöppl, Marvin Damschen, Florian Schmaus, Andreas Fried, Manuel Mohr, Matthias Blankertz, Lars Bauer, Jörg Henkel, Wolfgang Schröder-Preikschat, and Michael Bader. "Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model using Reconfigurable Hardware in Invasive Computing". In: *Euro-Par 2017: Parallel Processing Workshops*. Lecture Notes in Computer Science. Heidelberg, Berlin: Springer-Verlag, Aug. 2017

There are only two hard things in Computer Science: naming things, cache invalidation, and off-by-1 errors.

Leon Bambrick, based on quote by Phil Karlton

2 Non-Cache-Coherent Architectures

In this chapter, we give an overview of non-cache-coherent shared-memory architectures. First, we cover fundamentals about parallel hardware architectures. Then, we give a more precise definition of cache coherence and present hardware-based and software-based implementation techniques. Subsequently, we discuss reasons for abandoning hardware-based coherence and give examples of resulting architectures. Lastly, we investigate the impact of missing hardware-based coherence on programming models and compilers.

2.1. A Taxonomy of Parallel Architectures

In this section, we give an overview of different types of parallel hardware architectures. We base our presentation on [HP11], but deviate in some details. We look at two orthogonal aspects:

1. How is memory organized?

2. How do cores communicate?

2.1.1. Memory Organization

We differentiate between architectures with centralized and distributed memory.

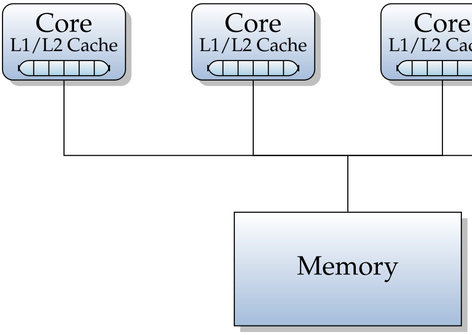
Centralized memory. Figure 2.1a shows the basic structure of machines with a centralized memory. Following this model, one or more cores share a single memory. Typically, the cores are connected to the memory via a bus. When adding more and more cores to such an architecture, the memory becomes a bottleneck as it cannot satisfy the bandwidth requirements of a large number of cores. While larger caches can mitigate this effect, past a certain core count it becomes necessary to have multiple memories.

Distributed memory. Figure 2.1b shows the basic structure of such systems with a physically distributed memory, i.e., multiple memories. Each core or group of cores has a local memory and all cores are connected by a scalable global interconnection network. The main advantage of distributed-memory machines is that multiple memories also multiply the possible memory bandwidth, i.e., it is easier to supply enough data to all cores than with a single memory. Distributed memory has the main disadvantage of higher implementation complexity, especially for the global interconnection network, which must be able to support the higher available memory bandwidth.

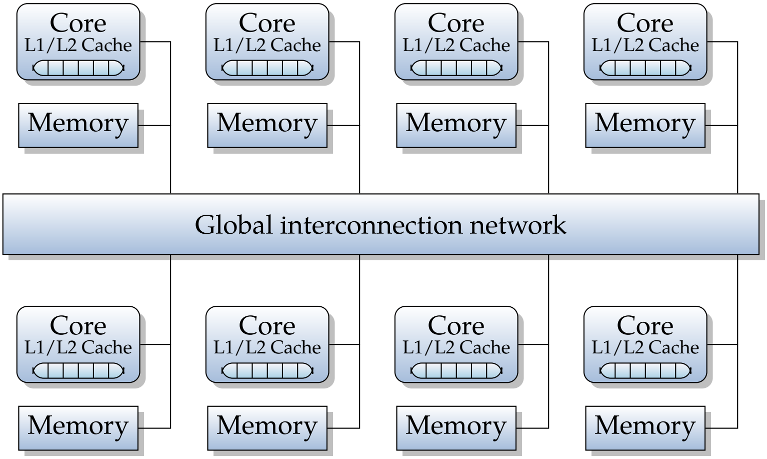
2.1.2. Communication Model

We differentiate between communication via shared memory and via message passing.

Shared memory. In a shared-memory system, the hardware offers a single shared address space. Each core may read from and write to this address space. Hence, cores can communicate via loads and stores to the shared address space.



(a) A centralized memory architecture. In this case, four cores are connected to a single main memory by a bus.



(b) A distributed-memory architecture. Eight cores, each connected to a local memory, are connected via a global interconnect.

Figure 2.1: A comparison of memory architectures.

Message passing. In a pure message-passing system, the hardware does not offer a shared address space. Hence, it is not assumed that every core can access all available memory. Thus, in general, cores cannot share data and therefore communicate by sending explicit messages. Passing a message is inherently linked to copying the necessary data to the receiver's address space as otherwise the receiver is unable to access the data.

2.1.3. Typical Combinations

In theory, memory organization and communication model are completely independent. In practice, the following three combinations are important.

Shared memory with centralized memory. Machines that provide shared memory with a single centralized memory are often called *symmetric multiprocessing* (*SMP*) systems. As all cores have the same distance from the single main memory, they also have the same access latency. Therefore, these machines are also called *uniform memory access* (*UMA*) architectures. This is the most popular type of memory organization for single-core and multi-core machines.

Shared memory with distributed memory. Machines that provide shared memory with distributed memory are usually referred to as *distributed shared-memory* (*DSM*) architectures. Here, the hardware still provides a single address space, hence every core can still access the complete memory. However, there is now a notion of locality as accessing a local memory is faster than accessing a remote memory. Therefore, these machines are also called *non-uniform memory access* (*NUMA*) architectures. The NUMA model is the standard for today's server machines. Typically, this is due to the memory controller being integrated into the processor. Hence, as soon as a machine possesses multiple physical processors, i.e., CPUs in multiple sockets, it automatically becomes a NUMA architecture.

Message passing with distributed memory. Such machines typically provide multiple private address spaces. Each core or group of cores has its own private address space, which is not addressable by remote cores.

Hence, the same physical address can refer to different memory locations for different cores. A typical representative of this class of machines is a cluster computer. Often, clusters are not pure message-passing systems. For efficiency reasons, shared memory is offered and used for small groups of cores, e.g., one node of a cluster, and message passing is used between core groups.

2.2. Cache Coherence

Usually, systems introduce caches to exploit spatial and temporal locality of data accesses. Typically, every memory address accessed by a core is first looked up in the core's cache. For example, when a core loads from memory address A, it is first checked if there is a valid copy of the data from A in the cache. This is called a *cache hit*, where the cache returns the value from the local data copy without consulting the memory. Only in case of a *cache miss* is the memory actually accessed.

In a shared-memory system with multiple cores, the caching of data can lead to *incoherent* situations unless special measures are taken. In general, incoherence refers to a situation where stale, i.e., outdated, data is accessed. As an example, suppose that we have two cores c_1 and c_2, each with a private cache. Further, suppose that the shared memory holds the value 100 at address A. First, both cores read from A and therefore have copies of that datum (100) in their local caches. Now, c_1 writes the value 200 to A. After the write by c_1, c_2 reads from A. If we do not take any precautions, the situation has now become incoherent, as c_2 would still read the old value 100 from its cache.

In practice, this incoherent situation is prevented by using a coherence protocol. In our example, this protocol must prevent c_2 from observing the old value while c_1 observes the new value. There exist numerous possible protocol variants and even more implementation possibilities, but all protocols have in common that they maintain *coherence invariants*. To understand what a coherence protocol must accomplish, first we have to define coherence in a precise manner.

We follow the definition by Sorin et al. [SHW11, section 2.3]. Sorin et al. use the *single-writer-multiple-reader* (*SWMR*) *invariant* as the foundation for their definition of coherence. The SWMR invariant states that, for any given memory location M at any given moment in time, there is

(i) either a single core that may read and write M, or
(ii) any number of cores that may only read M.

Especially, there must not exist a point in time, so that some memory location M may be written by a core and at the same time read or written by another core.

Sorin et al. propose another way of viewing this definition. They divide the lifetime of each memory location into *epochs*. Viewed this way, during each epoch there must be either a single core with read-write access or any number of cores with read-only access.

However, the SWMR invariant alone is not enough to capture our intuitive understanding of coherence. For example, in an epoch where two cores have read access to a memory location, the SWMR invariant does not state anything about the values that the cores read. Hence, it would be allowed that they read different values. Clearly, this is an incoherent situation like the one from our first example and therefore we must complement the SWMR invariant.

Sorin et al. add the *data-value invariant*. This invariant regulates the propagation of values from one epoch to the next. More precisely, it states that the value of a memory location M at the start of an epoch is the same as the value of M at the end of M's last read-write epoch.

Definition 1 We call a system *coherent* if the following two invariants hold [SHW11, p. 13]:

1. Single-Writer, Multiple-Reader (SWMR) Invariant: For any memory location M, at any given (logical) time, there is only a single core that may write to M (and can also read it), or any number of cores (possibly zero) that may only read M.

2. Data-Value Invariant: The value of a memory location M at the start of an epoch is the same as the value of M at the end of M's last read-write epoch. □

$$\begin{array}{l|l}
\text{Core } c_1\text{:} & \text{Core } c_2\text{:} \\
S_1\text{:}\quad \mathtt{x} \leftarrow 1 & S_2\text{:}\quad \mathtt{y} \leftarrow 1 \\
L_1\text{:}\quad r_1 \leftarrow \mathtt{y} & L_2\text{:}\quad r_2 \leftarrow \mathtt{x}
\end{array}$$

Figure 2.2: Program running on two cores. Initially, memory locations x and y hold the value 0.

2.2.1. Separating Coherence from Consistency

Following Sorin et al. [SHW11], we separate the issue of coherence from the issue of memory consistency. A memory-consistency model, or memory model for short, specifies the allowed behavior of a system where multiple cores execute loads and stores on a shared memory. For a given program, program input, and initial memory state, the memory model specifies what values the load operations executed by a core may return, and the memory model defines a final memory state.

Viewed another way, if we look at the set E of all possible executions for a given program, a memory model partitions E into a set of allowed executions (that adhere to the rules of the model) and a set of disallowed executions (that do not adhere to the model's rules). In contrast to execution on a single core, with multiple participating cores a memory model usually allows multiple correct program executions and disallows many incorrect executions.

Figure 2.2 shows an example program inspired by Dekker's algorithm [Dij02] for mutual exclusion. We use x and y to denote memory locations, use r_i for machine registers, and use L_j and S_k for load and store operations, respectively. Initially, memory locations x and y hold the value 0. In the program, core c_1 writes 1 to x and then reads from y into a local register. Similarly, core c_2 writes 1 to y and then reads from x into a local register.

Now, which outcomes of this program are allowed? Intuitively, $(r_1, r_2) = (1, 1)$, $(r_1, r_2) = (0, 1)$, and $(r_1, r_2) = (1, 0)$ are possible due to different interleavings of the instructions. These outcomes are *sequentially consistent* as the interleavings respect the partial orders defined by the program order of instructions in each sequential program part.

c_1	c_2	coherence state of `x`	coherence state of `y`
L_1		read-only for noone	read-only for c_1
	L_2	read-only for c_2	read-only for c_1
S_1		read-write for c_1	read-only for c_1
	S_2	read-write for c_1	read-write for c_2

Table 2.1: Coherence states for the execution L_1, L_2, S_1, S_2 of the program from Figure 2.2.

But what about an outcome where both cores load the value 0, i.e., $r_1 = 0$ and $r_2 = 0$ after execution of the program? A first intuition could be that this can only happen due to some coherence-related problem, where the system is incoherent and the cores read stale, i.e., not updated yet, values of `x` and `y`.

Indeed, a faulty coherence mechanism could lead to this situation. Suppose the writes from both cores are cached in the core's respective caches. Now, assuming a faulty coherence implementation, the following reads by both cores could load 0, i.e., stale values. This would violate the data-value invariant of Definition 1.

However, as we see in the following, there is an execution order that conforms to our definition of coherence and at the same time leads to the observed program behavior. Table 2.1 shows the coherence states of memory locations `x` and `y` during the execution L_1, L_2, S_1, S_2. We see that neither of our invariants from Definition 1 is violated, hence, this execution is coherent.

It may seem strange that load and store operations are executed in an order different from the program order. Perhaps suprisingly, even common memory models, including the Java memory model and the x86 memory model, allow this execution order[2]. The reason for this is that allowing such executions enables a multitude of performance optimizations, both on the software level, i.e., in the compiler or virtual machine, and on the hardware level.

[2] Modern memory models guarantee sequential consistency for data-race-free programs (the so-called DRF guarantee). However, our example program contains a data race.

For example, if a compiler can prove that `x` and `y` from Figure 2.2 do not alias, i.e., always refer to distinct memory locations, the compiler is allowed to generate code that first loads a value and then stores the other, even if in the program the write precedes the read[3]. Similarly, the CPU cache may include write buffers to hold data that must be written to memory. The buffer enables the cache to service following load operations without waiting for the memory to actually finish writing back the value of the store operation, effectively allowing a load to overtake a preceding store.

Hence, it is useful to separate the concepts of cache coherence and memory consistency. Sorin et al. give two more reasons why these are two separate issues. First, an important difference between coherence and consistency is that coherence is only concerned with a single memory location, while memory-consistency models consider accesses to multiple memory locations. This is also apparent in Definition 1, which only deals with accesses by multiple cores to a single memory location. And second, the question of whether the discussed execution of the program from Figure 2.2 is allowed also arises in a system without any caches. In such a system, there is clearly no need for cache coherence; however, it still needs a memory model. In practice, most implementations of memory models assume and exploit cache coherence.

2.2.2. Implementation

Definition 1 tells us what a coherence protocol must achieve but not how it can maintain these invariants. Again, we have orthogonal design decisions: we can choose between different coherence policies; we can choose a granularity; and we can put the responsibility for implementing coherence onto hardware or software.

The implementation of a coherence protocol often depends on the cache configuration. We can configure caches in *write-through* or *write-back* mode. Write-through caches update the main memory on every write. Hence, the main memory always contains an up-to-date value for a certain

[3] All modern programming languages that define a memory model offer means to restrict such reorderings, e.g., by using the keyword `volatile` [Gos+14, section 17.4] in Java.

address. Write-back caches do not update the main memory on every write. Therefore, with multiple write-back caches, it is more difficult to find the most up-to-date value of a data item, as it can be solely located in one of the caches. In such a case, cache terminology usually refers to this copy as *dirty*. In general, write-through caches are simpler to implement but have higher main-memory-bandwidth requirements.

2.2.2.1. Coherence Policy

In general, we can classify a coherence policy as either *write-invalidate* or *write-update* [Ste90; PP84]. Both types of policies must maintain the invariants from Definition 1.

Write-invalidate policies maintain coherence as follows. When a core c updates its local data copy of memory location L, write-invalidate policies enforce the invalidation of all other copies of L. Hence, the SWMR invariant is maintained by forcing the coherence state of L to "read-write for c". The next time another core c' reads L, either c provides the new value directly to c', or c first writes back its local value to memory location L, where c' then fetches it from.

With a write-update policy, when a core updates its local data copy of memory location L, at the same time, it updates all other copies of L. Hence, the SWMR invariant is maintained by forcing the coherence state of L directly to "read-only for all cores that had a copy of L". The policy implementation decides whether the copy in memory is updated as well.

Write-invalidate policies distribute updated data items lazily, while write-update policies do so eagerly. In general, write-invalidate policies are far more common than write-update policies.

2.2.2.2. Granularity

Common processors can perform loads and stores at various granularities, usually ranging at least from 1 to 8 bytes, and sometimes including wide memory operations, e.g., 64 bytes for vector instructions. In theory, it would be possible to manage coherence at the finest granularity, i.e., 1 byte.

However, this would considerably increase overhead for implementation and coherence traffic.

Therefore, in practice, implementations manage coherence at a coarser granularity, most commonly cache lines. Enforcing the coherence invariants per cache line is, in general, more efficient, as they comprise multiple bytes (16–64 bytes are common). However, managing coherence at a coarse granularity can also cause other performance problems, such as false sharing [BS93].

False sharing occurs when multiple cores access and modify different, non-overlapping data objects within the same cache line. For example, suppose that core c_1 repeatedly modifies memory location L_1, core c_2 repeatedly modifies memory location L_2 (different from L_1), and L_1 and L_2 happen to be part of the same cache line. Here, the coherence protocol still maintains the SWMR invariant for the whole cache line. Hence, every time c_1 modifies L_1, a coherence action is triggered. For example, assuming a write-invalidate protocol, c_2's cache line containing L_2 is invalidated, although L_2 did not change at all. The same happens for c_1's cache line containing a copy of L_1 on the next update to L_2 by c_2. Thus, an unfortunate combination of memory layout, access behavior, and coherence granularity can lower performance significantly.

2.2.2.3. Responsibility

We can implement the system that maintains the coherence invariants either in hardware or in software [Adv+91; TM97].

Hardware-based coherence. If we implement coherence in hardware, it is functionally invisible to software. Hence, for a shared-memory system with hardware-based cache coherence, the caches behave like in a single-core system. Correctly implemented, hardware-based cache coherence makes it impossible for the programmer to determine whether a system has caches by inspecting the results of load and store operations [SHW11]. However, it may be possible to deduce the presence of caches using timing information.

In the following, we give a brief overview of the two most important implementation techniques for hardware-based cache coherence: snooping protocols and directory schemes. Snooping protocols rely on a medium that is able to broadcast information, e.g., a bus, and distribute the information about the sharing status of each memory block. On the other hand, directory schemes centralize the information about the sharing status of memory blocks in one location, called the directory. As a consequence, they do not require the ability to broadcast information. In general, snooping protocols are easy and cheap to implement, while directory schemes are more complex, but scale to higher core counts. We base our presentation on Hennessy et al.'s [HP11] and also refer to the same source for details.

The idea behind snooping protocols is that addresses are broadcast on the shared medium (e.g., a bus) and all participants observe, or "snoop", these addresses to potentially trigger actions in their respective local caches to maintain coherence [HP11, section 4.2]. To illustrate this idea, we discuss the implementation of a snooping coherence protocol for a simple memory architecture using a bus as shown in Figure 2.1a. We assume a write-invalidate policy, as it is the most commonly used strategy.

As an example, suppose we have two cores c_1 and c_2 in such a system, each with a private cache configured in write-back mode. If c_1 reads from address L, it puts a copy of the data at address L in its local cache. Suppose c_2 now wants to write a new value to L, which proceeds as follows. After c_2 has successfully acquired bus access[4], it broadcasts an invalidate operation on the bus. Core c_1 reacts by invalidating its local copy of L, i.e., the next access to L by c_1 will cause a cache miss. Then, c_2 performs the actual write operation. With write-back caches, now only c_2's cache holds the new value of L; the copy of L in c_1's cache is marked invalid and the copy in memory is outdated.

If c_1 now reads from L again, we must (i) somehow notice that reading from main memory is incorrect (as the new value is in c_2's cache), and (ii) transport the new value to c_1's cache. Fortunately, we can implement the notification mechanism exactly as with the original write described before. Thus, we require caches to also observe read operations on the

[4] If multiple cores want to write to the same address L concurrently, the bus-acquisition process serializes their write operations.

bus and to check if they have a modified copy of the data at the requested address. If this is the case, they abort the other core's memory access and then provide the new value. In our example, c_2 would see c_1's read to L on the bus and then abort it.

How exactly then c_2 makes the new value available to c_1 is another design decision. One option is that c_2 writes back the new value to main memory and then sends a retry signal to c_1, which restarts the read operation. The alternative is that c_2 sends the new value directly to c_1, without a detour via main memory. The first option is easier to implement but potentially slower as updated values are distributed via main memory. The second option requires additional bookkeeping and increases implementation complexity, but distributes updated values over a potentially faster interconnect between cores, without involving main memory. This design decision differentiates the two well-known coherence protocols MESI [PP84] and MOESI [Adv10, section 7.3].

As we have seen, bus snooping protocols need to broadcast, i.e., communicate with all other caches, on every cache miss. On a read miss, we have to inform all other caches of our intent to read the address and they might respond by aborting our read request, followed by providing the updated data item. On a write miss, we also have to inform all other caches as they might need to invalidate their copy. In total, the amount of coherence-related traffic can soon overwhelm the capabilities of the bus as we increase the number of cores (and caches).

An alternative offering better scalability are directory protocols [HP11, section 4.4]. They build upon the idea of the *directory*, which is a data structure that holds the sharing status of each cacheable memory block. The most important improvement compared to snooping-based protocols is that we save the sharing status of a block in a single, well-defined location (the directory) instead of replicating information in multiple locations. This avoids the need to broadcast information to synchronize multiple copies of the sharing status.

However, we can still distribute the directory itself. Directory schemes are often used for distributed shared-memory machines as depicted in Figure 2.1b. In such a setting, each core with its local cache and local memory is extended with a directory responsible for the memory blocks in the respective local memory. Hence, while the sharing information is distributed, it is not replicated, as we save the current sharing status of each memory block in exactly one location.

In their simplest form, directory schemes maintain one directory entry per memory block. Each entry holds the block's current sharing status. A basic protocol differentiates between the following sharing states (with more fine-grained states allowing potentially higher performance at the cost of increased complexity):

Uncached: No core has a copy of the memory block.

Shared: The block is cached by at least one core, and the values of this block in memory and in all caches match. This means no core has modified the block. Additionally, we have to save information about which cores have copies of the block in their caches (the *sharer set*).

Modified: Exactly one core (the owner) has a copy of the block, and the block is modified. Hence, the copy in memory is outdated. We also save which core is the owner.

In a directory scheme, up to three types of cores may be involved in a memory access:

- the local requesting core that reads or writes the cache block;
- the home core whose memory holds the requested cache block; and
- the remote core whose cache holds a copy of the requested cache block.

As an example, suppose we have three cores c_1, c_2, and c_3 in a DSM system as shown in Figure 2.1b, each with private write-back caches. We assume that initially, all caches are empty, i.e., all entries in all directories are set to Uncached. Furthermore, we assume memory location L is physically located in c_1's memory.

Now, suppose that in our example c_1 reads from L. Here, c_1 is both the local and the home core; no remote core is involved as all caches are empty. Hence, c_1 puts L in its local cache and updates in its local directory the state of L to Shared as well as the respective sharer set to $\{c_1\}$.

Now, assume that c_2 reads L next. The local core c_2 then sends a read request to home core c_1, which adds core 2 to the set of sharers registered for L in c_1's directory, and then returns the data at L to c_2.

Now, suppose that the next action is a write to L by c_3. Hence, the local core c_3 sends a write request to home core c_1. Core c_1 responds by (i) sending the requested block back to c_3, (ii) reading the set $\{c_1, c_2\}$ of sharers and sending them invalidation requests, and (iii) setting the state of L to Modified in c_1's directory while registering c_3 as the owner. Cores c_1 and c_2 then invalidate their local copies of L.

In summary, directory schemes scale better than snooping-based protocols as they do not depend on broadcasts. However, they are also difficult to scale to large numbers of cores. For example, the sharer set is often implemented as a bit set with one bit per core, where a 1 at position i means that core c_i currently has a copy of the respective memory location in its cache. For 1024 cores, storing the bit set requires 128 bytes, which may be more than the size of the memory block whose sharing state the bit set is supposed to track. Another issue is the significantly increased power usage due to the high number of messages for coherence traffic [KK10].

Software-based coherence. Alternatively, coherence can be implemented in software. This means that the software must trigger necessary cache operations, such as invalidations and write-backs. To be able to do that, the hardware must provide appropriate support. In the context of the following discussion, we assume the existence of an *invalidation* instruction and a *writeback* instruction with the following semantics:

- The invalidation instruction `invalidate` L takes a memory location L as an operand and invalidates the copy of L in the executing core's cache (if a copy is present). This enforces that L is fetched from main memory on the next access. Note that invalidating a locally modified copy discards these local changes.
- The writeback instruction `writeback` L takes a memory location L as an operand and writes the copy of L in the executing core's cache back to memory (if a copy exists and it has been modified locally).

We can implement these instructions as part of the cache logic. For example, the invalidation instruction looks up L in the cache and in case of a cache hit, marks the respective cache line as invalid, e.g., by clearing the cache line's valid bit. We can implement the writeback instruction similarly.

To demonstrate the usage of these instructions, suppose that we have two cores c_1 and c_2, each with private caches, and a shared variable x in memory. Suppose further that the cores execute the following program:

Core c_1:

```
x ← 1
```

Core c_2:

```
A: r ← x
   if (r == 0) goto A;
```

On a system without hardware-based cache coherence, this program potentially runs forever, as there is no guarantee that c_2 will ever see the updated value of x. Only when the cache line containing the updated value of x is evicted from c_1's cache, the memory is updated and the updated value therefore becomes visible to c_2. If the cache line is never evicted from c_1's cache, the loop on c_2 does not terminate.

In this example, a possible software-based solution to maintain coherence looks as follows:

Core c_1:

```
x ← 1
writeback x
```

Core c_2:

```
A: invalidate x
   r ← x
   if (r == 0) goto A;
```

Here, we placed a writeback instruction after the write to x and an invalidation instruction before the read from x. The writeback instruction placed after the write operation by c_1 ensures the propagation of x's new value to memory. Hence, the updated value becomes visible to c_2. Moreover, in the loop, c_2 first invalidates its local copy of x, which forces a cache miss for the following read operation and thus forces a retrieval of the updated value from memory.

While this example demonstrates that managing coherence in software is possible in principle, it is unclear which component triggers the coherence actions. Manual coherence management is unrealistic, as the process is too error-prone. Hence, some part of the system software should trigger the cache operations, e.g., the operating system, a library, or the compiler as in our example.

In general, we can classify software-based coherence schemes as either static or dynamic [TM97]. Static schemes are compiler-based and rely on program analysis at compile time. Dynamic schemes are implemented in operating systems or libraries and monitor memory-access behavior at run-time. We discuss this topic in more detail in Section 2.4.2, but give an intuition of the trade-offs here.

For static schemes, the compiler must identify potentially conflicting accesses to shared data and extend the program to trigger cache operations at appropriate program points, e.g., by generating additional instructions as seen in our example. To guarantee correctness, compilers must be conservative and assume the worst case when adding coherence-related actions to the program. Clearly, the compiler can insert cache operations before and after each memory-access operation. However, this effectively disables the system's caches.

Hence, to make static schemes viable performance-wise, the compiler has to reduce the number of inserted cache operations. The fundamental problem is that compile-time information must be used to predict run-time access behavior. This works well for programs with a regular predictable access behavior. However, for programs with an irregular memory-access behavior, many unnecessary coherence actions may be performed at run-time, lowering performance significantly.

For dynamic schemes, a library or the operating system maintains cache coherence at run-time. Operating systems usually exploit virtual memory to manage coherence at page granularity (a typical page size is 4 KiB). They enforce the SWMR invariant for pages: if a core writes to a shared page, this page must be invalidated on all other cores. The next time another core accesses the shared page, the page-fault handler manages coherence in software, e.g., by writing back the changed page contents on the core that previously wrote to it. Similarly, libraries can manage coherence in software at the granularity of objects or memory regions.

Dynamic schemes detect problematic accesses at run-time, so they may reduce unnecessary coherence actions. However, in order to limit coherence-related overhead, they have to work on a coarser granularity, such as whole pages. Depending on the memory layout and access behavior of the program, false sharing (cf. Section 2.2.2.2) can decrease performance significantly.

Overall, software-managed coherence was a field of active research in the 1980s and 1990s. Tartalja et al. [TM97] and Stenström [Ste90] provide overviews and classifications of early software-based coherence schemes. Ultimately, however, hardware-based cache coherence became the standard for shared-memory multi-core architectures. Snooping-based protocols were reasonably simple to implement and offered good performance. Most importantly, hardware-based cache coherence enables all parallel software and their respective tools, such as compilers, to be oblivious to the existence of caches.

However, more recently, software-based coherence has regained interest in the context of non-cache-coherent many-core architectures. Such architectures do not implement hardware-based coherence; their properties and the implications are the topic of the next sections.

2.3. Hardware Architecture

Shared-memory multi-core architectures as described in Section 2.1 with hardware-based cache coherence as described in Sections 2.2 and 2.2.2.3 are by far the most common type of machine currently in use. Recently, however, non-cache-coherent shared-memory architectures have become attractive for two reasons: power and scalability.

Power. In power-constrained contexts, such as mobile computing, (partially) giving up cache coherence can enable more aggressive power savings. As an example, many multi-core systems in modern mobile phones consist of a group of “strong” cores and a group of “weak” cores [Tex14; LWZ14]. The idea is to use the weak cores when the system is not actively used and to switch to the strong cores for more demanding tasks.

However, hardware-based coherence between weak and strong cores restricts the architectural asymmetry, i.e., it prevents the weak cores from being much weaker (and thus from consuming much less power) than the strong cores. Additionally, the coherence mechanism itself consumes significant power [LWZ14; Cho+11; KK10]. Thus, some multi-core systems for mobile computing do not keep caches of weak and strong cores coherent in hardware while still allowing access to shared memory.

Hence, this constitutes a non-cache-coherent shared-memory architecture with two *coherence islands* or *coherence domains*, where all caches inside a coherence domain are kept coherent by a hardware-based mechanism, but the hardware provides no coherence between different domains.

Scalability. The other drivers toward non-cache-coherent architectures are scalability and performance. Following the trend of putting more cores on a chip, scaling chip designs with hardware-based cache coherence to high core counts has proved to be challenging. A centralized memory quickly becomes a bottleneck as more and more cores compete for bandwidth. Additionally, simple bus snooping coherence protocols do not scale well as the number of cores is increased due to the need for broadcasts (see Section 2.2.2.3).

Switching to a distributed shared memory mitigates memory-bandwidth issues by distributing the bandwidth demand to multiple memory modules. However, the lack of a common bus now requires more complex directory-based coherence protocols. While these directory schemes scale better, it is unclear whether they can be used at the core counts envisioned for future chips. We refer to [FNW15] for a recent overview of the challenges with scaling directory schemes.

This scalability problem has been called the “coherence wall” [Kum+11]. Whether or not this wall actually exists, and if it does, what the correct answer to this challenge is, is the subject of an ongoing debate.

In a widely cited work, Martin et al. [MHS12] argue that by combining known techniques to improve existing coherence protocols, on-chip cache coherence scales better than commonly anticipated by the community. They predict that future multicore architectures will keep full hardware-based coherence as the scalability benefits of giving up hardware coherence do not justify sacrificing backwards compatibility with existing operating systems, compilers, or software. However, they also point out that their prediction is based on a model and not on an actual implementation or simulation with realistic benchmarks.

Lotfi-Kamran et al. [Lot+12] propose that, in order to scale performance of a chip, architectures should focus on putting many independent servers, called "pods", onto a single chip. Each pod is a complete server, consisting of cores, caches, and interconnect, that runs its own copy of the operating system. As there is no interdependence between pods, there is no need for inter-pod communication or coherence support, which improves scalability.

Komuravelli et al. [KAC14] observe that often, scalability improvements are made through an even more complex coherence protocol, making implementation and verification significantly more difficult. Improving scalability of cache coherence is a field of active research, e.g., by hybrid software/hardware-based coherence [Kel+10]; by requiring a disciplined parallel programming model [Cho+11]; or by restricting coherence to smaller domains, such as applications [FNW15]. A radical answer to the scalability challenge is (at least partially) disposing of hardware-based cache coherence, resulting in a non-cache-coherent architecture.

Hence, we have seen that there are two reasons for abandoning global cache coherence: power and scalability. The main difference between these two classes is the number of coherence domains. In power-motivated systems, we see relatively few, i.e., two or three, coherence domains; for example, the mentioned weak and strong core groups. In scalability-motivated systems, there may be significantly more coherence domains, i.e., in the order of tens or hundreds.

The other interesting parameter is the size of coherence domains. In the extreme case, each core is in its own coherence domain, i.e., there are as many coherence domains as there are cores. Alternatively, domains include multiple cores each. We follow Fatourou et al. [Fat+16] and refer to the former class of architectures as *fully* and to the latter class as *partially* non-cache-coherent (or, equivalently, partially cache-coherent) architectures. When we refer to non-cache-coherent machines without further qualification, we mean both classes.

In the context of this dissertation, we investigate a scalability-motivated non-cache-coherent architecture with many coherence domains. We focus on partially non-cache-coherent systems. In the following section, we describe examples of non-cache-coherent architectures and then look at possible programming models in Section 2.4.

2.3.1. Examples of Non-Cache-Coherent Architectures

A radical solution to the challenge posed by the coherence wall are non-cache-coherent shared-memory hardware architectures. Various such architectures have been proposed recently, some partially and other fully non-cache-coherent.

The IBM Cell processor [Che+07; Hof05; Pha+05; Kah+05] is a heterogeneous non-cache-coherent multicore architecture with a focus on multimedia processing known for powering the Sony PlayStation 3 gaming console. Figure 2.3 shows its architecture.

The Cell processor consists of a conventional PowerPC-based core (PPE) and eight Synergistic Processing Engine (SPE) cores with a custom instruction set. All cores share access to an off-chip DRAM memory. While the PowerPC core has a conventional two-level cache hierarchy, each SPE only has a private local memory (called "LS" for "local store memory" in Figure 2.3). SPEs have to transfer data from main memory to their local SPE memories before they can access it. Dedicated DMA units accelerate these transfers. The Cell architecture provides no hardware cache coherence between local SPE memories. Hence, for multithreaded applications, coherence must be handled completely in software [MS10].

Intel's Single Chip Cloud Computer (SCC) [How+10; Mat+10] is a homogeneous non-cache-coherent architecture with 48 cores. The SCC does not provide any hardware cache coherence, not even for groups of cores. Figure 2.4 shows an overview of the SCC's hardware architecture.

The basic building block of the Intel SCC is a *tile* (cf. right half of Figure 2.4). Each tile consists of two x86 cores, each with private L1 and L2 caches. Caches are not kept coherent, neither inside a tile nor across tile boundaries. Additionally, each tile contains a message-passing buffer, which is a dedicated on-chip memory (16 KiB) for message passing between cores. To connect the tile to the rest of the system, each tile contains a mesh interface unit. All tiles are arranged in a 4 × 6 mesh and connected by

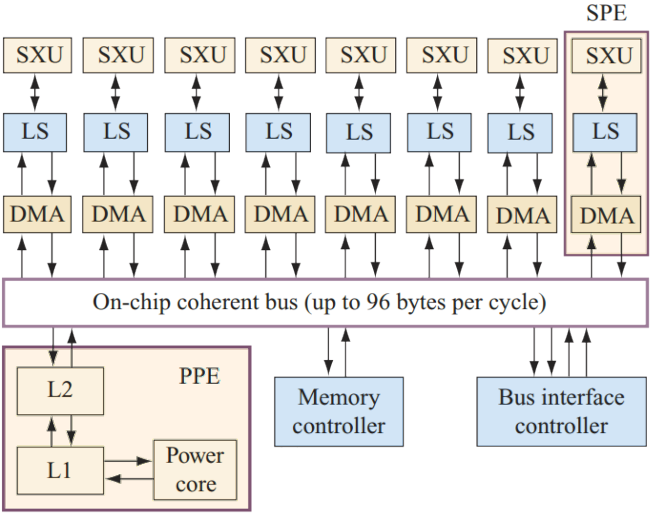


Figure 2.3: Overview of the hardware architecture of the IBM Cell processor. Image taken from [Kah+05] and slightly adapted. The bottom left shows the PowerPC-based core with caches (PPE). The upper half shows the eight SPEs, each consisting of the execution unit (SXU), local store memory (LS), and a DMA engine. SPE-local memories are not kept coherent in hardware.

a network-on-chip (cf. left half of Figure 2.4). A *network-on-chip* [BM02; Hei+14] (*NoC*) applies principles from networking, e.g., the concept of routers, to on-chip communication to improve scability and power efficiency. Four memory controllers (MC) at the edges of the chip provide access to off-chip memory.

The Intel SCC offers a shared physical address space that includes all message-passing buffers and the shared DRAM. Cores can then communicate by two means: (i) either by using dedicated message-passing hardware, or (ii) by using the off-chip memory. To use direct messaging, cores write to the destination tile's message-passing buffer (or read from the sender's memory). As coherence must be provided in soft-

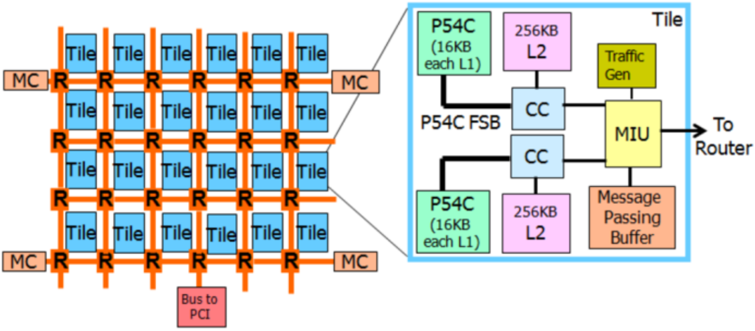


Figure 2.4: Overview of the hardware architecture of the Intel SCC [Int12]. On the left, *R* denotes NoC routers and *MC* are memory controllers. On the right, *MIU* stands for "mesh interface unit", which connects the tile to the NoC. We see that the caches (denoted by "L2") inside a tile are not connected to the same bus, as cache coherence does not need to be guaranteed.

ware, the SCC offers dedicated invalidation instructions for use with message-passing buffers (we give more details in Section 4.5).

Alternatively, cores can communicate via the shared DRAM. Usually, each core of the SCC is assigned a private partition of the off-chip memory, i.e., the hardware enforces that only the owner accesses this partition. However, it is also possible to create memory regions that are shared between cores. By default, these shared regions are marked non-cacheable, thereby avoiding possible coherence-related problems. Consequently, accessing these regions is slow as accesses do not happen at cache-line granularity but at the granularity of individual loads and stores. However, shared regions can also be marked as cacheable. In this case, the software has to manage coherence. While the SCC does not provide specialized hardware support for this use case, it is still possible to force invalidations and writebacks [Rot+12].

Intel's Runnemede [Car+13] is a proposed design of a heterogeneous non-cache-coherent many-core architecture. Its basic module is the *block* (shown in Figure 2.5).

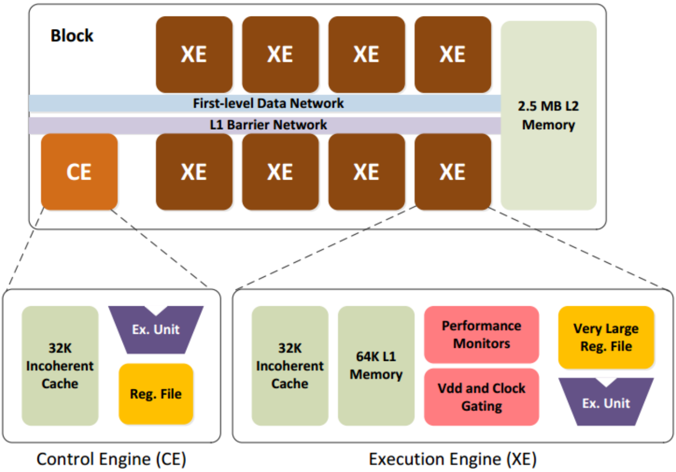


Figure 2.5: The contents of a block in the Runnemede architecture [Car+13]. The control engine (CE) executes the operating system and distributes tasks to the specialized execution engines (XEs). There is no hardware-based coherence provided between caches of CE and XEs.

Each block contains one general-purpose core, called the "control engine" (CE). The CE executes the operating system. Additionally, a block contains multiple execution engines (XEs), which are typically custom architectures. Both CEs and XEs have caches but there is no hardware-based coherence. Thus, the number of XEs per block is not limited by the scalability of a coherence protocol. Instead, there are as many XEs as the CE can supply with work without becoming the bottleneck.

Figure 2.6 shows that Runnemede combines multiple blocks to form a *unit* and multiple units to build a complete chip. Each level (block, unit, and chip) has its own network to transfer data, making Runnemede a hierarchical design. Across the whole chip, Runnemede provides a single 64-bit physical address space. To enable software-managed coherence between memories, Runnemede provides dedicated cache management instructions to invalidate and write back cache lines.

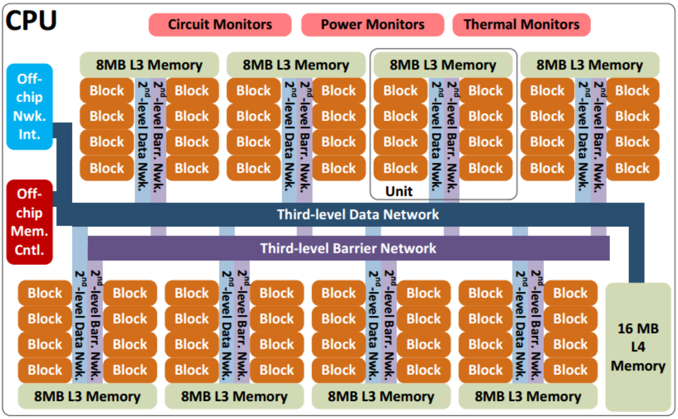


Figure 2.6: The overall chip architecture of the Intel Runnemede [Car+13]. Blocks are combined to form units; multiple units form a chip; and an off-chip network can connect multiple chips to form even larger systems.

The EUROSERVER project [Dur+14] is a homogeneous non-cache-coherent architecture aimed at servers in data centers. Figure 2.7 shows an overview of the architecture. EUROSERVER proposes *chiplets* composed of 8 ARM cores. Inside a chiplet, a classical hardware protocol provides full cache coherence. Each chiplet has a local DRAM connected to its internal bus.

The hardware offers a global physical address space. Chiplets are connected via a global interconnect and can access remote memory that resides on a different chiplet. There is no hardware-based coherence between chiplets. Instead of providing dedicated means for software-managed coherence, the EUROSERVER project proposes restrictive cache and access policies.

Suppose a core from chiplet A accesses a remote memory area M from chiplet B's DRAM, then one of the following policies shall be used:

1. Only A accesses M and caches it locally. If B was allowed to access M as well, incoherent situations could arise. For example, if A accesses and caches a part of M locally and then B modifies M, A is not notified and would operate on stale data. Using this policy means

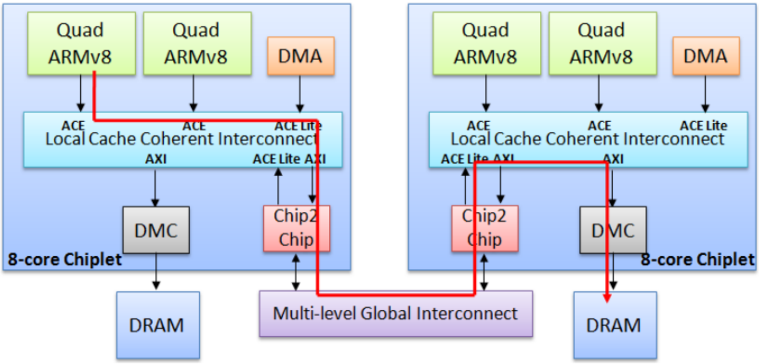


Figure 2.7: The EUROSERVER hardware architecture [Dur+14]. Eight ARM cores form a coherence domain, called chiplet. All chiplets share a physical address space and can access remote DRAM via the global interconnect (depicted by red arrow).

that A can “borrow” memory from B for exclusive use. Hence, it does not enable communication. Access to this memory happens on cache-line granularity; the performance depends on the global interconnect.

2. Both A and B access M, but only B caches it. As A does not cache M, every load and store request is sent to B via the global interconnect. The component that receives the requests on B's side is connected to B's local bus and is therefore covered by the hardware coherence protocol. From B's point of view, remote load or store requests are handled just as local loads or stores. While this policy allows sharing memory regions, its downside is that accesses from A are not cached locally, i.e., exploitation of temporal or spatial locality is impossible. Moreover, individual load and store requests are sent via the global interconnect, causing high protocol overhead.

Hence, EUROSERVER does not employ software-managed coherence. Instead, they propose using more restrictive caching policies to avoid incoherent situations at the cost of decreased performance when multiple chiplets access the same memory area.

The Formic Cube [Lyb+12a; Lyb+16] is a non-cache-coherent many-core architecture with 520 cores in total. The system has 8 fast ARM-based cores and 512 slower Xilinx MicroBlaze cores. The 512 slower cores are arranged in a 3D mesh. Each CPU has a full private cache hierarchy (L1 and L2 cache) but the caches are not kept fully coherent. Specifically, there is no hardware-based coherence between the L2 caches of the slower MicroBlaze cores. The application itself runs on the MicroBlaze cores, while the ARM cores execute a runtime system. The runtime system manages coherence between caches in software.

The OpenPiton project [Bal+16] presents an open-source many-core processor that allows building architectures with up to 500 million cores. To scale to such core counts, Fu et al. [FNW15] advocate the use of coherence domain restriction (CDR). CDR is based on the observation that the majority of cache lines are only shared by a small subset of cores, e.g., those belonging to a particular application. Hence, CDR restricts coherence to the level of applications or pages. While this requires additional hardware support, existing directory coherence protocols can be adapted to work with CDR and Fu et al. demonstrate good scalability.

Current graphics processor unit (GPU) architectures can also be considered non-cache-coherent architectures. They allow accessing shared memory but require disabling core-private caches if memory operations should be visible across more than one core [Sin+13]. Disabling caches avoids incoherent situations but lowers performance significantly. Moreover, there are heterogeneous systems composed of a combination of CPUs and GPUs that provide shared memory between CPU and GPU. By default, these systems do not implement hardware-based coherence between the caches of CPU and GPU.

For both pure GPU and mixed CPU-GPU systems, there has been work on providing hardware-based cache coherence [Sin+13; Pow+13]. However, existing coherence protocols do not scale to the core count and memory bandwidth of GPUs. Therefore, Power et al. [Pow+13] employ region coherence to manage coherence at a coarser granularity than individual cache lines. Basu et al. [Bas+16] argue that these proposed changes are hard to adopt due to their complexity. Instead, they suggest a hybrid software-hardware mechanism that exploits the semantic knowledge by system software (operating system or runtime system) to use hardware coherence only when needed.

The Intel Xeon Phi processor [Chr14] is a fully cache-coherent many-core architecture providing up to 72 cores. Christgau et al. [CS16] point out that the second-generation Xeon Phi processor (codename "Knight's Landing") may not be used in a multi-socket system that combines multiple Xeon Phi processors. They report that Intel restricted this use case as the coherence traffic between the processors would exceed the capabilities of the interconnect. The Xeon Phi can also be used as a coprocessor, e.g., in the form factor as a PCIe extension card. In such a configuration, the Xeon Phi runs alongside the regular system processor. Each processor can remotely access the other processor's main memory, but there is no hardware-based coherence [Bar+15]. Thus, the overall system has two coherence domains and coherence must be managed in software.

Hence, we have seen that a wide range of diverse many-core architectures reach the scalability limits of hardware-based cache coherence.

2.4. Programming Model

We now have a detailed understanding of the hardware structure of non-cache-coherent architectures: they still offer a shared physical address space; however, not all caches in the system are kept coherent by the hardware. Instead, the systems aim to improve scalability by only offering coherence islands of varying size, ranging from just one core to a few, e.g., four or eight. This raises the question of how to program these machines.

In the following, we will look at different programming models and investigate the work required by compiler and runtime system to bridge the gap between the guarantees expected by the programmer using a particular programming model and the guarantees provided by non-cache-coherent shared-memory machines.

First, we will briefly discuss what a programming model is and how it relates to the underlying hardware. Then, we will discuss the use of the following programming models on non-cache-coherent shared-memory architectures: (i) the shared-memory programming model, (ii) the message-passing programming model, and (iii) the partitioned global address space (PGAS) programming model.

2.4.1. Parallel Programming Models

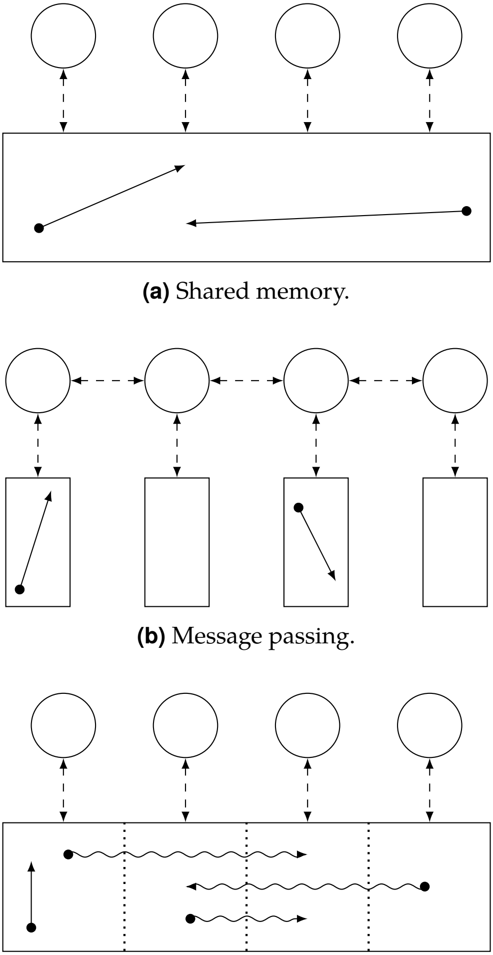
A parallel programming model is an abstraction of a parallel computer system architecture [MSM04; Bar16]. This model governs which tools programmers can use to express their algorithms. The two most important parallel programming models are the shared-memory model and the message-passing model.

In the shared-memory model, processes or threads share a common address space, which they read from and write to. Threads communicate by exchanging data via this common address space. Access to shared data is controlled using synchronization mechanisms, such as locks.

In the message-passing model, processes do not share a common address space. Instead, they communicate by sending and receiving messages. Usually, transferring data requires cooperation between sender and receiver, i.e., a send operation must have a matching receive operation.

Figure 2.8 shows an abstract view of the shared-memory programming model (Figure 2.8a) and the message-passing model (Figure 2.8b). Circles denote execution contexts (threads or processes), rectangles denote address spaces, dashed arrows denote memory accesses or communication operations, and solid arrows represent pointers. We see that in the shared-memory model, multiple threads operate on a shared uniform address space. In the message-passing model, we have completely separated address spaces and processes communicate via messages.

This looks very similar to the types of hardware architectures presented in Section 2.1. However, as programming models are an abstraction above hardware architecture, they are not tied to particular hardware capabilities. Theoretically, every programming model can be implemented on any underlying hardware. As examples, we sketch how we can implement the shared-memory model and the message-passing model on hardware architectures that suggest a different model.



(a) Shared memory.

(b) Message passing.

(c) Partitioned Global Address Space (PGAS).

Figure 2.8: Schematic comparison of the shared-memory, the message-passing, and the PGAS programming models. Circles denote execution contexts (threads or processes); rectangles denote address spaces; dashed arrows denote memory accesses or communication operations; and solid arrows represent pointers. Depiction based on [Sar+10].

We can easily realize a message-passing programming model on top of a shared-memory system. We can implement the primitives `send()` and `receive()` using write and read operations to the shared address space combined with appropriate synchronization. In fact, this is what many MPI implementations do internally when they are used on a shared-memory machine, such as a regular desktop computer.

Vice versa, we can also realize a shared-memory programming model on top of message-passing-based hardware. This technique is also known as software distributed shared memory (software DSM) [NL91]. The fundamental idea is to provide the illusion of a shared address space by hiding the required message passing. When accessing a piece of data that is physically located in a remote memory, some layer beneath the programming model triggers the required message(s) to fetch the data item and mediates access to it. This management can, for example, be performed by libraries or by the compiler. In both cases, references to data items actually consist of two parts: a description of the location of the data item (i.e., the number of the owning core) and the actual address that is only valid at the remote site.

In practice, not every programming model is a good fit for a particular hardware architecture and the overhead of choosing an unsuitable model may be high. Therefore, it is important to determine the cost of using a particular programming model on non-cache-coherent architectures as described in Section 2.3.

2.4.2. Shared-Memory Programming Model

As non-cache-coherent architectures provide a shared physical address space, it seems intuitive to continue using a shared-memory programming model. However, using this programming model is not directly possible. As described in Section 2.2, most memory model implementations exploit cache coherence by assuming a coherent memory system. This means that when mapping the memory model of the programming language to the memory model provided by the hardware, compilers assume a coherent memory system. Hence, if we run code generated assuming these guarantees on a hardware platform that does not implement cache coherence itself, the program will most likely not work as expected.

Thus, if we want to keep the familiar shared-memory programming model for the programmer, we must compensate for the missing hardware-based cache coherence on a level above the hardware, but below the programming language. Hence, either (i) the compiler, (ii) a runtime system or library, or (iii) the operating system must provide coherence.

In the following, we will give an overview of recent work in these areas. As mentioned in Section 2.2.2.3, software-managed coherence in general and compiler-managed coherence in particular was a field of active research in the 1980s and 1990s. We refer to [Ste90] and [TM97] for an overview of this early work. As hardware-based coherence became the standard, interest in software-based alternatives declined. In the context of this dissertation, we focus on the more recent work conducted due to the architectural trends described in Section 2.3.

Compiler-based approaches. McIlroy et al. [MS10] and Zakkak et al. [ZP16b; ZP16a] present Java virtual machines (JVMs) that can execute standard parallel Java programs on architectures without hardware-based cache coherence. McIlroy et al. describe Hera-JVM, which targets the Cell architecture; Zakkak et al.'s implementation, DiSquawk, targets the Formic Cube (both architectures are described in Section 2.3.1). In both cases, the Java virtual machine ensures coherence by explicitly triggering cache actions when necessary.

As mentioned in Section 2.2.2.3, triggering cache actions, i.e., invalidations and writebacks, too conservatively decreases performance considerably. The extreme case of writing back dirty data after every write and invalidating cached data before every read is correct but effectively disables the system's caches. Therefore, both JVMs mentioned above exploit the guarantees provided by the Java memory model [Gos+14, §17] [MPA05; Loc12] to reduce the number of required cache invalidations and writebacks. In the following, we use these JVMs as case studies for compiler-based (or VM-based) coherence based on a well-studied memory model.

The Java memory model (JMM) is built upon the notion of the *happens-before* relationship. The happens-before relation is a partial order. Certain actions, such as synchronization operations like acquiring and releasing a lock or accessing a volatile field, impose a happens-before order on program execution. More formally, following Gosling et al. [Gos+14, §17.4.5], we define the relation on actions that are part of an execution trace (see also Section 2.2.1). We say that a read action r of a variable v is allowed to observe a write action w to v if, in the happens-before partial order of the execution trace, (i) r does not happen before w, and (ii) there is no intervening write action w' to v, i.e., no w' so that w happens-before w'.

Practically speaking, this definition means that updates to heap objects can stay local to a thread, i.e., with values not visible to other threads, until the next synchronization point. Only then must changes become visible to other threads. For example, suppose a thread acquires a lock and changes some non-volatile fields, then these updates do not need to become immediately visible to other threads.

In terms of the formal definition, the write by the modifying thread and the reads by other threads are not ordered with respect to the happens-before relation. Only before releasing the lock must the thread make sure that all updates it has made are visible to any other thread that later acquires the same lock. The synchronization operations (acquiring and releasing), together with the order of the operations in the program, enforce a happens-before relationship. Hence, reads following an acquisition of the lock must be able to observe the changes made by other threads preceding their release of the lock (assuming no intervening writes).

On regular platforms, JVM implementations usually exploit these guarantees by holding updated values of heap objects in machine registers. They only perform potentially costly write operations to memory when required by the JMM, e.g., when releasing a lock. From the compiler's or virtual machine's perspective, synchronization operations restrict the mobility of certain memory-related actions. For example, the JMM forbids reordering a write followed by a release operation in the program, i.e., it is illegal to move the write after the release operation. Depending on the hardware memory model, JVMs may also need to issue memory-barrier instructions (also called memory fences) to prevent the hardware from performing illegal reorderings. See [How+16] and [BA08] for details on memory barriers.

On a non-cache-coherent architecture, synchronization operations must additionally trigger explicit cache operations. As described above, a thread acquiring a lock must be able to observe updates to the shared heap performed by another thread that held and released the same lock before. Thus, the JVM implementation of McIlroy et al. writes back and then invalidates the complete data cache whenever the current thread acquires a lock or reads a volatile field [MS10, section 5.3]. Before releasing a lock or writing to a volatile field, the implementation issues a writeback of the complete cache to make the changes visible to other threads. The Hera-JVM also performs explicit cache actions in other situations, such as context switches. Similarly, the DiSquawk JVM writes back and invalidates, i.e., flushes, any cached data before volatile accesses, and writes back dirty cached data directly after writes to volatile variables [ZP16b, section 3.3].

This raises the question of precisely characterizing the locations that require such cache actions. To this end, Zakkak et al. [ZP14] present the JDMM, the Java Distributed Memory Model, a formalization of the JMM for non-cache-coherent architectures. The JDMM extends the JMM with additional cache-related actions, in particular writeback and invalidation. Zakkak et al. then show that the JDMM adheres to the JMM. However, their formalization is not machine-checked. See [Loc12] for a machine-checked formalization of the JMM.

Tavarageri et al. [Tav+16] present a compiler-assisted approach for inserting necessary cache-coherence instructions into parallel programs. Their modified compiler inserts writeback and invalidation instructions as presented in Section 2.2.2.3. To avoid false-sharing problems, their approach requires per-word dirty bits for each cache line, which increases hardware overhead.

Tavarageri et al. differentiate between regular code, where control flow and data flow are known at compile time (mostly well-formed loops), and irregular code. For regular code, they use the polyhedral model [Bas04] to precisely identify locations for cache actions. For irregular code, they fall back to more conservative approximations, with invalidation or writeback of the complete cache as a last resort. Hence, their baseline approach is similar to the previously presented JVM-based approaches, however, they can exploit a more regular program structure to improve the precision of coherence actions.

Library-based approaches. Library-based approaches implement a software DSM system (see Section 2.4.1) on top of non-coherent shared memory. We refer to [Nür+14] for an overview of software DSM systems in the context of many-core architectures.

Prescher et al. [PRN11; Rot+12] implement library-based DSM for the Intel SCC. Their C++ library offers smart pointers that, in addition to the actual object address, save information required for cache management. Specifically, these smart pointers refer to *consistency controller* objects that manage the necessary invalidations and writebacks.

To avoid too frequent coherence actions, the smart pointers do not actually allow accessing the underlying shared object. Instead, the user has to create access-proxy objects that grant either read-only or exclusive write access. The actual cache operations are triggered on object construction and destruction of the access proxies.

Their library offers multiple strategies for coherence management. The simplest one uses one needs-invalidate flag on each core for each shared object. This flag signifies if another core has changed the shared object, which must therefore be fetched from main memory.

For example, when acquiring write access to a shared object by creating the matching access-proxy object, the consistency controller checks the local needs-invalidate flag. If it is set, the controller invalidates the shared object's memory range in the local cache. All subsequent accesses then cause cache misses and thus the up-to-date version of the shared object is fetched from main memory. After the core is done working with the object, it destroys the access proxy. The destruction triggers a writeback of the dirty data in the cache and at the same time sets the needs-invalidate flag on all other cores.

In comparison to the previously presented compiler-based approaches, this enables shared-memory programming on a much lower level. The programmer must be actively aware of the non-cache-coherent memory and must manage shared objects with the provided smart pointers and access proxies. If the programmer accesses a shared object using raw pointers, no coherence actions are triggered, potentially leading to subtle bugs. In summary, the proposed library-based software-DSM system is more flexible and potentially more efficient with the downside that it requires changes to the source program and is unsafe.

Operating-system-based approaches. Multiple projects modify Linux to run on non-cache-coherent architectures, while differing in their implementation details. For example, K2 [LWZ14] targets mobile systems-on-chip that consist of multiple but few, i.e., two or three, heterogeneous coherence domains. Popcorn [Bar+15] modifies Linux to run on platforms consisting of multiple OS-capable multi-core processors with different ISAs, such as a regular x86 multi-core extended with a PCIe-based Intel Xeon Phi processor.

Both operating systems provide transparent coherence via distributed shared memory implemented by managing coherence in software at page granularity. The key idea is to maintain the invariant that there is only one writer per memory page (typically of size 4 KiB). For example, K2 maintains a simple state flag, valid or invalid, per page and core. A simple protocol then ensures that at each point in time, each page is only valid on at most one core.

This protocol works as follows. A core can read or write a locally valid page. However, accessing a locally invalid page triggers a page fault. The page-fault handler notifies the current owner of the respective page to flush the page from the owner's cache to memory and then give up ownership by setting the local state of the page to invalid. Then, the core that caused the page fault becomes the new owner of the page, i.e., the page is now locally in the valid state and can be accessed by the core.

These approaches manage coherence in software on a level below the code generated by compilers. Hence, it is possible to reuse existing compilers and generated binaries that expect coherent shared memory. However, the granularity of coherence is coarse (whole pages), which can lower performance significantly depending on the access behavior.

Overall, we see that using the shared-memory programming model on non-cache-coherent many-core architectures is feasible. The necessary cache actions can be managed by a library or runtime system, by the compiler, or by the operating system. For the latter two, we do not need to adapt the code of existing parallel applications; for the last, even compilers can stay unchanged.

2.4.3. Message Passing

In the previous section we have seen how we can program a non-cache-coherent shared-memory machine using the shared-memory programming model. However, different programming models can be used as well. In the following, we look at the message-passing programming model in more detail.

As explained in Section 2.1, message passing is commonly used on machines that do not provide a shared physical address space, i.e., provide fewer capabilities than shared-memory machines. Therefore, it is not surprising that we can easily use message passing also on non-cache-coherent shared-memory machines.

The fundamental idea, shown in Figure 2.9, is to partition the shared address space and assign each partition to one coherence domain. This coherence domain is the sole owner of this part of the address space. Hence, only cores from the owning coherence domain access the respective address-space partition.

The cores may also have caches that cache memory contents from this partition. As the caches inside a coherence domain are coherent by definition, accesses to the same address by multiple cores from the same domain (if domains contain more than one core) do not cause problems. On the other hand, due to the partitioning of the address space, cores from separate domains never access a common address. Hence, we do not need global cache coherence. The address space partitioning happens on a logical level, for example in the programming model, i.e., by preventing the creation of pointers to foreign memory partitions in the programming language, or in the operating system, i.e., by not mapping foreign memory partitions into a domain's virtual address space.

Fundamentally, we hide the fact that the hardware actually provides a shared physical address space from the programmer. We disallow problematic memory accesses, i.e., to the same address from different coherence domains, on a logical level. By doing this, we shield the programmer from coherence-related problems.

This raises the question of how communication between coherence domains actually happens, i.e., how we implement the primitive operations

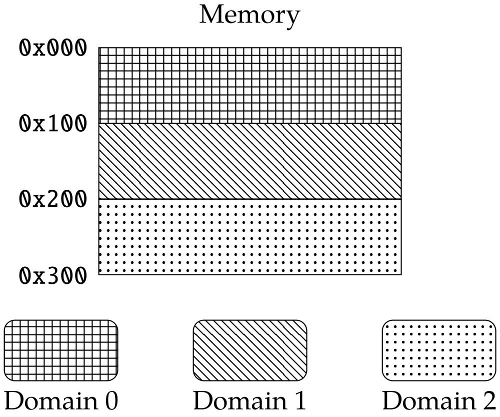


Figure 2.9: A non-cache-coherent shared-memory architecture with a partitioned address space. Each coherence domain, encompassing one or more cores with coherent caches, is assigned a partition of the address space.

`send()` and `receive()`. In the following, we will see that we can do this by exploiting the shared physical address space, potentially using specialized message-passing hardware if available. While our address-space partitioning prevents the programmer from using shared memory to transfer data between coherence domains, it does not prevent the compiler or runtime system from exploiting shared memory for implementing the needed message-passing primitives. We will see that we can apply the ideas from Section 2.2.2.3 to guarantee coherence.

For the sake of simplicity, we look at a two-sided synchronous communication operation, i.e., the sending domain S calls `send()`, the receiving domain R calls `receive()`, and both calls block until the transmission has been completed. Figure 2.10 shows our scenario. We assume that both `send()` and `receive()` take the message, i.e., a buffer of known length, as a parameter. We further assume that the message of length L bytes is located at address M in S's memory partition and should be copied to address M' in R's memory partition.

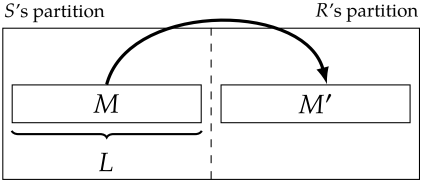


Figure 2.10: Transferring a message from sender S to receiver R. The message M shall be copied from S's address space partition to M' in R's partition.

As we have a shared physical address space, the address M' is also valid in S. Hence, we can use a core from S to load L bytes from M and store them to M'. However, because of the missing cache coherence, cores in R would not necessarily observe the correct values at M', as (parts of) the message could still be in local caches in S.

Thus, following Section 2.2.2.3, we need to write back all cache lines spanned by the message, i.e., all lines caching data from the address interval $[M', M' + L)$. Similarly, cores in R need to invalidate this address range in their local caches before reading the message to guarantee that they observe up-to-date values. This extends the ideas from Section 2.2.2.3 to address ranges in a straightforward manner.

The IBM Cell architecture (see Section 2.3.1) uses this model. On this architecture, each SPE has its own private local memory. To access data in main memory, the data must first be copied to the local memory. However, the local copy and the copy in main memory are not kept coherent automatically. Hence, after processing, the data must be copied back to main memory, which corresponds to a manual writeback operation. To speed up the copying, the hardware provides DMA units to copy data asynchronously. Multiple SPEs can communicate by copying data back and forth via the main memory.

In general, the problem with this approach is that the main memory may become a bottleneck as the number of cores increases. Additionally, communicating via off-chip memory has a comparatively high latency. As message passing is often regarded as the preferred programming model for non-cache-coherent shared-memory architectures [Kum+11], many of them provide additional memories dedicated to message passing that enable a more decentralized form of communication with significantly lower latency.

The idea is that each coherence domain has a small but fast on-chip memory that is visible in the global physical address space. Hence, we can implement `send()` by having the sender execute regular stores to the on-chip memory of the receiving domain; or, alternatively, by having the receiver execute regular loads from the on-chip memory of the sending domain. Thus, we omit using the off-chip main memory for every communication operation. For performance reasons, writes to remote on-chip memories are often cached as well. In this case, we have to manage coherence in software just as described above. We extensively discuss data transfers and their performance characteristics on non-cache-coherent architectures in Section 4.3.

As a concrete example, the Intel SCC provides message-passing buffers (MPBs, see Figure 2.4) [Mat+10, section III]. These MPBs are small (16 KiB) fast on-chip memories dedicated to direct communication between cores. Cores can load from and store to local and remote MPBs. Data from MPBs is cached in the L1 cache and the SCC provides a dedicated instruction for software-managed coherence of MPBs. We refer to Mattson et al.'s description [Mat+10, section V] for details. We also discuss this topic in Section 4.5.

Similarly, the Runnemede platform provides on-chip scratchpads that are part of the global address space. Additionally, it provides DMA units to accelerate data transfers. The hardware provides dedicated invalidation and writeback instructions. The Formic Cube uses the same approach. Additionally, it offers a faster message operation for very small transfers (a single 32-bit word).

We see that while the message-passing model does not expose shared memory to the programmer, the actual implementation of message-passing primitives exploits the physical address space of non-cache-coherent architectures and manages coherence in software.

2.4.4. The PGAS Model

The Partitioned Global Address Space (PGAS) model [Alm11; Sar+10; De +15] extends the shared-memory programming model to better handle the presence of distributed memory.

The fundamental observation is that shared-memory programming works worse as the cost of accessing remote data items increases. The different costs of local and remote memory accesses are often summarized by the "NUMA factor" in the literature (cf. NUMA architectures introduced in Section 2.1). The NUMA factor is the ratio of the cost of a remote memory access and the local memory access cost. Hence, a NUMA factor of 2 means a 2× slowdown when accessing remote data items.

Shared-memory programming works well if the NUMA factor is relatively low. However, as the NUMA factor grows, the illusion of a uniform address space becomes increasingly unrealistic as some data items are much more costly to access. At the same time, all data items and references look the same to the programmer, whether local or remote. This inability to reflect the properties of the underlying hardware can lower performance significantly, e.g., when accidentally accessing remote memory.

To counter this problem, the PGAS model adds a notion of data locality to the shared-memory programming model. Here, the programmer can and must explicitly manage the location of each piece of data. Additionally, references to local data items are explicitly distinguishable from references to remote data items, e.g., by having a different type.

However, unlike the message-passing model, the PGAS model still offers a global address space. This means that every process can point to every memory location, even if it is physically located in a remote memory. Yet, in contrast to the pure shared-memory model, now the notion of near and far memory is explicit, i.e., the address space is partitioned.

Figure 2.8 shows a schematic comparison of the shared-memory, the message-passing, and the PGAS programming models. We see that in a shared-memory model, as explained in Section 2.4.1, multiple threads run inside the same uniform address space. Every thread can have references to data items created by other threads. In a message-passing model, we express a computation using multiple processes with private address spaces that exchange data via messages. It is impossible for one process to reference data from another process; the address spaces are strictly separated. In the PGAS model, the address space of multiple processes is unified, as with the shared-memory model. However, the address space is not uniform; instead, it is partitioned to reflect the location of data items. Figure 2.8c uses different arrow types for references to local and remote data items, respectively.

Additionally, the PGAS model makes communication partially implicit. Accessing remote data items usually happens via simple assignment or dereference operations. Hence, the programmer does not have to explicitly insert communication operations, such as `send()` and `receive()` with message passing, into their program. Instead, it is the task of compiler and runtime system to perform the necessary communication to access a remote data item, e.g., exchanging messages on a hardware platform without a shared physical address space.

These properties make the PGAS model attractive to use on non-cache-coherent shared-memory machines. As such hardware offers a shared physical address space, shared-memory programming seems like a good fit in principle. However, the cost of accessing data from other coherence domains may be high due to the required communication, e.g., due to software-managed coherence. This makes the illusion of a uniform address space hard to maintain, which is exactly the problem tackled by the PGAS model.

Hence, the PGAS model maps naturally to non-cache-coherent architectures: we interpret locality not in terms of physical memory location but regarding coherence domains[5]. Thus, we first partition the address space as shown in Figure 2.9. Then, from the view of a coherence domain, local data items reside in the domain's own memory partition while remote data items reside in foreign memory partitions associated with other coherence domains.

The programmer still has most of the benefits of shared-memory programming, e.g., a global address space. However, at the same time, they are

[5] The physical memory location may (and for performance reasons should) coincide with coherence boundaries. Imagine a system like Figure 2.1b with a shared physical address space but no hardware-based coherence. Then, a coherence domain consists of a core with its (physically) local memory.

aware of potentially costly accesses to remote data from other coherence domains as locality is exposed in the programming model.

The PGAS programming model does not specify a particular implementation of remote data access. On message-passing hardware without a shared physical address space, accessing a remote data item triggers the sending of one or more messages. On non-cache-coherent architectures, we can therefore reuse the techniques from Section 2.4.3 to implement the required communication.

However, PGAS runtime systems usually prefer so-called one-sided communication [Mes15, chapter 11]. Here, one party specifies all necessary communication parameters, both for the sending side and the receiving side. Hence, on a non-cache-coherent architecture, a PGAS runtime system can be implemented using a message-passing mechanism. Its preferred mode of operation is, however, one-sided, which we can implement using explicit writebacks and invalidations on non-cache-coherent systems [CS16; CS17]. We discuss this in detail in Section 4.3.

We see that the PGAS programming model is a good candidate for use on non-cache-coherent architectures. By making the address space unified, it maintains many of the programmability advantages of the shared-memory model. By making the address space non-uniform, it exposes more of the hardware, in particular the existence of multiple coherence domains, which may improve performance. Moreover, implicit communication operations instead of explicit messaging operations reduce programmer burden.

Summary

- Coherence is a property of a system guaranteeing that the presence of caches never enables new or different functional behavior.
- Coherence can be maintained by hardware or software.
- A non-cache-coherent shared-memory system offers a shared physical address space but no hardware-based coherence.
- Multiple non-cache-coherent systems have been built or proposed for power and scalability reasons. They differ in the number and size of their coherence domains.
- Efficient implementation of the shared-memory programming model requires hardware support for fine-grained cache control to enable software-based coherence.
- Libraries, operating systems, or compilers can manage coherence in software.
- Efficient implementation of message passing benefits from fast on-chip memories.
- Both the message-passing and the PGAS programming model benefit from hardware support for coarse-grained software-managed coherence.

What is usually meant by the term craftsmanship is the production of things of high quality; Ruskin makes the crucial point that a thing may also be judged according to the conditions under which it was built.

Tracy Kidder, The Soul of a New Machine, p. 272

3 Invasive Computing

While the work presented in this dissertation is generally applicable to modern parallel architectures, the prototype implementations presented in Chapters 4 and 5 make heavy use of infrastructure developed in the context of the research project Invasive Computing [Tei+11; Tei+16]. This project investigates ways to improve the efficiency and predictability of resource usage on future many-core systems using a holistic approach that takes into account every system component, i.e., ranging from low-level hardware to high-level software.

In the following sections, we start by introducing the reader to the overall idea of Invasive Computing in an abstract way. Then, we provide a bottom-up view of the project starting from the hardware and then covering components such as system software, programming language, and compiler. We relate each component to the recent developments in the context of many-core architectures presented in Chapter 2. While we cover most parts of the invasive ecosystem, we describe in detail only the aspects relevant in the context of this dissertation. For details on other aspects, we refer the interested reader to the referenced material.

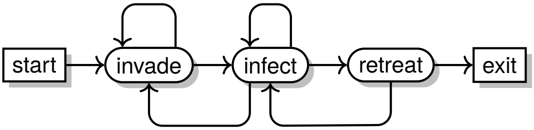


Figure 3.1: State chart of an invasive program, adapted from [Han+11].

3.1. The Invasive Paradigm

The two fundamental ideas of Invasive Computing [Tei+11] are (i) resource-aware programming, and (ii) exclusive resource allocation. Resource-aware programming means that programs can (and shall) examine the system state, relate this information to their computation needs, and then request a matching set of computing resources (such as cores, memory, or communication links) from the operating system, which distributes resources using its global system view. Exclusive resource allocation means that when resources are granted to a certain application, only this application is allowed to use them[6].

Hence, Teich et al. [Tei+11] define invasive programming as follows:

> Invasive Programming denotes the capability of a program running on a parallel computer to request and temporarily claim processor, communication, and memory resources in the neighborhood of its actual computing environment, to then execute in parallel the given program using these claimed resources, and to be capable of subsequently freeing these resources again.

The goal of invasive programming is to optimize the overall efficiency of resource usage in a parallel system. Exclusive resource allocation avoids paying the overheads for resource virtualization, while resource-aware programming enables exploiting application-specific knowledge to guide resource distribution among multiple running applications.

Figure 3.1 shows the life cycle of a program that follows the invasive paradigm. Initially, the program inspects the current system state, de-

[6] At least as the default behavior.

termines a sensible set of initial resources, and then issues a resource request, called *invade*, to the system. If the request is granted, the system responds with a *claim* that contains resources exclusively allocated to the application. The program then uses these newly claimed resources in a phase called *infect*.

Every time the program reaches a point where it is possible and sensible to adapt its set of resources, the program should inform the system. The program may do that using *retreat*, which releases the claim's resources, or using (re-)invade, which potentially changes the claim and enables the system to redistribute resources. Following the resource-aware paradigm, each time a program wants to change its claim it should first analyze the system state, figure out a sensible resource change, and then send a request to the system via (re-)invade. Once a program retreats from all its claims, it terminates execution and exits.

Programs formulate resource requests in a sophisticated constraint language [ZBS13]. The language allows expressing multiple alternative resource requests (using a logical-or construct) and ranges (e.g., requesting 1 to 10 cores). Hence, granting a resource request is not necessarily a binary decision. Additionally, constraints allow passing application-specific knowledge (e.g., about scaling behavior) to the system. Thus, the system ideally has many degrees of freedom when distributing resources among multiple running applications. Furthermore, it has the necessary information about applications to make a globally sensible decision. The actual resource distribution is performed in a decentralized way to ensure scalability [Kob+11].

We can also exploit exclusive resource allocation to optimize for goals other than efficiency. For example, exclusive allocation enables precise control over interference with other applications. Hence, it can also significantly simplify reasoning about non-functional properties, such as timing predictability [Wil+16].

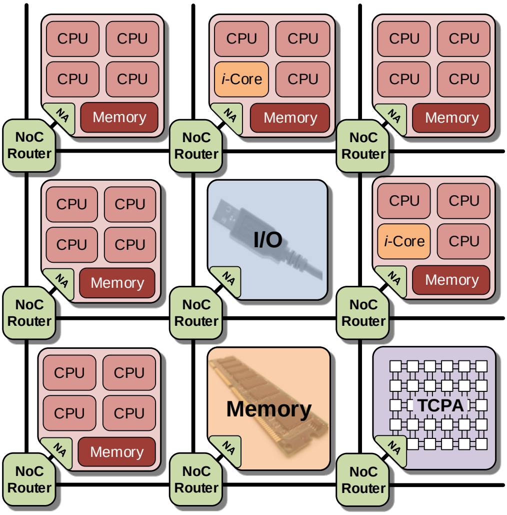


Figure 3.2: A 3 × 3 design consisting of six compute tiles, one memory tile, one I/O tile, and a specialized accelerator tile.

3.2. Hardware Architecture

Invasive hardware architectures [Hen+12] are a family of heterogeneous many-core architectures. Fundamentally, they are partially non-cache-coherent shared-memory architectures with distributed memory. To ensure scalability to high core counts, they provide cache coherence only for small groups of cores, but not between core groups. Figure 3.2 shows an example of such an architecture.

The architecture's basic building block is a *tile*: a standard compute tile consists of relatively few general-purpose cores (four in the example) that share some resources, such as an L2 cache or a small on-chip memory (*tile-local memory (TLM)*). Most importantly, cache coherence is guaranteed between the cores of a tile. In general, the number of cores inside a tile must be low enough so that classical hardware coherence protocols, such as bus snooping, are still applicable. Thus, a single tile behaves exactly like a common cache-coherent multicore processor and can be programmed using the traditional shared-memory programming model.

Multiple tiles can be combined to create a larger system. Each tile contains a network adapter [Zai+15] ("NA" in Figure 3.2), which connects the tile to a scalable network-on-chip [Hei+14] (NoC) that transfers data between tiles. However, the hardware provides no cache coherence between different tiles.

While this improves scalabilty, it raises the question of how to best communicate between tiles. The designated communication means is message passing. To this end, the architecture provides a shared physical address space across all tiles. In particular, all tile-local memories are visible. Thus, to send a message, a core stores the data into the receiving tile's TLM. In this case, coherence must be managed in software (as explained in Section 2.4.3). Additionally, the NoC and network adapter provide hardware-accelerated DMA transfers. DMA transfers asynchronously copy a block of data from the sending tile's TLM to the receiving tile's TLM (we present more details on DMA transfers in Sections 3.3 and 4.3).

Invasive architectures provide hardware support for important higher-level operations. Inside a tile, a special hardware unit, the *Core* i-*let Controller* (CiC), accelerates scheduling tasks to a tile's cores [Rav15]. Besides improving scheduling and dispatching throughput, the C*i*C's latency is low enough to be able to consider various sensor values, such as power or temperature readings. For example, it can schedule a task to the coolest core. To simplify and accelerate communication between tiles, the network adapter provides hardware support for starting tasks on remote tiles [Zai+15]. Network adapter and C*i*C cooperate closely; they can schedule and dispatch a newly started task on a remote tile without operating system assistance.

Not all tiles are (pure) compute tiles. Invasive architectures may also include memory tiles and I/O tiles. Memory tiles are connected to off-chip DRAM, which, in general, holds most of a program's data. This memory usually makes up the bulk of the shared physical address space. I/O tiles provide access to peripheral devices, such as networking. Cores from other tiles use these resources by accessing them over the NoC.

Furthermore, not all compute resources are homogeneous. Invasive architectures may include specialized hardware, both in the form of individual specialized cores or whole specialized tiles. An example of the former is the *i*-Core [Bau09]. The *i*-Core consists of a regular core extended with an FPGA-based fabric. It allows loading accelerator modules onto the FPGA, which can be used by the core via special instructions exposed as an instruction set extension. The *i*-Core behaves like a regular core as long as the regular instruction set is used. Applications aware of its capabilities can use the special instructions to benefit from hardware acceleration.

Alternatively, complete tiles can be dedicated to accelerators. For example, such accelerator tiles may contain tightly-coupled processor arrays [Han+14] (*TCPAs*). TCPAs consist of processor elements with a domain-specific instruction set that are arranged in a 2D grid and connected by a low-latency network. They are particularly well-suited for computationally intensive applications from domains such as image or signal processing. Accelerator tiles are connected to the rest of the system via a regular control processor that is part of the tile (not shown in Figure 3.2). The control processor receives input data, initiates computations, and sends back the results once the computation on the accelerator has completed.

3.2.1. Related Work

Invasive architectures are non-cache-coherent shared-memory architectures and thus closely related to the architectures described in Section 2.3.1.

When comparing invasive architectures to the Intel SCC, both share many properties: they are tile-based, with multiple cores on each tile sharing some resources; they employ a network-on-chip as their scalable interconnect; and they encourage message passing by providing fast tile-local memory. However, there are also important differences. Unlike invasive architectures, the Intel SCC does not guarantee cache coherence inside a tile, i.e., it is a fully non-cache-coherent architecture. In contrast, invasive architectures are partially cache-coherent. Thus, invasive architectures suggest a hybrid programming model using shared memory inside a tile and message passing between tiles, while the Intel SCC was designed as a pure message-passing platform. Moreover, the SCC is homogeneous, whereas invasive architectures may contain specialized cores or tiles, using a different (or extended) instruction set.

When looking at the Intel Runnemede, we find similar commonalities and differences. The Runnemede also has a tile-based structure; scalable interconnects; and fast tile-local memories. In contrast to invasive architectures and the Intel SCC, Runnemede is strongly heterogeneous and asymmetric inside a tile. Here, only one core per tile is capable of running an operating system and all other cores are specialized accelerators that possibly use a different instruction set.

The EUROSERVER architecture is another partially non-cache-coherent architecture. In contrast to invasive architectures, EUROSERVER is homogeneous and has a more uniform memory hierarchy, with one DRAM module per tile and no on-chip memories. Most importantly, it proposes the use of restrictive caching policies to avoid incoherent situations. Thus, it does not require software-managed coherence.

3.3. Operating System

As the programming paradigm and the architecture proposed by Invasive Computing place new demands on the operating system, a novel operating system has been developed in the scope of the research project. OctoPOS [Oec+11; Moh+15] is an operating system designed specifically for non-cache-coherent shared-memory architectures, including the family of invasive architectures as described in Section 3.2. The primary design goal of OctoPOS is to exploit fine-grained parallelism in applications directly on the operating-system level. Additionally, it integrates resource-distribution functionalities needed for resource-aware programming.

OctoPOS offers an execution model that is more lightweight than the traditional UNIX model of processes and threads. The principal idea is that the operating system represents parallelizable control flows not as coarse-grained, long-running threads with preemption, but as short snippets of code called i-*lets*[7]. An *i*-let consists of (i) a pointer to a function to be executed, and (ii) a piece of data passed as an argument.

A typical parallel application running on top of OctoPOS splits its work into many packages, creates an *i*-let for each work package, and hands these *i*-lets to the operating system for execution. The OS scheduler distributes the *i*-lets to the available CPU cores, where they are processed sequentially. Like user-level threads, *i*-lets use cooperative scheduling. For *i*-lets that run to completion, creation and dispatching are very efficient because the respective execution contexts (i.e., stacks) can simply be reused. The OS performs a costlier context switch only if an *i*-let performs a blocking operation.

Using cooperative scheduling becomes possible by exploiting the exclusive resource-allocation scheme of Invasive Computing. Following Section 3.1, all resources of an application, including CPUs, belong to its claim. Claims are a central data structure in OctoPOS. The scheduler distributes an application's *i*-lets only to CPUs in its claim. Thereby, it enforces spatial separation of concurrently running applications. Hence, no preemption is necessary, as applications have full control over their core set anyway.

OctoPOS follows a multikernel design [Bau+09]. On an invasive architecture, each tile runs a separate instance of the operating system. Internal state is replicated on each tile. The instances communicate via message passing, e.g., to synchronize the initial system boot process.

Following the hardware/software codesign approach of Invasive Computing, the hardware provides dedicated support for *i*-lets. To this end, the *i*-let format has been fixed to include a function pointer and two 32-bit data words. The data words can hold by-value arguments, or pointers in case of larger input data, which is then transmitted separately.

Applications spanning more than one tile, i.e., more than a single coherence domain, can communicate using the following two OS-level primitives.

Remote *i*-let spawning. Code execution on a remote tile is triggered by sending a fixed-size packet containing an *i*-let over the NoC. On the

[7] *i*-let is short for "invasive-let", inspired by the term "servlet" [Tei+11].

receiving side, the *i*-let is inserted into the regular scheduling queue and executed asynchronously to the sender's control flow.

As this is a frequent operation, OctoPOS can exploit special hardware provided by an invasive architecture to accelerate it. In fact, the cooperation between NoC and C*i*C as described in Section 3.2 is based on *i*-lets. Thus, when the NoC has transmitted an *i*-let to a remote tile, the network adapter directly hands the *i*-let to the C*i*C for scheduling and dispatching on the tile's cores. Following the invasive paradigm from Section 3.1, spawning a remote *i*-let corresponds to infecting the respective claim.

Push-DMA transfer. To allow transferring larger chunks of data between tiles, OctoPOS offers a push mechanism that allows copying an arbitrarily large contiguous memory region to a buffer in another tile's local memory. The receiving tile is guaranteed to have a coherent view of the destination buffer after the transfer has completed. The operation is performed asynchronously as a DMA transfer, allowing the sending process to continue work without blocking. The caller of a push-DMA operation can optionally pass a pair of *i*-lets along with the data:

1. The first *i*-let will be executed on the sending tile once the operation has completed, and can be used for releasing the source buffer or for implementing custom blocking if desired.
2. The second *i*-let will be spawned on the receiving tile, where it can begin processing the transferred data.

Again, as this is an important and frequent operation, invasive architectures provide special hardware support. After triggering the DMA operation, the hardware completely handles the data transfer as well as the dispatching of local and remote *i*-lets. This is achieved by cooperation of NoC, network adapter, and C*i*C.

Synchronization. For the synchronization of *i*-lets, OctoPOS offers a lightweight barrier-like concept called signal, which is optimized for a fork-join scenario. The standard pattern in this scenario is one *i*-let that spawns multiple other *i*-lets for parallel work, and then waits for their termination. An OctoPOS signal is initialized with a counter value equal to the number of jobs. After creating the jobs, the spawning *i*-let invokes the `wait()` primitive, which blocks until the counter reaches zero. Each job does its work and afterwards calls `signal()`, which decrements the counter by one. If the number of jobs is not known in advance, `add_signalers()` can be called for new *i*-lets created dynamically to increment the counter.

OctoPOS signals are similar to blocking semaphores, but more lightweight: Only a single *i*-let per signal is allowed to wait, so there is no need for a waiting queue. Activities that were spawned on another tile can signal back to their original tile by sending an *i*-let that performs the signaling.

In summary, OctoPOS is an operating system for non-cache-coherent architectures that implements a lightweight *i*-let-based execution model and offers asynchronous operations for cross-tile data transfers and task spawning. On invasive architectures, the most important operations are accelerated by dedicated hardware units.

Resource management. As we allocate resources exclusively, we need a way to adapt the resources of an application to its needs. Otherwise, resources sit idle or applications cannot fully exploit their inherent parallelism. Therefore, in Invasive Computing, applications are expected to inform the system of changed resource requirements. To avoid a single bottleneck, resource management proceeds in a distributed fashion.

Each application is represented by an *agent*. Every resource request of an application goes via its agent. Agents communicate in a distributed manner and bargain for resources [Kob+11]. After the bargaining has finished, the agent then notifies the application of the result. The agent system is part of the operating system. Together, OctoPOS and the agent system form the invasive runtime support system (*i*RTSS).

3.3.1. Related Work

The Barrelfish operating system [Bau+09] aims into a similar direction as OctoPOS as it pioneered the idea of using multikernels on shared-memory many-core architectures. Hence, as with OctoPOS, multiple OS instances communicate via message passing. The use of Barrelfish on non-cache-coherent architectures has been investigated using the Intel SCC as a platform [Pet+11b]. Unlike OctoPOS, however, the Barrelfish kernel implements a traditional, heavyweight threading model.

Multiple projects modify Linux to run on non-cache-coherent architectures, while differing in their implementation details. For example, K2 [LWZ14] targets mobile systems-on-chip that consist of multiple but few, i.e., two or three, heterogeneous coherence domains. K2 also runs one kernel per coherence domain. However, K2 uses a "shared-most" approach that replicates most OS services in all coherence domains but maintains state coherence. K2 provides transparent coherence via distributed shared memory (see Section 2.4.1) implemented by managing coherence in software at page granularity.

Popcorn [Bar+15] modifies Linux to run on platforms consisting of multiple OS-capable multi-core processors with different ISAs, such as a regular x86 multi-core extended with a PCIe-based Intel Xeon Phi processor. In this scenario, the fundamental idea of Popcorn is not to view the Xeon Phi as a coprocessor used by offloading but to view both processors as a unit. Popcorn does not assume cache coherence and thus uses replicated OS kernels per coherence domain. In case of missing hardware-based cache coherence, Popcorn provides software DSM. Other than K2, Popcorn follows the shared-nothing principle.

Unlike OctoPOS, both K2 and Popcorn are designed to execute regular programs written in a shared-memory style and thus provide software DSM. Instead, OctoPOS exposes the structure of the underlying hardware platform to applications. Thus, they must be able to cope with non-cache-coherent shared memory.

Gruenwald et al. [Gru+15] present Hare, a file system for non-cache-coherent many-core architectures. They manage coherence in software by using a protocol based on invalidations and writebacks. Hare can be integrated into an operating system to provide a shared file system even on architectures without hardware-based cache coherence.

Related to OctoPOS's execution model, project Runnemede [Car+13; Zuc+11; SZG13] introduces codelets, which are similar to *i*-lets and are supported directly by the operating system [KCT12]. Codelets are small self-contained units of computation with run-to-completion semantics assumed by default. Similar to *i*-lets, codelets can still be blocked if need be. In contrast to *i*-lets, codelets are expected (but not required) to work functionally, i.e., to only work locally without leaving state behind and with their output only depending on the input values.

Additionally, the communication patterns between codelets are restricted. Codelets are arranged in a codelet graph according to their data dependencies, and act as producers and/or consumers, making them similar to dataflow actors in a dataflow graph. Hence, Runnemede makes parallelism more explicit and gives the runtime system additional optimization opportunities. However, programs must either be written in a codelet style in the first place, or a sophisticated compiler is required that decomposes programs written in traditional programming languages into codelets.

3.4. Programming Language

We investigated possible programming models for non-cache-coherent architectures in Section 2.4. In Section 3.3, we saw that OctoPOS exposes the underlying hardware's properties to the application, i.e., it does not implement a software DSM system as described in Section 2.4.2. Therefore, the preferred programming model in the scope of Invasive Computing is the PGAS model. The programming language X10 [Sar+16] developed by IBM was chosen as a modern representative of this class of programming languages.

Since the reader may not be familiar with X10, we give a short overview of the language. We discuss relevant language features in more detail in Section 4.2 (and following) and refer to the X10 language specification [Sar+16] for in-depth information.

At its sequential core, X10 is a statically-typed object-oriented imperative programming language with garbage collection. It supports a functional programming style with first-class functions and closures. X10 borrows its syntax from Scala [Ode14]. Restricted to its sequential core, X10 offers very similar features to Java [Gos+14]. However, there are a few notable differences that we show in Figure 3.3 and briefly explain in the following.

- X10 offers constrained types [Nys+08], a form of dependent types. Constrained types allow to statically express additional information about values. For example, the constrained type `String{self!=null}` is the type of non-null references to `String` objects. Constrained types integrate with subtyping in the natural way, i.e., `String{self!=null}` is a subtype of `String`, but not vice versa.

- X10 has local type inference for method return types, and for variable declarations using the keyword `val`. For example, `val x = 42;` is a valid statement where `x` has type `Long`[8].

- X10 permits operator overloading and, as of version 2.6, also the overloading of control structures [MMT16]. For example, the type `Complex` for complex numbers overloads all common arithmetic operators.

- X10 offers user-defined value types, using the `struct` keyword. Hence, variables of this type are *not* implicit references to a value of the type but directly contain the value. As a concrete example, the type `Array[Complex]`, where `Complex` is a value type, can be represented in memory as a sequence of `Complex` objects in contrast to a sequence of *references* to `Complex` objects as would be the case for non-value types.

More importantly, however, X10 is a *parallel* programming language: it directly supports programming both shared-memory and distributed-memory systems. It employs a language-based approach to concurrency and distribution, so the programmer writes parallel applications using first-class language constructs rather than using libraries or compiler directives.

As a side note, technically, the term "distributed-memory parallelism" is a misnomer. As we have seen in Section 2.4, using message passing, which is what distributed-memory parallelism usually refers to, does not require the existence of distributed memory in the sense of Section 2.1. However, these terms are ubiquitous in the literature, so we use them in the following as well.

[8] More precisely, `x` has the constrained type `Long{self==42}`.

```
struct T {
  val x: Int;
  def this(x: Int) { this.x = x; }
  operator this + (t: T) { return T(this.x + t.x); }
}

class Seq {
  def foo() { return 21; }
  public static def main(args: Rail[String]) {
    val s: Seq{self!=null} = new Seq();
    val t = T(s.foo());
    val r = (t + t).x;
    Console.OUT.println(r);
  }
}
```

Figure 3.3: Sequential X10 program highlighting key differences to Java. The program's output is 42.

3.4.1. Shared-Memory Parallelism

For shared-memory parallelism, X10 provides *activities* [Sar+16, §14]. An activity is a lightweight thread. Hence, in general, the programmer should not be worried about creating too many activities. Again, the reader may look at Figure 3.4 to get an intuition of parallel shared-memory programming in X10 and we explain the constructs it uses in the following.

Initially, every X10 program runs inside a single root activity. The programmer can create additional activities using the **async** keyword. For any statement S, **async** S is a statement and spawns a new activity that executes S asynchronously while execution of the original activity continues. If an activity a_1 spawns activity a_2, we say that a_1 is the parent of a_2 and a_2 is a_1's child. The set of all running activities created by a given X10 program together with the is-parent-of relationship forms a tree.

```
class SharedMem {
  public static def foo() {
    async Console.OUT.println("foo");
  }

  public static def main(args: Rail[String]) {
    finish for (i in 1..10) {
      async Console.OUT.println(i);
      foo();
    }
  }
}
```

Figure 3.4: X10 program exploiting shared-memory parallelism. The program outputs the numbers 1 to 10 and ten copies of the string `foo` in a non-deterministic ordering.

X10 distinguishes between local and global termination of a statement [Sar+16, §14]. The execution of a statement by an activity terminates *locally* when the activity has finished all computation related to the statement. For example, the statement **async** *S* terminates locally as soon as the new activity has been created. The execution of a statement by an activity terminates *globally* when the statement has terminated locally and all activities, which the statement may have spawned, have terminated globally. For example, assume the statement **async** *S* creates an activity a. Then, a terminates globally only when all its (transitive) children have terminated globally.

The statement **finish** [Sar+16, §14.3] converts global to local termination. Hence, **finish** *S* terminates locally when *S* has terminated globally. This means that an activity executing **finish** *S* waits for all its (transitive) children to terminate globally, before it terminates locally. There is an implicit **finish** statement surrounding the body of an X10 application's main method.

Applied to the example from Figure 3.4, we see that starting an activity to print `i` terminates locally as soon as the activity has been created. The same applies to the activity containing the print statement in method `foo`.

Hence, the loop statement terminates locally as soon as all activities have been created. The enclosing **finish** then waits until these activities have all terminated globally, i.e., the **finish** block is only left after the complete output has been printed. In summary, **finish** allows synchronization at an arbitrary level in the tree of activities created by a program.

X10 provides additional support for synchronization between activities in the form of unconditional and conditional atomic blocks [Sar+16, §14.7], as well as barriers called clocks [Sar+16, §15]. We refer to the language specification for more information on these constructs.

3.4.2. Distributed-Memory Parallelism

For distributed-memory parallelism, X10 provides the concept of *places* [Sar+16, §14]. A place is a set of computing resources, i.e., data and activities that operate on the data, that behave like a shared-memory system. Places introduce a notion of locality: accessing a piece of data local to a place has the same cost for all activities running on that particular place. Accessing remote data on other places may take significantly (orders of magnitude) longer. See Figure 3.5 to get a feeling of distributed-memory parallelism in X10; we explain the constructs it uses in the following.

```
class DistMem {
  public static def main(args: Rail[String]) {
    finish for (p in Place.places()) at (p) async
      Console.OUT.println("Place " + here.id);
  }
}
```

Figure 3.5: X10 program exploiting distributed-memory and shared-memory parallelism. The program outputs the string(s) `Place` i, where i depends on the number of places, in a non-deterministic ordering.

X10 exposes this locality to the programmer, so they must explicitly manage the place where they store each piece of data. The user cannot create places themselves; there either exist a fixed number of places throughout the execution of a program, or the runtime environment changes the number of places [Bra+14; IBM14]. The programmer sees places as instances of type `x10.lang.Place`. Each place has a unique id; execution starts on the designated place `Place.FIRST_PLACE`. The special variable **`here`** always refers to the place that executes the current activity (similar to **`this`**).

X10 provides the place shifting operation **`at`** [Sar+16, §13.3] to perform computations on other places. The **`at`** operation is a synchronous operation and does not spawn a new activity. Instead, the current activity changes its place of execution to the target place, continues executing there and, after it has terminated locally, control flow changes back to the original place.

The **`at`** operations exists in both statement and expression form. Hence, in addition to usage as in Figure 3.5, X10 also allows

```
val res = at (p) compute();
```

to call the method `compute()` on place `p` and receive the result in the local variable `res`. If evaluating an **`at`** statement or expression requires additional values (e.g., if `compute` required arguments), necessary values are copied to the respective place before the statement or expression is evaluated. We will discuss this in more detail in Section 4.4.

Synchronization using **`finish`** works across place boundaries. In the example from Figure 3.5, the **`finish`** waits until the print operation on all places has finished.

To directly support the PGAS model, X10 provides the generic type `GlobalRef[T]` as part of its standard library. `GlobalRef[T]` allows to refer to values of type `T` that are (potentially) located on other places. The application operation (implemented as an overloaded **`operator`**`()`) allows accessing the value referenced by a `GlobalRef`.

```
public static def foo(g: GlobalRef[String]) {
  val s = at (g.home) g();
  // ... use string ...
}
```

In this example, we see the definition of a method that takes as a parameter a global reference `g` to a string object. We then retrieve the referenced string value by using `g()`. Before we can access the value referenced by `g`, we must use **`at`** to shift to the place where the value lives. Each `GlobalRef` provides this place via the property `home`. The X10 type system enforces that we only access values of `GlobalRefs` on their respective home places.

3.4.3. Related Work

In general, X10 is related to all programming languages following the PGAS model. We refer to [Alm11; De +15] for a comprehensive overview and restrict our brief discussion to a few selected languages. We base our presentation on [Cha+05] and [De +15].

X10's features for shared-memory parallelism are similar to Cilk [Blu+95]. Where X10 has `async` and `finish`, Cilk has `spawn` and `sync`. However, X10's constructs are more general due to the distinction between local and global termination described in Section 3.4.1. This distinction allows the parent activity to terminate while its children are still running.

Concerning distributed-memory parallelism, X10 falls in the group of languages designed as part of the High-Productivity Computing Systems project initiated by DARPA. Other languages developed as part of this project are Chapel [CCZ07] and Fortress [All+05]. All these languages integrate the PGAS model in the language itself instead of merely providing it via a library. Additionally, in contrast to earlier PGAS languages or libraries, they use the so-called asynchronous PGAS model. They abandon the traditional SPMD model, used, e.g., in MPI, in favor of a model where programs can spawn new threads dynamically and each thread can execute different code.

X10 shares some characteristics with Chapel. The concept of a place is similar to locales in Chapel. However, in Chapel, objects can migrate between locales, whereas in X10, an object is bound to a particular place throughout its lifetime.

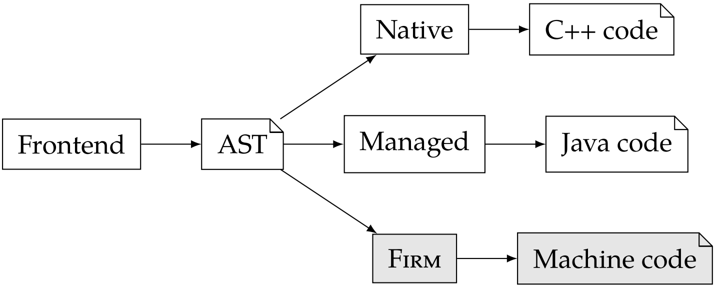


Figure 3.6: Structure of the modified X10 compiler. Adapted components are highlighted gray.

3.5. Compiler

The X10 programming language originally aimed at clusters [Cha+05], which combine a large number of multi-core machines via an interconnect network. Partially non-cache-coherent architectures exhibit a similar structure, although situated on a single chip. Thus, while X10 is a good match for partially non-cache-coherent architectures in principle, its compiler and runtime system were adapted in the scope of the Invasive Computing project to the non-standard hardware platform and operating system interfaces.

The existing X10 compiler developed by IBM is a source-to-source compiler. Figure 3.6 shows that it provides two code-generation backends: Managed X10 [Tak+11] translates X10 to Java, and Native X10 [Gro+11] translates X10 to C++. A post-compiler then generates bytecode or an executable, respectively. In the scope of the Invasive Computing project, a third backend was added that does not take a detour via another high-level language.

The new backend [Bra+12] targets the intermediate representation Firm [Fir17; BBZ11]. Firm is a graph-based intermediate representation (IR) designed for use in optimizing compilers. Its abstraction level and goals are similar to those of LLVM [LA04]. We provide more details on Firm in Appendix A.2.

The existing compiler pipeline could be reused up to and including the semantic-analysis phase. Using the new backend, the compiler then translates the resulting attributed abstract syntax tree (AST) into a Firm representation.

While the translation is straightforward for most AST structures, some constructs caused issues. The main cause of these issues was that both existing backends compile X10 to another high-level language of a similar level of abstraction. However, common compiler intermediate languages model programs on a significantly lower level than C++ or Java. Hence, the existing compiler made some assumptions about the target language that did not hold for Firm.

In the following, we briefly discuss two required major modifications to the X10 compiler concerning (i) the compilation of generic classes and methods, and (ii) the handling of native methods. The modifications are not specific to Firm but required for every target language of a similar abstraction level. We refer to [Bra+12] for details.

3.5.1. Compilation of Generic Classes and Methods

One important feature of modern programming languages is support for generic programming. For this purpose, Java and C++ offer generics and templates, respectively. Therefore, the existing X10 compiler backends can map X10 generics to Java's and C++'s available language mechanisms. The Java or C++ post-compiler then takes care of compiling the generic code.

However, on the abstraction level of intermediate representations like Firm, no support for genericity exists. Hence, in contrast to the existing backends, for our new backend, we have to handle generic classes and methods within the X10 compiler itself instead of leaving the handling to the post-compiler. In the following, we will briefly explain our strategy for handling generic methods and classes (referred to as "generic entities" in the following).

```
public class C {
  public static def id[T](x: T): T = x;
  public static def foo() {
    id(42);
    id("Hello");
  }
}
```

In this example, id is a generic method that implements the identity function. It is called twice in the program, once with T = Int and once with T = String. The fundamental question when generating code for generic entities is whether

(i) to differentiate at compile time and generate multiple specialized monomorphic versions that each work for a single argument type, in our example two versions of id for T = Int and T = String; or

(ii) to generate one polymorphic version that works for all argument types and distinguishes between different argument types at run-time.

Option (i) is called *expansion* in the literature [AP03]. In general, expansion offers the best performance, but can lead to significantly increased code size due to many specialized monomorphic versions. On the other hand, option (ii) trades decreased code size for increased run-time cost. There are multiple implementation techniques for option (ii), see [AP03, section 16.3] and [App97, chapter 16] for details.

We chose to expand generic entities, which has substantial performance advantages in the context of X10. For example, arrays are not built into X10, but exist as the generic class x10.array.Array[T] (or x10.lang.Rail[T] as of version 2.4) as part of the standard library. Arrays are fundamental in many applications and must therefore be as efficient as possible. Generating specialized code versions allows the compiler to generate maximally efficient code for ubiquitous types, such as Array[Int]. Additionally, it allows efficient arrays of value types defined via **struct**.

Expansion of generic entities in the X10 compiler itself (instead of using a post-compiler) required significant changes to the compilation process. We follow an implicit instantiation approach and expand generic entities as needed. Hence, the compiler does not generate any code when encountering the definition of a generic entity. Only when the compiler encounters an instantiation of a generic entity with a previously unseen combination of type arguments is a new specialized code version generated. In our example from above, the compiler would create code versions for `id[Int]` and `id[String]` when encountering the respective uses in method `foo`.

For code generation, we keep a single AST of each generic entity, which refers to uninstantiated type variables, such as `T` in our examples. For each specialized code version, we then set up a new context that maps `T` to the requested concrete type. This way, the same AST is traversed multiple times in different contexts, each time generating a different monomorphic code version.

3.5.2. Handling of Native Methods

Languages often declare some methods in their standard library as **`native`** [Lia99]. Usually, this applies to methods that need to access system or hardware resources, such as dealing with file I/O or accessing network devices, and the required system interfaces are not directly accessible in the language itself. In these cases, the compiler (or virtual machine) provides an implementation of the needed functionality in another language (typically C) and then takes care of rerouting calls to the native method to the actual implementation.

```
@NativeRep("c++", "int")
public struct Int {
  @Native("c++", "((#0) + (#1))")
  public native operator this + (x:Int): Int;
}
```

Listing 3.1: Excerpt from the definition of `x10.lang.Int`. The annotation syntax has been slightly simplified for presentation reasons.

Listing 3.1 shows that X10 uses this approach extensively in its standard library. We see that X10 defines even basic types, such as `Int`, in its standard library and defines all operations on them, such as integer addition, as native methods. In this case, X10 overloads the operator + to provide the familiar addition syntax.

This strategy has the advantage that there are fewer special cases in the compiler itself, e.g., all operations on data types are represented as method calls in the AST. However, even if we represent basic arithmetic operations, such as addition, as methods, we do not want to actually call a method to add two integers. The existing backends use annotations as shown in Listing 3.1 to directly map fundamental types and their methods to existing primitive types and operations in the target language. The annotation syntax has been slightly simplified for presentation reasons.

We could have applied the same strategy to our new backend by adding FIRM-specific annotations. However, as FIRM operates on a significantly lower level than C++ and Java, some operations would be very cumbersome to express and the resulting annotations difficult to maintain.

Instead, we implemented a form of link-time optimization, enabling cross-language optimization. As shown in Figure 3.7, we implement all native parts of X10's standard library as C functions, translate these implementations to FIRM using the existing C frontend, and then combine the resulting FIRM IR with the FIRM representation of the X10 program. We create the FIRM representation of the standard library only once during build time of the compiler and then load it when compiling an X10 program.

To briefly illustrate how this process works, take the type `Int` from Listing 3.1 as an example. We provide a C implementation of a function returning the sum of its two integer arguments. After combining the FIRM IR of this function with the FIRM representation of the X10 program, we have both the uses and the definition of the addition function available in the same format (i.e., FIRM graphs). Hence, regular function inlining can inline the body of the addition function (originating from C code) at the call sites (originating from X10 code) leading to efficient code, in this case the desired single machine instruction for an integer addition. This approach offers the high flexibility and compactness of providing implementations in a high-level language like C while making it unnecessary to introduce special cases or annotations.

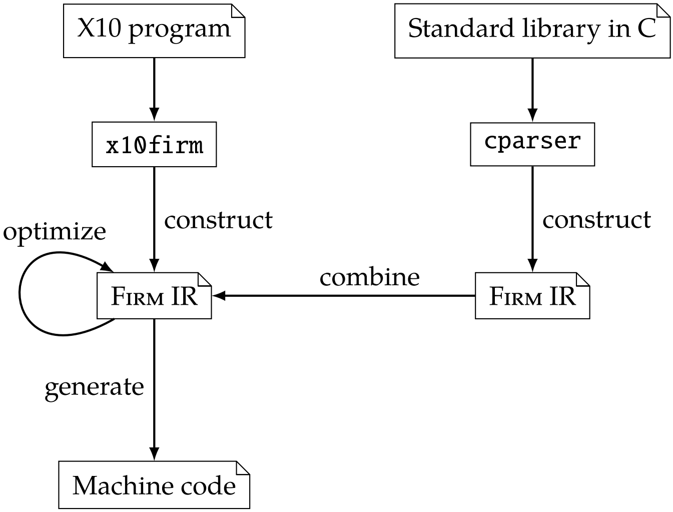


Figure 3.7: Structure of the modified X10 compiler.

3.6. Hardware Prototype

Section 3.2 presents invasive hardware architectures as a family of heterogeneous many-core architectures. In the scope of the Invasive Computing project, multiple instances of this architecture family have been built as FPGA prototypes [Bec+; Fri16]. As we use one of these prototypes for our evaluation in Chapter 4 and a derived prototype platform for our evaluation in Chapter 5, we give a brief overview of the platform's characteristics.

Figure 3.8 shows the structure of the prototype platform. The architecture consists of 3 homogeneous compute tiles with 4 cores each and one memory tile. Each tile forms a coherence domain and guarantees cache coherence via a classical bus snooping protocol. However, there is no cache coherence between tiles.

Inside a compute tile, all cores are Gaisler LEON 3 [Cob17b] processors. The LEON 3 is a 32-bit RISC processor that implements the SPARC V8 instruction set [SPA92]. On the FPGA prototype, each core runs at 25 MHz.

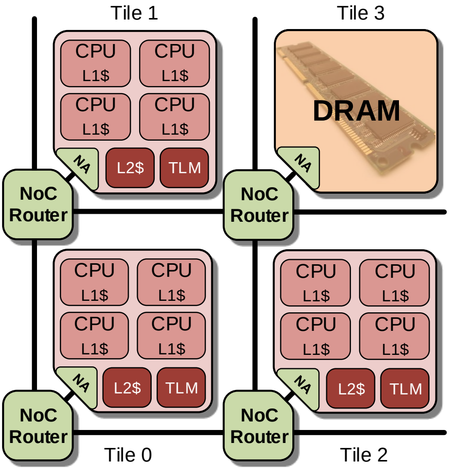


Figure 3.8: The 2 × 2 design consisting of three compute tiles and one memory tile. Depiction based on internal Invasive Computing material.

Each core has a private 16 KiB 2-way instruction cache and a private 8 KiB 2-way write-through L1 data cache. Additionally, the 4 cores of each tile share a 64 KiB 4-way write-back L2 cache. Each tile has 8 MiB of SRAM-based on-chip memory (tile-local memory, TLM). The TLM of each tile is part of the system's global physical address space. The memory tile has 256 MiB of DDR3 memory attached to its internal bus.

Each tile contains a network adapter that connects the tile to the network-on-chip (see Section 3.2). Every access to remote memory (either remote TLM or shared DRAM) is turned into a data transfer on the NoC. In this prototype, every compute tile except for tile 0 has the same distance from the memory tile (1 hop); tile 0 has a distance of 2 hops.

The hardware design was synthesized [Bec+] to a CHIPit Platinum system [Syn15] shown in Figure 3.9. The system consists of six Xilinx Virtex 5 LX 330 FPGAs. Each FPGA is connected to 8 MiB of SSRAM, which backs the TLM. Additionally, the system has a DDR extension board for the DRAM.

Figure 3.9: The Synopsys CHIPit Platinum prototyping system. Picture taken from internal project material.

Summary

- Invasive Computing aims to increase resource usage efficiency and predictability through a hardware/software codesign approach.
- Its two fundamental ideas are resource-aware programming and exclusive resource allocation.
- Invasive architectures are tiled partially non-cache-coherent shared-memory architectures with distributed memory.
- The operating system exposes these hardware properties to the programmer.
- The PGAS programming language X10 offers means to safely program invasive architectures.
- A working FPGA-based prototype of an invasive architecture combining all novel hardware and software components exists.

```
Hello Woddd
```

Output of the first distributed X10 program running on an early hardware prototype

4 Compiling X10 to Invasive Architectures

In this chapter, we investigate the compilation of X10 (cf. Section 3.4) to invasive hardware architectures (cf. Section 3.2). First, we focus on intra-tile parallelism and describe the mapping of X10's shared-memory parallelism features to hardware inside a tile. Then, we turn towards inter-tile parallelism. Here, we focus on data transfers between tiles. More specifically, we exhaustively study techniques for efficiently transferring flat as well as pointered data structures. We implement and thoroughly evaluate these techniques on a prototype of an invasive architecture. Parts of this chapter have been published in [MT17], [Moh+15], and [Bra+14].

Motivation. We saw in Section 2.4 that different programming models have been proposed for and used on non-cache-coherent architectures. The shared-memory programming model is the most familiar model to programmers, but requires either fine-grained software-based coherence management by the compiler or coarse-grained coherence management by the operating system.

Alternatively, we can partition the address space on a logical level and make each coherence domain the owner of one partition. This prevents accesses to the same memory location from different coherence domains, as each coherence domain only reads and writes addresses from its own memory partition, thereby sidestepping the issues caused by missing hardware-based cache coherence. To communicate between domains we then use explicit messages, which may make programming the system more difficult.

The PGAS model offers a compromise between both models where the programmer keeps some of the flexibility of the shared-memory programming model, namely the ability to point to arbitrary objects, while still having the obligation to explicitly handle data placement.

However, just like the message-passing model, this model requires frequent data transfers between memory partitions. If one coherence domain R requires access to data located in the partition of another domain S, we must, in general, copy this data to R's memory partition. As these operations can occur frequently, it is important to implement them efficiently.

We presented the general idea of how to implement data transfers on non-cache-coherent architectures in Section 2.4.3. However, general-purpose programs, especially if written in modern object-oriented languages, pose additional challenges. Here, programs often use pointered data structures, e.g., linked lists or trees. The standard approach to copy such a data structure is to serialize it to a byte stream. We can then easily transfer this representation to another memory partition to deserialize a copy of the original data structure. However, this serialization can cause a large overhead, especially concerning memory usage.

Contribution. In this chapter, we investigate the compilation of the PGAS language X10 to invasive hardware architectures. Concerning intra-tile parallelism, we show how we efficiently handle the creation of a large number of activities without needing a user-level scheduler. Concerning inter-tile parallelism, we elaborately discuss possibilities to implement data transfers between off-chip memory partitions. First, we focus on simple flat data structures and exhaustively discuss the state of the art in the context of invasive architectures. Then, we turn towards complex pointered data structures. Here, our main contribution is a novel data-transfer technique to accelerate the transfer of pointered data structures. Our technique is based on object cloning, which we extend with automatic compiler-controlled software-based coherence management to make it usable on non-cache-coherent architectures, such as invasive architectures. We implement a selection of the discussed data-transfer techniques in the X10 compiler and extensively evaluate the techniques on an FPGA-based prototype of an invasive architecture. Moreover, we identify opportunities for hardware support of coarse-grained software-based coherence management. We propose a matching hardware extension and evaluate the area overhead of an FPGA-based prototype implementation.

Structure. The structure of the following chapter is as follows:

- In Section 4.1, we explain how we efficiently implement X10's features related to intra-tile parallelism on invasive architectures.
- In Section 4.2, we discuss inter-tile parallelism and identify data transfers between off-chip memory partitions as an important building block.
- In Section 4.3, we first describe the state of the art for transfers of simple contiguous data structures and then provide a detailed overview of implementations on invasive hardware architectures.
- In Section 4.4, we then turn towards more complex pointered data structures, where we also present our novel technique based on object cloning.
- In Section 4.5, we show that cache operations on address ranges complement the previously proposed data-transfer techniques. We present an instruction-set extension and hardware implementation of non-blocking range-based cache operations.
- In Section 4.6, we evaluate the performance of data-transfer techniques for both flat and pointered data structures on an FPGA-based prototype of an invasive architecture using both synthetic benchmarks as well as an existing testsuite of X10 programs. We also evaluate the overhead of our hardware extension.

Acknowledgments. This chapter is based on publications that are joint work with Matthias Braun, Sebastian Buchwald, Christoph Erhardt, Daniel Lohmann, Benjamin Oechslein, Jens Schedel, Carsten Tradowsky, and Andreas Zwinkau [Bra+14; Moh+15; MT17]. We include a description of how we map X10's shared-memory features to hardware inside a tile in Section 4.1. This is not a central contribution of this dissertation and joint work with Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. We still include this material for the sake of completeness.

The complete implementation of the range-based cache operations we describe in Section 4.5 was done by Michael Mechler and Carsten Tradowsky [Mec16; MT17]. Additionally, the FPGA-based platform we use in Section 4.6 is the result of many contributions in the scope of the Invasive Computing project. In particular, we use the following components:

- the X10 compiler (cf. Section 3.5), developed by Matthias Braun, Sebastian Buchwald, Eduard Frank, Andreas Fried, Tobias Kahlert, Tobias Rapp, Martin Seidel, Andreas Zwinkau, and the author;
- the operating system OctoPOS [Oec+11] (cf. Section 3.3), developed by Gabor Drescher, Christoph Erhardt, Daniel Lohmann, Sebastian Maier, Benjamin Oechslein, Jens Schedel, and Florian Schmaus;
- the network-on-chip [Hei+14] including the network adapter [Zai+15], developed by Stephanie Friederich, Jan Heißwolf, Sven Rheindt, and Aurang Zaib;
- the C*i*C [Rav15], developed by Ravi Kumar Pujari; and
- the FPGA-based prototype implementation [FHB14], developed by Srinivas Boppu, Stephanie Friederich, and David May, and Sven Rheindt.

4.1. Intra-Tile Parallelism

X10 maps naturally to non-cache-coherent architectures, particularly to partially non-cache-coherent ones like invasive hardware architectures (cf. Section 3.2). The fundamental idea is as follows. We follow Section 2.4.4 and partition the physical address space. Each coherence domain, i.e.,

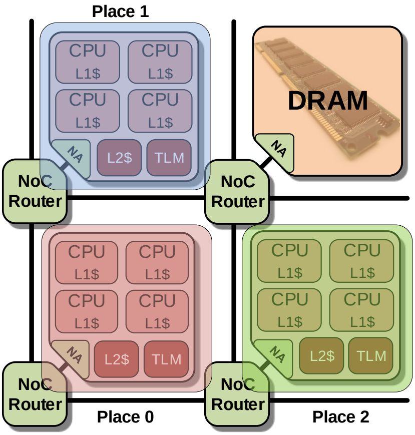


Figure 4.1: The 2 × 2 design as viewed by the X10 runtime system. By default, the tile that contains the memory controller and DRAM is not visible to the programmer. Depiction based on internal project material.

each tile on an invasive architecture, then corresponds to one place. As each tile behaves like a shared-memory system, this matches the semantics of a place. Thus, we can employ X10's shared-memory features (mainly **finish** and **async**) to exploit parallelism inside the coherence domain, i.e., tile. Figure 4.1 shows that we view each compute tile as one place. By default, we exclude the memory tile from being available as a place. Hence, every access to off-chip memory proceeds via the NoC.

In its standard runtime system, X10 employs user-level scheduling for activities. Hence, executing an **async** statement creates an activity object (represented as an actual X10 object) and hands it to the runtime system for execution. The runtime system maintains a pool of so-called worker threads, which are kernel-level threads. It then schedules activities, i.e., user-level threads, to these worker threads in a many-to-one fashion. It employs work stealing [TWL12] to balance load between worker threads.

This approach is common, but has the well-known downside that a blocking call into the operating system (e.g., for I/O) blocks the worker thread with all activities it manages. The default X10 runtime system works around this problem by starting additional worker threads before potentially blocking operations. This implicates some overhead for the creation and termination of worker threads before and after potentially blocking operations to keep the number of non-blocked worker threads close to the available hardware parallelism. Additionally, if this parallelism adaption is forgotten for a potentially blocking call, a core might idle although runnable activities exist.

On invasive architectures, we can implement **async** more efficiently. As invasive architectures offer special hardware and operating system support for fine-grained parallelism in the form of *i*-lets, we can map each activity directly to an *i*-let. Hence, no representation of an X10 activity exists on the level of the runtime system; each activity *is* an *i*-let.

This greatly simplifies the runtime system, as we do not need a user-level scheduler at all. As *i*-lets are lightweight, have run-to-completion semantics, and use cooperative scheduling, OctoPOS can efficiently create and dispatch large numbers of *i*-lets. Hence, it is no problem to create one kernel-level thread per activity. Kernel-level threads of traditional operating systems are too heavyweight for this. Viewed another way, OctoPOS puts a user-level-like scheduler (and its properties) into its kernel.

We implement **finish** using one signal primitive (cf. Section 3.3) per **finish** block. Each *i*-let remembers its corresponding **finish** as *i*-let-local data, which resides at a designated location in the *i*-let's context.

Concerning memory management, X10 requires a garbage collector. Hence, we ported a conservative garbage collector [BW88] to OctoPOS. We run a separate instance of the garbage collector per tile (i.e., one per place, which is the default in X10). The garbage collector follows the stop-the-world approach and uses a mark-and-sweep strategy. As it is conservative, the garbage collector does not require much functionality from the operating system or assistance by the compiler. Its two main requirements are (i) an interface to stop all running *i*-lets (except the current one) on all cores of the tile, and (ii) an interface to query the stack bounds of all existing (blocked and unblocked) *i*-lets on the tile.

By default, we configure the garbage collector to place the heap in the tile's partition of the DRAM. Hence, we allocate all X10 objects in the tile's DRAM partition. The TLM is reserved for use by the runtime system[9]. During its mark phase, the garbage collector scans the registers of the active core, all stacks, the TLM, and the used part of the heap for potential root pointers.

As mentioned in Section 3.4, X10 provides `GlobalRef` to enable one place to point to an object residing on a different place. We have to ensure that these objects are not deleted on their home place, even if the only references to them exist on other places. As we do not use a distributed garbage collector, we employ the same workaround as the original X10 runtime: we save the addresses of objects referenced by a `GlobalRef` in a special data structure on their home place. These objects thus never get deallocated automatically by the garbage collector.

4.2. Inter-Tile Parallelism

As explained in Section 3.4.2, X10 exposes data locality in the form of places. Initially, all data resides on place `Place.FIRST_PLACE` and must be distributed to other places in the course of a program run. Additionally, a distributed computation usually requires frequent data exchange between places.

Hence, on an invasive architecture, efficient data transfers between tiles are important for the performance of X10 programs. More specifically, we usually want to transfer data in the off-chip memory partition of one tile to the off-chip memory partition of another tile. In the following, we discuss possible implementation techniques for such data transfers. We first focus on simple bytewise copying of data, as it can serve as a building block for all other transfer types. Then, we investigate transferring more complex structured data, such as linked lists or trees.

As we have seen in Section 3.2, the family of invasive hardware architectures is diverse. To simplify our following discussion, we restrict ourselves

[9] We provide the annotation `@TLMAllocate` to enable user-controlled allocations in the TLM. However, these objects are not garbage-collected and have to be deleted manually.

to one (simplified) instance of an invasive architecture that captures all properties relevant to data transfers. We base our further discussions on this instance. Subsequently, we discuss concrete implementations of the presented data transfer approaches for the prototype hardware described in Section 3.6.

We make the following observations about invasive architectures, which lead to the model shown in Figure 4.2. Note that the upper box in Figure 4.2 depicts the actual off-chip memory and not a logical address space as in Figure 2.9. Off-chip memory and all TLMs are part of the global physical address space.

The whole system consists of the chip itself and off-chip main memory. A memory controller connects the chip to the off-chip memory. There may be multiple off-chip memories attached to multiple controllers, e.g., one controller at each border of the chip with separate memory modules. Off-chip memory is DRAM-based and large (in the order of gigabytes). We only model one off-chip memory, as the existence of multiple memories or controllers is irrelevant in our context.

The cores are grouped into tiles. Cores have private caches and all caches inside a tile are kept coherent by the hardware. A tile usually contains more than one core, constituting a partially non-cache-coherent architecture. In this case, multiple cores may share additional cache levels. The cores of a tile can also be heterogeneous. We restrict ourselves to one core with one private cache in write-back configuration per tile. This is the simplest setting that is at the same time complex enough to require software-managed coherence due to lack of hardware-based coherence across tile boundaries. However, the presented techniques also work with more complex cache hierarchies.

Additionally, there is on-chip memory available to each tile in the form of tile-local memory (TLM). TLM is SRAM-based and small (in the order of kilobytes to a few megabytes). In general, TLM offers higher bandwidth and lower latency compared to off-chip memory.

There is a global physical address space that includes off-chip memory as well as all TLMs. Hence, we can use TLMs for direct tile-to-tile communication. We assume that caches cache the complete address space, i.e., accesses to both off-chip memory and all TLMs. Hence, in general, we need to manage coherence in software for parts of the address space that we access from multiple tiles.

Tiles are connected to each other and to the memory controllers by a scalable network-on-chip. Accesses to off-chip memory and to remote TLMs use this interconnect. We restrict ourselves to two tiles and thus study data transfers from one sending tile to one receiving tile, i.e., no broadcasts or gather-like operations. Point-to-point communication is a fundamental operation on top of which we can implement all other communication patterns. We further assume that the off-chip memory has two logical partitions, one for each tile.

Throughout the following discussion, we assume a cache that offers two operations: invalidate and writeback (cf. Section 2.2.2.3). Furthermore, we assume that all operations can be executed on the respective cache line for a given address. Invalidate marks a cache line as invalid, meaning that the next time an address from the cached range is accessed, it will be fetched from memory. Writeback writes a dirty cache line back to memory. The cache line stays valid after this operation. We use flush as a shorthand for a write-back followed by an invalidation.

4.3. Block-Based Data Transfers

In this section, we study the bytewise copying of contiguous memory blocks between shared-memory partitions on invasive architectures. This operation is available to the X10 programmer in the form of the methods `Array.asyncCopy()` and `Rail.asyncCopy()`[10], which perform bytewise copying of (parts of) an array between places[11]. In the following, we refer to this operation as a shallow copy.

Hence, we want to copy a contiguous memory block B from the off-chip memory partition of the sending tile S to a copy B' of that memory block in the off-chip memory partition of the receiving tile R; see Figure 4.3 for

[10] In X10 terminology, a `Rail` is a one-dimensional zero-indexed dense array.

[11] Hence, these methods should only be used on arrays of value types, not on arrays of references. Unfortunately, the X10 type system is not powerful enough to express this restriction, so the programmer must be careful.

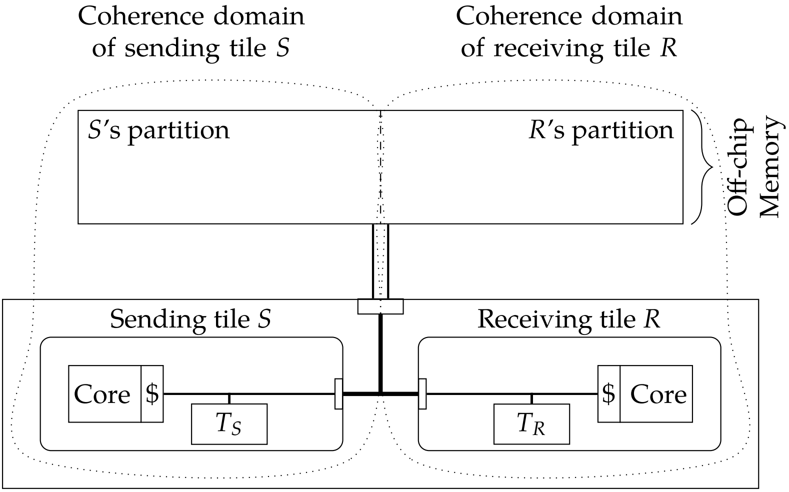


Figure 4.2: Model of an invasive architecture. The system consists of off-chip memory (upper half) and chip (lower half). We have two tiles: sender S and receiver R. Each tile owns a logical partition of the off-chip memory address space. On the chip, we model one core with a private cache (abbreviated as \$) and a TLM T per tile.

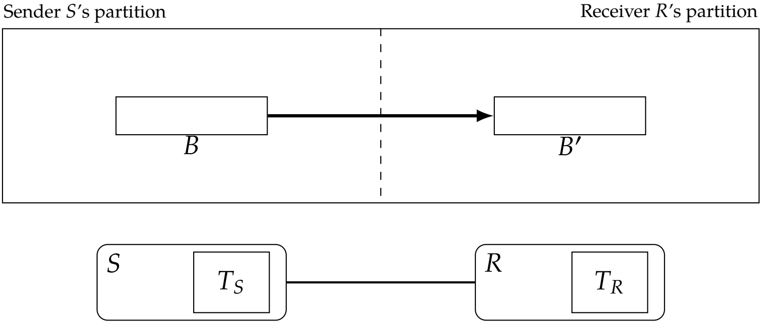


Figure 4.3: Transferring a memory block B to a copy B' from sending tile S to receiving tile R.

an illustration. In the following, if we speak of data being located "in off-chip memory" we mean that it is located in a part of the address space that is backed by the off-chip memory. As our architecture has caches, the actual data might not be completely located in off-chip memory physically, but can also be (partly) held in a cache.

We assume that our X10 programs hold their data in off-chip memory by default. Due to the characteristics of on-chip and off-chip memory, especially the TLM's severely limited size, it is unrealistic to assume that programs hold a significant amount of data in TLM. Thus, as explained before, the X10 runtime system places the heap into off-chip memory and therefore allocates all X10 objects there.

On invasive architectures, we have two different types of memory at our disposal: fast but small TLMs, as well as slower but larger off-chip memory. It is interesting to investigate the design space of data-transfer implementations regarding the characteristics of these memory types. As shallow copies form the core of message-passing libraries, such as MPI, for non-cache-coherent architectures, there is a lot of prior work on this topic. In the following, we give an overview of the state of the art. We discuss using both types of memory separately: first TLMs in Section 4.3.1, then off-chip memory in Section 4.3.2.

4.3.1. Using TLM

We can use TLMs for copying B to B'. The main reason for using TLMs is that, ideally, they enable fast on-chip communication without potentially slower accesses to off-chip DRAM. In the best case, the sending tile holds the relevant data in a local cache. From there, it is transferred via the on-chip network to the TLM of the receiving tile. The receiving tile can then read the data. This avoids (blocking) accesses to off-chip memory completely.

The design space of using on-chip memories, such as TLMs, for message passing is large [Rot11]. To simplify discussion, we start out with the description of a simple approach first and then give a brief overview of the design space.

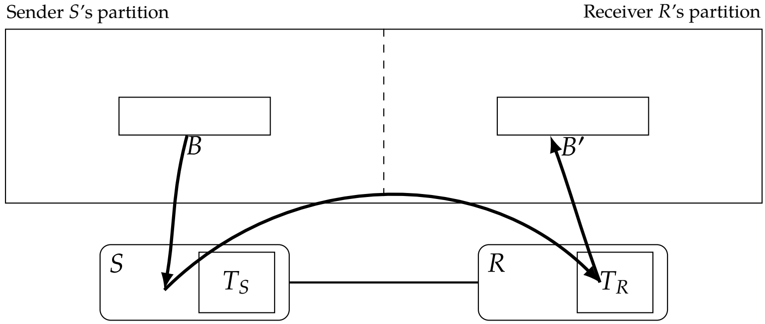


Figure 4.4: Transferring a memory block B to a copy B' via TLM from sending tile S to receiving tile R using a push-style approach.

One possibility to copy B to B' using TLMs T_S and T_R of sender S and receiver R, respectively, proceeds as follows (cf. Figure 4.4):

1. The sender S copies from B to T_R. As writes to remote TLMs are cached in S's private cache, we must then force a writeback of the relevant cache lines after writing. It is trivial to determine the relevant cache lines, as B is contiguous in memory and we know its starting address and size. S waits until all relevant cache lines have been written back to T_R.

2. S notifies R that T_R now contains a copy of B.

3. R copies from T_R to B'. As read operations are cached in R's local cache, we must invalidate the relevant address range of T_R before reading. Then, the receiver R has a coherent view of the data written to T_R.

This initial description anticipates a number of design decisions that we now investigate in a structured manner. More specifically, we explore the following five aspects:

(i) the placement of data and the responsibility of transferring data;
(ii) the actual implementation of data transfers;
(iii) the allocation strategy;

(iv) the synchronization mechanism to wait until write-back operations have finished; and
(v) the notification mechanism to inform a remote tile of incoming data.

In the following, we briefly discuss each of these aspects. We partially base our presentation on [Rot11, section IV] and refer to the same source for a detailed discussion of the message-passing design space in the context of the Intel SCC.

Placement and responsibility. Does the sender push messages to the receiver's TLM or does the receiver pull the message from the sender's TLM? In our initial description we assumed the former, however, we could also copy from B to T_S and then let R copy from T_S to B'. While the situations seem symmetric, architectural peculiarities or communication patterns other than point-to-point communication can make one approach superior. For example, to broadcast information it may be more efficient to place the message in the sender's TLM and let all receivers pull from it.

Transfers. How do we copy contiguous memory blocks? In the simplest case, we use a core of either the sending or the receiving tile and execute regular load/store instructions in a loop. As our invasive hardware provides DMA units, we can also implement the transfer operation without CPU interaction. Both approaches may involve software-managed coherence.

Allocation. How do we allocate the TLM, i.e., how do sender and receiver agree on which area of the TLM to use for a particular communication operation? One possibility is dynamic allocation, i.e., if the sender wants to put a message into the receiver's TLM, the sender explicitly requests a memory area of the needed size before sending the message. This does not waste any space, because we allocate exactly what is needed when it is needed. However, it requires an additional forth-and-back communication between sender and receiver, which may be expensive relative to the transfer of the actual message content, e.g., for very small messages.

Alternatively, we can use a static allocation scheme. For example, on a system with n tiles, we could partition each TLM into $n-1$ partitions and, for each memory, exclusively assign one partition to each *other* tile in the system. Hence, with a push-style data placement, the sending tile could store the message to this sender's exclusive part of the receiver's TLM. Thereby, we avoid all run-time allocation overhead, but significantly decrease the amount of data that can be sent with a single message. For example, Ureña et al. [URK] report a maximum message size of 160 bytes on the Intel SCC when using 48 cores with a static allocation scheme for the 16 KiB on-chip memory per tile.

Synchronization. How do we wait until writing back cache lines has finished? Here, the hardware must provide appropriate support. If the hardware guarantees that write operations are processed in program order, implementation is straightforward. This is the solution chosen by the Intel SCC [Mat+10, section III; Rot11, section II.B]. Here, the software only has to ensure writing complete cache lines. Then, the hardware guarantees that pending write requests complete before the next write request starts. If the hardware only gives weaker guarantees, it must provide support for awaiting the destination's response that the write operation has completed.

Notification. How do we inform our communication partner that data has arrived? In the simplest case, we manage, in each TLM, an array of $n-1$ boolean flags; one for each other tile in the system (assuming n tiles). If tile i wants to signal tile j, it sets the i-th flag in tile j's array. With software-managed coherence and appropriate synchronization means (see above), this allows us to realize the required notification scheme. However, the resulting notification mechanism requires polling, which is, in general, inefficient. As such notification between tiles is a potentially frequent operation, invasive hardware provides specialized support via the remote spawning of i-lets described in Section 3.3.

As mentioned before, using TLMs provides a number of advantages: they are decentralized and they provide higher bandwidth as well as lower latency compared to off-chip memory. Hence, we see that TLM allows us to implement efficient message-passing functionality.

However, they also require copying data between address space backed by off-chip memory and TLMs. This becomes important if we consider larger messages. At some point, the message size exceeds the size of TLM[12]. Then, the sender must split the message into chunks and the receiver must reassemble these chunks. In the simplest case, the sender writes a chunk, waits until the receiver has acknowledged copying the chunk to its partition of the off-chip memory, and then writes the next chunk. This process is repeated until the complete message has been transmitted. Splitting and reassembling messages causes significant overhead.

To partially hide this overhead, we can use pipelined communication [Cla+11, section 3.2]. Here, we use multiple disjoint areas of the TLM. The sender writes a first chunk and notifies the receiver. Then, the receiver copies the first message chunk from its local TLM to off-chip memory while, at the same time, the sender writes the next chunk to a different area of the receiver's TLM.

However, regardless of how we design our message-passing scheme, we cannot avoid the problem that at some message size, the overhead for splitting and reassembling messages is higher than the bandwidth and latency advantages gained by using TLMs. At this point, it becomes worthwhile to exploit off-chip memory to transfer messages.

4.3.2. Using Off-Chip Memory

We can use off-chip memory to copy B to B' as follows (see Figure 4.5):

1. *copy & writeback:* S copies B to B' and then performs an explicit writeback of the address range of B'. Then, S waits until all relevant cache lines have been written back to off-chip memory.

2. *notify:* S notifies R via a message that it is now safe to read from B'.

3. *invalidate:* R invalidates the address range B' in its local cache to guarantee reading up-to-date values from memory. It can then work with B'.

[12] Or the part of TLM that is used for the communication operation in case of a static allocation scheme.

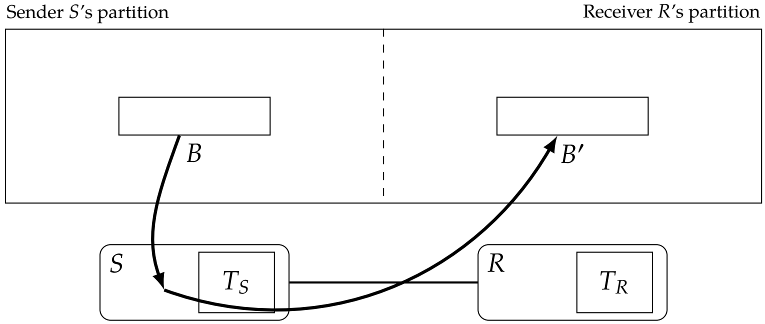


Figure 4.5: Transferring a contiguous buffer B to a copy B' via off-chip memory.

Here, we use TLM solely for notification. The actual data transfer happens purely using off-chip memory. The approach is a straightforward extension of the techniques from Section 2.2.2.3 from a single memory location to a memory block, i.e., an address range.

The design space is similar to that of using TLM. Regarding placement, we again have a symmetric situation: in our proposed push-style approach, we copy B to B' using tile S, however, we could also use tile R to perform the copy operation. In this pull-style case, S performs a writeback of B (so that off-chip memory contains up-to-date values), and notifies R of B and B''s size. R then invalidates B and copies B to B'.

For large messages, using off-chip memory can offer a performance advantage over using TLMs. This becomes clear if we look at messages significantly larger than the cache size. In this case, most of B already resides in the off-chip memory and not in caches in S. Hence, it is relatively cheap to write back the remaining parts from caches in S to off-chip memory. In contrast, chunk-wise copying of the complete buffer B from off-chip memory to TLM may be significantly more expensive.

4.3.3. Related Work

There has been a considerable amount of work on the efficient implementation of message passing on non-cache-coherent architectures.

Rotta [Rot11] investigates efficient message-passing implementations using on-chip memory on the Intel SCC. The author studies the design space of message-passing protocols, identifies six design dimensions, and classifies existing approaches according to this framework.

Chapman et al. [CHH11] port X10 (cf. Section 3.4) to the Intel SCC. They only use on-chip memory for message passing. They consider this a bottleneck and report their plans to also use off-chip memory for message passing.

Ureña et al. [URK] present an MPI implementation for the Intel SCC. They observe that using off-chip memory is faster than on-chip memory for passing messages of size 5.6 KiB and higher. Thus, their MPI implementation chooses the communication channel (on-chip or off-chip) depending on the message size. In their experiments, off-chip memory was marked uncacheable, i.e., accesses to off-chip memory were executed on a granularity of individual loads and stores (and not whole cache lines). This avoids the need for software-managed coherence, but significantly decreases performance [Cla+11]. In spite of this shortcoming, communicating via off-chip memory provided a performance advantage for sufficiently large messages. With caching of off-chip memory enabled, the observed break-even point is likely to be at significantly lower message sizes.

Clauss et al. [Cla+11] study the shared-memory and the message-passing programming models on the Intel SCC. They enable caching of the off-chip memory. Unfortunately, the Intel SCC provides no hardware means for fine-grained cache control. In particular, it does not allow forcing writebacks or invalidations of the L2 cache (neither on individual lines nor on the whole cache). Therefore, Clauss et al. use a workaround that reads from a sufficiently large contiguous memory area to evict the complete current L2-cache contents. Hence, this causes dirty lines to be written back to memory, i.e., flushes the complete L2 cache. However, while this workaround allowed Clauss et al. to confirm the functional correctness of their approach, the workaround's significant overhead renders their scalability measurements unrepresentative.

Van Tol et al. [Tol+11] investigate memory copy operations on the Intel SCC. They enable caching of the off-chip memory. Van Tol et al. propose "copy cores", which are dedicated cores that asynchronously copy memory regions and are used by other cores as a service. Their implementation is also hindered by the missing means for invalidating and writing back L2-cache contents. In principle, this idea is also applicable to invasive architectures. However, the DMA unit present in every tile is preferable.

Reble et al. [RCL13] and Christgau et al. [CS16] present implementations of MPI-based one-sided communication [Mes15, chapter 11] for the Intel SCC. With one-sided communication, one party is passive, i.e., the other party specifies all communication parameters. This contrasts two-sided communication, where each send operation requires a matching receive operation by a cooperating party. Reble et al. exploit both on-chip and off-chip memory for one-sided communication. However, they turn off caching of off-chip memory and do not manage coherence in software. Despite the significant performance loss due to disabled caching, using off-chip memory is faster than using on-chip memory for sufficiently large messages.

Christgau et al. improve upon this previous work by enabling caching of off-chip memory. However, they are also hindered by the missing L2-cache functionality. They work around this problem by using a special caching policy available on the Intel SCC for shared off-chip memory (we discuss this topic in more detail in Section 4.5). Here, stores to these memory areas are not cached. Read operations to this memory area are only cached in the L1 cache, for which the Intel SCC offers a dedicated invalidation instruction, but bypass the L2 cache. Despite being forced to use this workaround, they report a 5× reduction of communication costs for large messages when compared to the default message-based implementation. As we have seen in Section 2.4.4, one-sided communication is important for PGAS languages. This supports our idea that precise cache control is crucial on non-cache-coherent architectures, which we investigate in more detail in Section 4.5.

We developed our approach to transfer data via off-chip memory independently of Christgau et al. Christgau et al.'s main contributions are

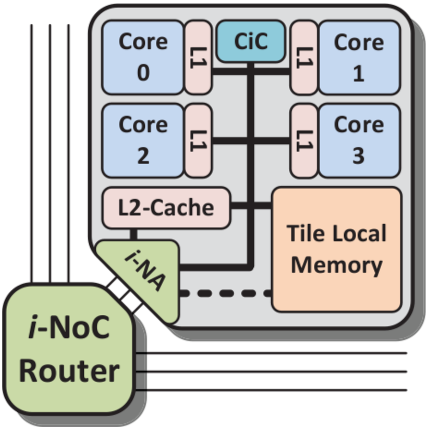


Figure 4.6: A detailed view of the structure inside a tile. Depiction taken from [Hei14].

the sophisticated exploitation of the Intel SCC's architectural features, the integration of their technique with MPI semantics, and the idea of remote invalidations (which we discuss in Appendix A.1). Concerning the aspect of software-managed coherence, our technique and theirs are equivalent.

4.3.4. Implementation on the Hardware Prototype

The FPGA-based prototype of invasive hardware (cf. Section 3.6) is a bit more complicated than the model we used in the previous sections to study data-transfer techniques. Figure 4.6 shows a detailed view of the structure inside a tile. We have private L1 caches per core and a shared L2 cache per tile. The L1 cache is configured in write-through mode and the L2 cache is configured in write-back mode. The caches cache all reads and writes to both remote TLMs and the off-chip memory. However, only the L1 cache caches the local TLM; the TLM is not cached by the L2 cache (compare the positioning of the L2 cache in Figure 4.6). The L2 cache's main purpose is to reduce utilization of the on-chip network due to possibly frequent L1-cache misses when accessing remote data (remote TLM or off-chip).

In the following, we present possible implementations of the necessary functionality we identified in Sections 4.3.1 and 4.3.2: (i) transfers, (ii) synchronization, and (iii) notification.

Transfers. To transfer data between TLMs of a sending tile S and a receiving tile R, we have three alternatives. First, we can use *i*-lets. An *i*-let can carry two 32-bit words. Hence, for very small transfers, we can directly encode data into an *i*-let, spawn it on R and let it write the data to the target location [Moh+15]. As we also use *i*-lets as our notification mechanism (see below), it is almost always preferable to integrate a data transfer if possible.

Second, we can write to the remote TLM using regular stores[13]. These stores are cached by the sender's local L2 cache. Hence, we now have to manage coherence in software to guarantee that written data is actually visible to the receiver. We use platform-specific operations [Cob16, section 74.3.3] to force writebacks of individual L2 cache lines. The provided interface allows to supply an address whose corresponding cache line is then looked up and written back. This allows to implement the writeback of an address range $[S, E]$ with a cache line size of L using a loop as follows:

```
for x := $S - (S \bmod L)$ to $E - (E \bmod L)$ step $L$:
  writeback(x)
```

We use mod to denote the modulus operation on integers. The term $A - (A \bmod L)$ rounds the address A down to the nearest multiple of L[14]. Hence, the loop issues one writeback operation per relevant cache line. Writing back one cache line takes 6 clock cycles [Cob16, section 74.3.3] plus the latency for handing the cache line to the network-on-chip.

We do not have to manage coherence in software on the receiving side. On the receiving tile, its network adapter receives the stores issued by

[13] The "swcpy" variant of OctoPOS uses this mechanism.

[14] If L is a power of two, we can express the term more efficiently using bitwise operations, e.g., `A & ~(L - 1)` in C notation.

the sending tile and translates them to stores inside the receiving tile. Hence, these stores trigger the tile-local hardware coherence mechanism and invalidate potential copies of the written TLM parts in the L1 caches of R's cores.

The third and preferred method for transferring data between TLMs are push-DMA transfers (cf. Section 3.3). The network adapter of each tile includes a DMA unit capable of copying contiguous memory blocks from the tile's local TLM to a remote memory without CPU interaction [Hei14, section 3.2.2.1]. Unfortunately, in the current hardware prototype, the remote memory can only be another TLM. Currently, the DMA unit cannot copy from TLM to off-chip memory.

Figure 4.7 shows the full control flow of a DMA-based data transfer. As the first step, we copy from B to B_S in the sender's TLM. We use a dynamic allocation strategy for our TLMs. Hence, before we can initiate a DMA transfer, we allocate the destination buffer in R's TLM. We spawn a remote *i*-let on R, allocate the buffer B_R in R's TLM, and pass the address back to S via another remote *i*-let spawning.

After setting up our target buffer, S initiates a push-DMA transfer from B_S to B_R. We specify two *i*-lets to execute once the data transfer has finished:

1. the first *i*-let runs on S and frees the source buffer B_S; and
2. the second *i*-let runs on R and copies B_R to B' in off-chip memory, to free up the TLM space of B_R subsequently.

In case S's TLM cannot hold a copy of B, we have to split B into as many chunks as needed and transmit each chunk using one DMA transfer. Then, only the last DMA transfer triggers *i*-lets to free the source buffer on the sender and work with B' on the receiver.

Copying data between TLMs using the hardware-based DMA transfer does not require software-managed coherence. As the L2 cache does not cache accesses to the local TLM and the L1 cache is write-through, the sender's network adapter can directly read up-to-date values from the local TLM. Then, the network-on-chip passes the data to the target tile's network adapter. On the target tile, the network adapter writes the transmitted memory block to the target tile's TLM. This write operation again triggers the tile-local hardware coherence mechanism, which invalidates potential

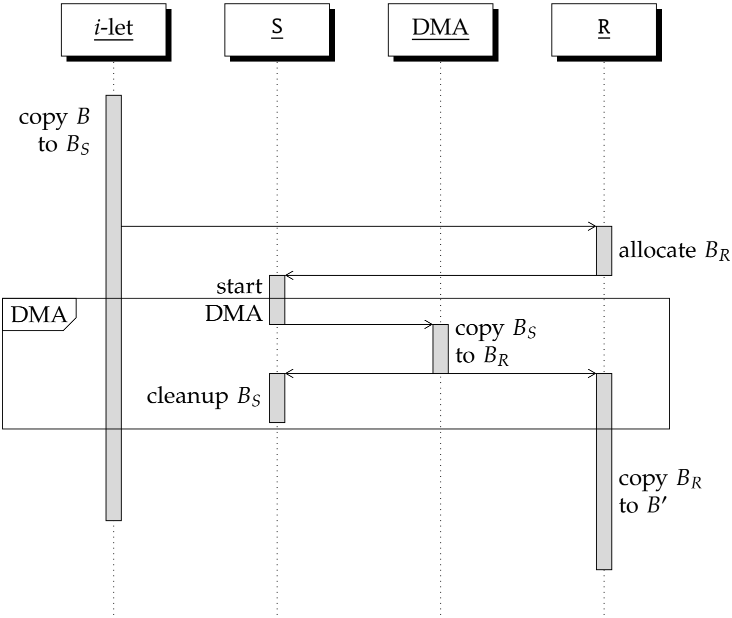


Figure 4.7: Sequence diagram for transferring data via TLM on invasive architectures. We use a dynamic allocation strategy. Depiction based on [Moh+15].

copies of the written TLM part in the receiver's cores' L1 caches. Thus, after the write has finished, all cores of the target tile have a coherent view of the copied data in the target TLM.

To transfer data via off-chip memory, we need to manage coherence in software on both sending and receiving side. The reason for this is that now the memory is "remote" from the view of both parties. Hence, in contrast to the on-chip case, the writes that affect the target memory are not visible to the receiver's caches, as the involved network adapter is located on the memory tile. Thus, no coherence actions are triggered, e.g., no L1-cache invalidations happen on the sending or the receiving tile.

Therefore, after writing a memory block located in off-chip memory, we must write back the respective address range. Before reading this memory block on the receiving side, we must invalidate its address range. We must do this on both levels of the cache hierarchy.

Hence, on the sending tile, we must write back the address range first in the L1 caches and then in the L2 cache. As the L1 caches are configured in write-through mode, the write-back can be omitted. We write back the relevant L2 cache part using the method described above.

On the receiving tile, we need to invalidate the relevant parts of both L1 cache and L2 cache. For the L2 cache, we use the same software-based loop construct as for write-backs, but with an `invalidate` operation. The provided interface is the same as for writing back cache lines [Cob16, section 74.3.3]: we supply an address whose corresponding cache line is then looked up and invalidated. Invalidating one cache line takes 5 clock cycles [Cob16, section 74.3.3].

Unfortunately, the L1 data cache offers no such fine-grained control: it can only be flushed completely [Cob16, section 77.10.7] (we later improve this situation in Section 4.5). However, as the cache is configured in write-through mode, no modified data is written back. Therefore, the resulting overhead consists of a higher than necessary number of subsequent L1 cache misses, which are mostly compensated by the L2 cache.

Synchronization. In order to wait until writing back cache lines has finished, we use the following implementation. The L2 cache blocks all accesses (i.e., further loads or stores) until prior write-back operations have finished [Cob16, section 74.3.3]. However, the invasive hardware prototype, in contrast to the Intel SCC as described in Section 4.3.1, does not guarantee that pending write requests to remote memory complete before the next write request can start. Thus, as soon as the network adapter has handed the cache line of a write-back operation to the network-on-chip, the next write-back operation can proceed. There is no acknowledgment, i.e., the hardware does not wait for this cache line to be actually written back to remote memory.

However, the invasive NoC guarantees that loads do not overtake preceding stores to the same destination tile. Hence, after issuing all our write-back operations, we load from a reserved address W in the remote memory. This load from W only completes once the previous write-backs have finished. It is crucial that we really load from remote memory and not from a local cache. To prevent that the load from W is served by a cache, we take the following precautions:

- For each core c in the system, we use a different address W_c. All addresses W_c are aligned so that they reside in different cache lines. Hence, there can be no interference between the synchronization operations of multiple cores.
- We invalidate the L2 cache line of W_c before reading.
- To load from W_c, we use a cache-bypassing load instruction [Cob16, section 77.10.2] that bypasses the L1 cache. On our platform, this does not bypass the L2 cache, hence the explicit invalidation of the L2 cache line is required.

Notification. To implement notification, we have two alternatives. First, we can use the mechanism described in Section 4.3.1, i.e., stores to dedicated notification flags located in TLMs in conjunction with software-managed coherence. OctoPOS uses this scheme during early bootup before all hardware has been properly initialized.

However, as notification is a frequent operation, invasive architectures provide dedicated hardware support. The preferred method for notification between tiles works via *i*-lets. Hence, to notify the receiver of a message, we spawn an *i*-let on the receiving tile. There, the *i*-let can locally trigger the appropriate action, e.g., using signaling primitives (cf. Section 3.3).

Before spawning the *i*-let, we have to ensure the completion of potential write-back operations using synchronization as described above.

In the case of a DMA transfer, the hardware provides support for starting *i*-lets on the sending and the receiving sides as soon as the data has been transferred completely (cf. Section 3.3). This allows us to merge synchronization, notification, and the actual data transfer into one operation that is fully hardware-accelerated.

Implementation of `asyncCopy()`. In the following, we use the presented building blocks for transfers, synchronization, and notification to describe two concrete implementations of X10's `Rail.asyncCopy()` method: Ac-Tlm, which uses TLM, and Ac-Off, which uses off-chip memory to transfer data. The method `Rail.asyncCopy()` actually exists in two variants: one to push data, i.e., to copy from a local array to a remote array, and one to pull data, i.e., to copy from a remote array to a local array. We focus on the push-style transfer; the pull-style variant works analogously. We assume a call like `Rail.asyncCopy(`B`, `B'`)`.

We implement Ac-Tlm as follows. We use a dynamic allocation strategy for the TLM. Then, we copy from B to B_S in S's TLM and use a push-DMA transfer to copy B_S to B_R. The receiver then copies B_R to B'. The hardware handles synchronization and notification; as described before, no software-managed coherence is necessary.

We implement Ac-Off as follows. We use a core of the sending tile to copy B directly to B'. Then, we trigger a write-back of all L2 cache lines relevant for B'. We synchronize and then spawn an *i*-let on the tile R that invalidates the complete L1 cache and the relevant L2 cache lines for B'.

Here, we have to be careful as multiple L1 caches exist on R. In general, to guarantee coherence, we must invalidate the address range of B' in all of them. Either we ensure that the *i*-let that issues the invalidation is the only *i*-let that ever accesses B' on R. Due to our restricted scheduling policy, *i*-lets never change their core unless they block, in which case they may be scheduled on a different core in the same claim. Hence, on invasive architectures, we have some control over whether an *i*-let gets rescheduled. Or, if other cores may also access B', we must issue invalidations on all cores of tile R. We discuss this topic again in Appendix A.1.

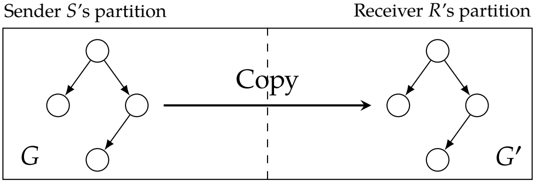


Figure 4.8: Copying an object graph to another memory partition.

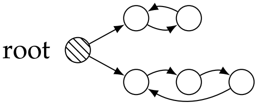


Figure 4.9: An object graph containing cycles.

4.4. Transferring Pointered Data Structures

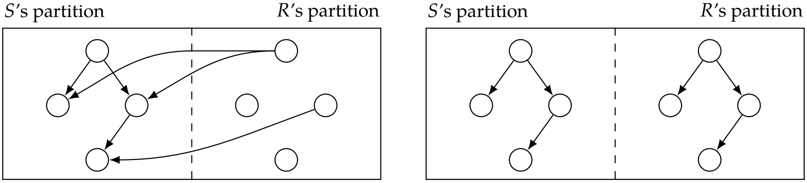
We now turn to the problem of implementing efficient data transfers of more complex structured data. We first define our task more precisely, before we discuss possible techniques using our simplified model of an invasive architecture. We then discuss concrete implementations for the invasive hardware prototype.

Figure 4.8 shows our starting point. For clarity, we zoom in on the upper part of Figure 4.2. We want to copy a data structure G from the off-chip memory partition of tile S to a copy G' of that data structure in the off-chip memory partition of the receiver R. Again, when we speak of data being located "in off-chip memory", we mean that it is located in a part of the address space that is backed by the off-chip memory. As our architecture has caches, the actual data might not be completely located in off-chip memory physically, but can also be (partly) saved in a cache.

An *object graph* is a rooted directed graph[15] [16] where the vertices are objects and an edge (x, y) means that x points to y. In the context of our

[15] We implicitly consider this graph to be connected.

[16] Strictly speaking, we have to model object graphs as multigraphs, because objects can contain multiple pointers to the same target object. However, for simplicity, we ignore this detail and use regular graphs.



(a) Shallow copy. **(b)** Deep copy.

Figure 4.10: Comparison of shallow and deep copy of an object graph. With a shallow copy, pointers contained in the object copies of the right memory partition still point to the original objects in the left memory partition. With a deep copy, this problem is avoided.

discussion, we restrict ourselves to object graphs with a single root vertex, i.e., we speak of "the" root. Object graphs can contain cycles, e.g., the graph of a cyclic linked list. Figure 4.9 shows an example of such an object graph.

We call a data structure *flat* iff its respective object graph has a single vertex and no edges, and *pointered* otherwise. Note that distinguishing between contiguous and non-contiguous data structures is not equivalent. A contiguous data structure may contain pointers to itself, hence its object graph may consist of a single vertex with one or multiple loops. While this case is somewhat contrived, in the following we make an effort to be precise and speak of pointered and flat data structures when referring to the presence (or absence) of pointers in the objects. We use contiguous and non-contiguous only to refer to the memory layout of objects.

It is important to understand what it means to make a copy of a pointered data structure or, equivalently, its object graph (cf. Figure 4.10). Making a copy of an object graph in a different memory partition requires creating a *deep copy*. Hence, we must copy all objects and at the same time modify the contained pointers so that they point to the newly created objects. A shallow copy, obtained by bytewise copying of the objects, is not sufficient as the contained references would point to the original objects. In the context of a non-cache-coherent architecture with a logically partitioned memory, these original objects reside in a different memory partition. Hence, accessing them is inherently unsafe in the sense that we have no guarantee of reading up-to-date values due to the missing coherence. This difference between shallow and deep copy only matters for pointered data structures. For flat objects, both types of copy coincide.

Additionally, we require *referential integrity* [Ora16]. Hence, if two objects in the original object graph point to the same object, the copies of these two objects must point to the same (copied) object as well. This must also hold if the object graph contains cycles. With referential integrity, operations behave the same whether executed on the original object graph or its copy[17].

We now explain why object graphs and their copies are important for X10. X10's primary language means for distributed-memory parallelism are the concept of places and the **at** construct for changing the place of execution (cf. Section 3.4). In the following, we discuss the semantics of X10's **at** operation in more detail.

X10 objects stay on their place of creation during their whole lifetime, i.e., objects cannot migrate between places. Additionally, all data accesses must be place-local in X10. This means that we can only access remote data by migrating our computation to the place where the data is located.

The **at** construct allows us to do exactly that; for example:

```
val x = ...;
at (B) {
  val y = ...;
  compute(x, y);
}
```

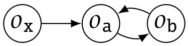
Here, we change to place `B` and execute the method `compute` on that place. We assume the expression is well-typed, i.e., `B` has type `Place` (or a subtype). Suppose we execute this **at** expression on place `A`. As explained in Section 3.4, the **at** operation is a synchronous construct[18]. Hence, conceptually, the activity executing the **at** operation changes its current place of execution from `A` to `B`. After it has finished computation on `B`, it shifts back to `A`.

[17] Referential integrity concerns object identity. If object addresses can be queried, different behaviors are possible. X10, just like Java, does not allow querying addresses.

[18] We can combine it with **async** to get an asynchronous variant.

```
class Foo {
  var p: Foo;
  def this(q: Foo) { p = q; }
  static def bar() {
    val a = new Foo(null);
    val b = new Foo(a);
    a.p = b;
    val x = new Foo(a);
    at (B) {
      val y = new Foo(null);
      compute(x, y);
    }
  }
}
```

(a) X10 code.



(b) Object graph.

Figure 4.11: An X10 program containing an **at** expression that captures variables, and the matching object graph.

The body `S` of a statement **at** `(B) S` is allowed to refer to variables that are not defined in `S` itself. In our example, the call to `compute` refers to `x`, which is defined in an enclosing scope. However, as all data access must be place-local, we cannot access `x` on `B`.

Therefore, X10 semantics dictate that the values of all free variables in `S` must be copied from `A` to `B` before `S` is executed on `B` [Sar+16, §13.3.1]. Hence, X10's **at** statements close over the values of the free variables in `S`. To implement this semantics, the X10 compiler first determines the set of free variables F in `S`. Extending our initial example to the example from Figure 4.11a, the X10 compiler determines the set of free variables $F = \{\mathtt{x}\}$.

Then, at run-time, at the program point where the **at** expression is to be executed, all variables in F are evaluated, resulting in a set V of values. The X10 runtime system now determines the set V^* of all values that are transitively reachable from values in V. Using V^* as the vertex set and the (immediate) reachability relation as the edge set, this results in an

```
val x = O();
val y = O();
at (B) use(x, y);
```

(a) Before transformation.

```
class C {
 val x, y;
 def operator()() {
  use(x, y);
 }
}
val x = O();
val y = O();
Runtime.runAt(B, new C(x, y));
```

(b) After transformation.

Figure 4.12: An example of how the X10 compiler transforms **at** statements.

object graph. In our example from Figure 4.11a, $V^* = \{o_{\texttt{x}}, o_{\texttt{a}}, o_{\texttt{b}}\}$, where $o_{\texttt{z}}$ denotes the value (or object) pointed to by z. This leads to the object graph shown in Figure 4.11b.

In general, an **at** statement may lead to an object graph with multiple roots (one per value of a free variable). However, the X10 compiler implements **at** blocks using closure objects. Hence, for each **at** block T, it creates a new class that contains a field for each free variable of T. At the program point of the **at** statement, the compiler then creates an instance of this new class, initializing its fields to the values of the variables at that point. See Figure 4.12 for an example of the transformation.

Essentially, for a particular **at** block T, this adds a new root object (the instance of C in the example) that points to all old root objects. In the following, we therefore assume that the arguments that must be copied for an **at** block always correspond to exactly one object graph with one root.

X10 also offers **at** in expression form to copy results back to the original place. Hence, it is possible to write:

```
val result = at (B) compute(x);
```

In this case, the result computed by `compute` on place `B` is transferred back to place `A` with the same deep-copy semantics as the arguments we studied before. Hence, this result object corresponds to exactly one object graph. In general, **`at`** expressions are allowed to refer to variables from enclosing scopes as well. Therefore, executing an **`at`** expression can involve the transfer of two object graphs: one from `A` to `B` for the arguments and one back from `B` to `A` for the result.

Hence, we see that in the context of X10, we must be able to transfer object graphs with deep-copy semantics between memory partitions.

Related Work. The problem of transferring object graphs with deep-copy semantics also arises in the context of other systems that support object-oriented programming in a distributed setting. Java RMI [Ora16] (for "Remote Method Invocation") is an official Java API that introduces the concept of remote objects. Remote objects appear like regular objects to the programmer, but may live in remote address spaces, i.e., another JVM, possibly running on another host machine.

Calling a method on a remote object triggers a remote method invocation, i.e., the method call is forwarded to a remote JVM that actually hosts the object. For all objects passed as arguments to such a remote method invocation, Java RMI uses the same deep-copy semantics as X10. Hence, all object graphs rooted at the objects passed as arguments are shipped to the remote host as part of the remote method invocation. The same deep-copy semantics is used for a potential method return value.

To this purpose, Java RMI employs serialization (cf. Section 4.4.1). Multiple articles [Phi11; VP03] identified this serialization step as one of the main performance bottlenecks of remote method invocations and proposed various optimizations to accelerate the process.

Task. In summary, we investigate the following scenario:

- There is an object graph G in the off-chip-memory partition of the sending tile S.
- Size and shape of G are not known a-priori.
- We require a deep copy G' of G in the off-chip-memory partition of the receiving tile R.

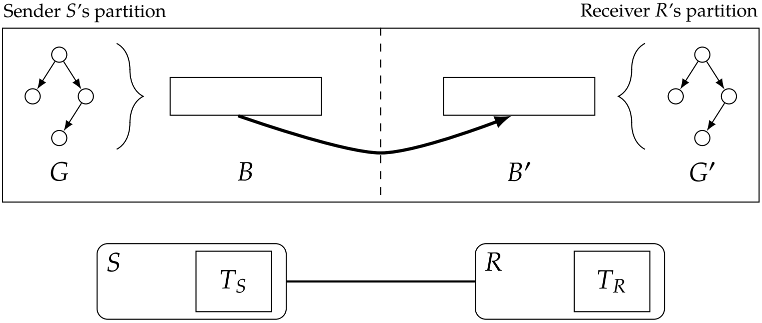


Figure 4.13: Using serialization to make a deep copy of an object graph G. Temporary buffers are denoted by B, B'; and G' is the resulting copy of G.

We assume no knowledge about the size and shape of the object graph G at the program point where the copy is required. We denote with the size of an object graph G the sum of all object sizes in G. It is unrealistic to assume that size and shape of this dynamic data structure are known to compiler or runtime system at the program point where data must be transferred. For example, suppose a program builds a tree data structure. This tree can depend on input data. Hence, size and shape can only be determined by traversing the data structure at run-time.

In the following, we discuss multiple approaches for creating this deep copy G'. We first focus on serialization-based approaches and then present cloning-based approaches.

4.4.1. Serialization-Based Approaches

One possibility to deep copy an object graph is using serialization. This approach proceeds according to the following three steps:

1. Serialize the object graph G to a flat representation B.
2. Copy B to B' in the receiver's partition.
3. Deserialize from B' a deep copy G' of G.

Figure 4.13 illustrates the process using a simplified depiction of Figure 4.2. Note that the buffers B and B' must, in general, also be placed in off-chip memory, as the size of G, and therefore the space requirement of B and B', is not known a-priori.

This approach uses serialization to reduce the problem of copying a pointered data structure to the problem of copying a single flat data structure. Serializing an object means converting its state into a contiguous byte stream in a reversible way so that we can reconstruct (deserialize) the object from the byte stream. To serialize an object graph we must serialize all objects in the graph. Algorithm 1 shows pseudo code for serialization. The algorithm is basically a depth-first search with cycle detection. We assume that `O` is the root of the object graph and that `B` refers to a buffer that holds the serialized representation. Furthermore, for ease of presentation, we assume that fields with compound types (e.g., **`struct`**s in X10) are "flattened" into their containing object. Hence, each field is either a pointer to another object or is a non-pointer type that cannot contain further pointers.

Algorithm 1 Object serialization.

```
procedure Serialize(O, B)
   if AlreadySerialized.contains(O) then
      // Get position of serialized version of O in B
      pos ← AlreadySerialized[O]
      B.append(reference to pos)
      return
   AlreadySerialized[O] ← current position in B
   for each F in O.type.fields do
      if ¬F.type.pointer then B.append(O.F)
      else Serialize(O.F, B)
```

After serializing the object graph, we have to copy the resulting byte stream B to B' in another memory partition. We discussed the general idea behind shallow copying of data between shared-memory partitions in Section 2.4.3. We saw that we can implement this operation using TLMs or directly via off-chip memory. We refer to these approaches as SER-TLM and SER-OFF.

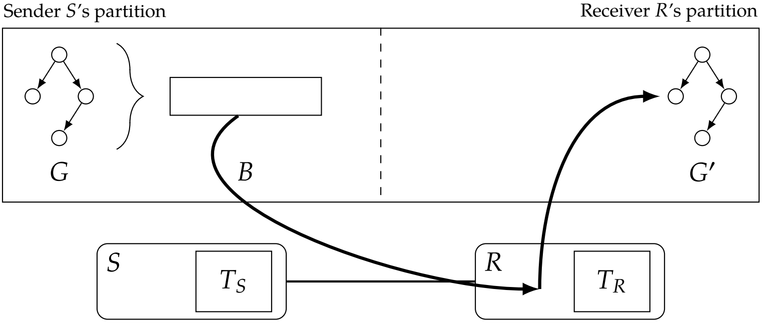


Figure 4.14: Optimized variant of transferring an object graph G using off-chip memory. B is a temporary buffer and G' is the resulting copy of G.

In our special setting of transferring pointered data structures, we can make one additional optimization. We can eliminate one of the buffers B and B'—after all, one serialized copy of G is enough on a machine with a shared physical address space. We only need two copies if we treat our system as a pure message-passing platform. However, in our setting we can exploit the shared physical address space and our knowledge that B and B' are only used temporarily.

Hence, our optimized variant for passing messages via off-chip memory (Ser-Off-Opt) follows these three steps (see Figure 4.14):

1. *write & writeback*: S serializes G into a buffer B located in its memory partition. Then, S forces a writeback for the cache lines of B from its local cache. The writeback guarantees that R can read up-to-date values for B from memory. S waits until all relevant cache lines have been committed to memory.

2. *notify*: S sends a message carrying the starting address and size of B to R. This informs R that it is now safe to read B.

3. *invalidate & read*: R invalidates the cache lines relevant for B. The cache invalidation is necessary to ensure that B is actually read from memory. Then, R deserializes from B a copy G' of the object graph.

Thus, SER-OFF-OPT avoids creating a copy of B on the receiving side. Hence, it reduces the memory requirement by 25% compared to our previous approach using off-chip memory. It also avoids copying B to B'.

The main advantage of using serialization is that we only have to transfer a single flat data structure. As message passing is often a preferred approach for programming non-cache-coherent architectures [Kum+11], architectures often provide optimized libraries for this purpose [URK; Mat+10].

However, serialization has a number of drawbacks in our scenario of transferring pointered data structures. In total, employing this approach requires up to four times as much memory as the initial object graph G. A serialized version of G requires about the same amount of memory as G itself and both sender and receiver hold the object graph and its serialized representation in memory. In the optimized case, the serialized representation is only held in memory once.

Additionally, (de-)serialization is itself a costly operation. It puts significant stress on the memory subsystem, especially the caches. Serializing an object graph requires reading every byte of each object in the graph and writing approximately the same amount of data for the serialized representation; the same for deserialization. This may evict more useful data from the caches. Moreover, sender and receiver only use the serialized representations in B (and B' in the non-optimized case) temporarily. Hence, after the transfer operation has finished, the buffers will not be used again, but accessing them evicts potentially more useful data from the caches.

4.4.2. Cloning-Based Approaches

There is an alternative approach to deep copy an object graph that does not require a serialized representation. In the context of our discussion, we call this operation *cloning* to differentiate it from serialization.

Algorithm 2 shows pseudo code for cloning. The clone operation is a depth-first traversal of G with cycle detection like serialization. Passing the root object of G to the procedure `Clone` returns a deep copy of G. The important difference between cloning and serialization is that cloning does not construct a flat representation of G. Instead, it traverses G and, at each object o, it directly creates a copy o' of o.

Algorithm 2 Object cloning.

```
procedure Clone(O)
   if AlreadyCloned.contains(O) then
      return AlreadyCloned[O]
   O’ ← Allocate(O.size)
   AlreadyCloned[O] ← O’
   for each F in O.type.fields do
      if ¬F.type.pointer then O’.F ← O.F
      else O’.F ← Clone(O.F)
   return O’
```

Cloning is often used on regular shared-memory machines (compare, e.g., `java.lang.Cloneable`). However, it is not dependent on a shared address space and we can also use it on message-passing systems. There, the process is more complicated due to the missing shared address space and requires cooperating actions on sending and receiving side. In general, using cloning with a message-passing system proceeds according to the following scheme (see Algorithms 3 and 4):

- Initially, we copy the address of the root object of G from S to R.
- We now traverse the object graph in lockstep on both sender and receiver. On the sender, at each object we have not visited before, we shallow-copy the object to the receiver. On the receiver, at each object we have not visited before, we receive the shallow object copy o'. We then repair all pointers contained in o' by awaiting additional object copies. We use sender-local addresses to differentiate between visited and unvisited objects on the receiver. However, we never access memory at these addresses on the receiver, hence this is safe.

Algorithms 3 and 4 illustrate the cooperation between sender and receiver. On the receiver, `A` is the sender-local address of the object to be cloned. Initially, it contains the address of the root object. We use C notation and denote with `*p` a dereference operation on address `p`.

In general, cloning-based approaches provide a different trade-off than serialization-based approaches. Cloning avoids constructing a serialized representation. However, each object in the object graph requires a separate message. Hence, cloning trades the need to reformat (i.e., serialize) the object graph for more frequent communication operations.

Algorithm 3 On sender.

```
procedure CloneSend(O)
  if AlreadySent[O] then
    return
  Send(*O)
  AlreadySent[O] ← True
  for each F in O.type.fields do
    if F.type.pointer then
      CloneSend(O.F)
```

Algorithm 4 On receiver.

```
procedure Clone(A)
  if AlreadyCloned.contains(A) then
    return AlreadyCloned[A]
  O' ← Recv()
  AlreadyCloned[A] ← O'
  for each F in O'.type.fields do
    if F.type.pointer then
      O'.F ← Clone(O'.F)
  return O'
```

In principle, we can implement the `send()`/`recv()` pair using the techniques from Section 4.3, i.e., using TLM (Clone-Tlm) or using off-chip memory (Clone-Off). However, in the case of using off-chip memory, there is again an optimization opportunity: why do we even copy the objects? Why not use the instances that are already in off-chip memory?

Hence, we propose the following approach (Clone-Off-Opt), which is a central contribution of this chapter. Our approach proceeds according to the following three step scheme (see Figure 4.15):

1. *writeback*: S forces a writeback of all objects in G. For each object we know its starting address and size. Hence, by traversing G, we can write back the relevant cache lines of each object. Then, S waits until all relevant cache lines have been committed to memory.

2. *notify*: S sends a message carrying the address of the root object of G to R. This notifies R that it is now safe to clone G.

3. *invalidate & clone*: R clones G, resulting in G'. Before reading an object o, R invalidates the relevant cache lines for o. Then, R creates a copy o' of o in R's memory partition.

Again, we have some freedom regarding the placement of data and responsibilities. In our description from Figure 4.15, we used a pull-style

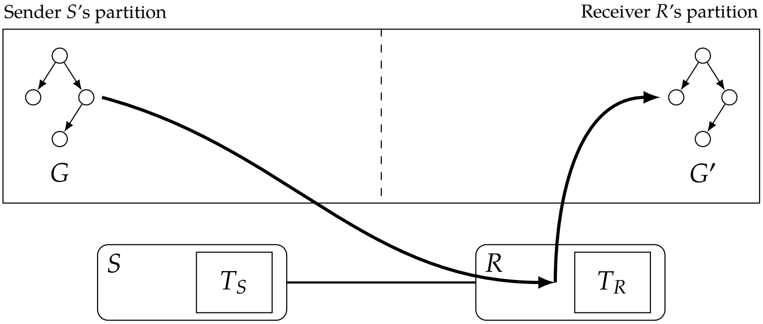


Figure 4.15: Transferring an object graph G using object cloning. G' is the resulting copy of G.

approach, i.e., the receiver performs the actual object cloning. However, it is also possible to use a push approach and let the sender perform the cloning. In this case, the sender traverses G and clones it while placing all newly created objects in R's memory partition. After creating an object o, S writes back the address range of o. After S has finished creating the copy G', S notifies R of the root object's address. R then traverses G' and at each object o invalidates the address range of o. This ensures that R reads up-to-date values for all objects in G'.

In practice, a push-style approach may be more difficult to realize as it requires to create objects in a foreign memory partition. Depending on the memory allocation scheme for objects used by the tiles, the required synchronization can cause significant overhead. For example, the tiles might classify objects according to their size and allocate all objects of a certain size range in dedicated memory areas to reduce fragmentation. Then, with a push-style approach, each object allocation would need to happen on the receiving tile. Hence, each allocation would require a forth-and-back communication between sending and receiving tile. This is expensive for object graphs with many objects. If the tile uses a simpler memory allocation technique, using a push-style cloning approach is unproblematic. With a push-style approach the write-back is integrated into the cloning process, whereas on the receiver we invalidate all address ranges spanned by the objects in the graph. Hence, this is the dual situation to the pull-style approach (compare Algorithms 3 and 4). In general, as PGAS languages prefer one-sided communication, the push-style is preferable in this context.

The main difference between Clone-Off-Opt and serialization-based approaches is that Clone-Off-Opt avoids serialization and thus requires no temporary buffers. Hence, it is cache-friendlier, as no temporary buffers pollute the cache.

For flat data structures, Clone-Off-Opt is equivalent to Ser-Off-Opt. In this case, there is no need for serialization on the sending side ("$G = B$") and "deserialization" is equivalent to copying the single object, i.e., cloning it. Viewed this way, Clone-Off-Opt is a generalization of Ser-Off-Opt from flat to pointered data structures. Viewed another way, Clone-Off-Opt augments the widely-used object-cloning technique with automatic writebacks and invalidations to allow its use on non-cache-coherent systems.

Correctness. At this point, we briefly discuss correctness of our approach. We cannot offer any formal proofs, hence we only discuss this topic informally.

In order to prevent data races, the object graph must not be modified concurrently to the clone operation. With a push-style approach, this requires appropriate synchronization on the sender (the same applies when using serialization). The pull-style variant has one additional complication. Here, the sender must wait until the receiver acknowledges that it has finished cloning the object graph. Hence, the necessary synchronization involves a remote party and can therefore be more expensive.

It is also not intuitively clear that multiple concurrent transfers of the same object graph do not cause issues. Let us first look at the situation where multiple receivers clone the same object graph G using pull-style cloning. Hence, multiple writebacks of G can happen concurrently on the sender. This is unproblematic, as writing back a non-dirty cache line is a no-operation. Hence, multiple cores may compete to write back G, but as long as they properly synchronize afterwards, this does not cause issues.

On the receiver, multiple invalidations of the same address range can happen concurrently. For example, it is possible that core 1 invalidates range A, begins to read data from A, and then core 2 invalidates A again. However, this is unproblematic, as the next read request, even if it comes from core 1, fetches the data from memory again. The data at A must still be the same as before, as otherwise there would be a data race in the program (as G was modified while being cloned). In general, multiple competing invalidations are not a problem, as we do not modify data in the sender's partition. We only read from this memory area, hence competing transfers may invalidate the same cache line multiple times, which causes unnecessary costs, but is harmless.

With a push-style approach, the write-back issued by the sender is potentially problematic. Imagine that we create an object in the receiver's partition and then issue a write-back of the relevant cache lines. If the receiver modified data located at a different location in the same cache line, we overwrite the receiver's changes. However, we can easily prevent this problem by using reserved memory locations for writing to foreign memory partitions (similar to MPI windows [Mes15, section 11.2]).

4.4.3. Related Work

The problem of transferring pointered data structures frequently arises on architectures with physically separate address spaces, such as clusters of machines connected by a network. Various prior work explores the simplification of transferring pointered data structures for programs using MPI [GRR00; WBJ16]. The authors focus on assisting the programmer with writing the necessary serialization routines and orchestrating the necessary communication operations.

A similar problem arises in the context of platforms composed of a host CPU and an accelerator, e.g., a GPU, with separate address spaces. The OpenACC standard [Ope17] provides an API for offloading tasks from CPU to accelerator. This involves copying data from the CPU's address space to that of the accelerator. As of version 2.5 of the standard, OpenACC only allows transferring flat data structures. Beyer et al. [BOS14] report that, according to user feedback, this restriction is the most important impediment to porting interesting data structures and algorithms to

OpenACC. Beyer et al. propose a solution based on compiler directives that allow (semi-)automatic deep-copy support.

This body of work shows that transfers of pointered data structures occur frequently and are important in real-world programs. In contrast to this body of work, non-cache-coherent architectures do allow accesses to remote memory partitions, albeit without coherence guarantees, as we only partition memory on a logical level.

In the context of non-cache-coherent architectures, our work is closely related to the work on data transfers presented in Section 4.4.1. However, this body of work only considers flat data structures. In some cases [CHH11], authors mention pointered data structures, but use serialization with the message-passing approach.

Regarding pointered data structures, it is interesting to look at the work of Prescher et al. [PRN11; Rot+12] and Lyberis et al. [Lyb+12b]. Prescher et al. present MESH, a C++ framework for distributed shared memory that supports non-cache-coherent architectures. While X10 enforces a central instance for each object, i.e., each object exists on exactly one place, MESH allows choosing between different sharing models (object replication, central instance, and mixtures of both). However, MESH is library-based as opposed to our compiler- and language-based approach. As such, existing software must be modified to be used with MESH. Moreover, their implementation requires a consistency-controller object per shared object and triggers additional communication for coherence management. We avoid this overhead, as we manage coherence in a more restricted environment under control of the compiler.

Lyberis et al. present Myrmics [Lyb+12b], a memory-allocation scheme based on regions aimed at non-cache-coherent architectures. They observe that transferring pointered data structures using messages is an expensive and complicated operation. As discussed above, low-level libraries, such as MPI, provide no dedicated support, i.e., the programmer must orchestrate the necessary serialization and deserialization.

Lyberis et al. propose to use regions, which are growable memory pools that contain objects. The Myrmics runtime system ensures that objects allocated in such a region have a globally unique address across all coherence domains. This means that if one coherence domain allocates an object at a globally valid address A, no other coherence domain allocates an object at A even if that address is backed by distinct memory on different coherence domains.

Users can then allocate logically associated objects, e.g., each element of a linked list, in the same region. If a core from a different coherence domain requires access to that data structure, Myrmics can transfer the whole region it is contained in as one block. As the object addresses, i.e., the pointers, are globally unique, even pointered data structures are valid without modification after a transfer. Hence, the receiving domain can operate locally on the copied data structure using the same pointers as the sending domain. Thus, Myrmics avoids the need for pointer translation completely.

However, using this approach requires program modifications. Specifically, the programmer must identify data structures that may be shared and use regions (and sub-regions) accordingly. Moreover, this approach requires a virtual-memory subsystem. Otherwise, the same addresses cannot refer to different memory locations on different domains.

Kumar et al. present HabaneroUPC++ [Kum+14], a C++ library that enables an asynchronous PGAS programming style in C++. The library implements constructs similar to those found in X10 (mainly **finish**, **async**, and **at**). The programmer must transfer necessary data manually.

Transferring non-contiguous data types is a frequent operation [KHS12]. Therefore, MPI provides explicit support for specifying so-called derived data types [Mes15, chapter 4]. For example, derived data types allow to describe the transfer of the first column of a matrix that is saved row-wise in memory. An MPI implementation can decide to pack this data into a contiguous format before the transfer. Alternatively, certain interconnect hardware supports the transfer of such derived data types directly from memory (“zero copy”) [KHS12]. However, derived data types still require regularly structured data and are unsuitable for irregular pointered data structures. For the same reason, scatter/gather DMA transfers are not ideal, as they can copy non-contiguous data structures but only make shallow copies.

4.4.4. Implementation on the Hardware Prototype

In this section, we describe the concrete implementations of the data-transfer techniques for invasive architectures that we use in our evaluation in Section 4.6. Our starting point is an object graph G located in the off-chip memory partition of sending tile S. We want to transfer G with deep-copy semantics to G' in the off-chip memory partition of a receiving tile R. First, we briefly describe our implementation of serialization needed for the message-passing-based approaches. Then, we cover the implementation of the actual data-transfer techniques.

For serialization, we implement Algorithm 1. We do not iterate over all fields of a type at run-time, i.e., we do not use reflection. Instead, our X10 compiler generates specialized serialization functions per type. At run-time, we invoke a type-specific serialization function that knows about the memory layout of the specific type. Therefore, it can directly invoke the serialization function for all non-pointer fields. For pointer fields, due to subtyping, we do not necessarily know the run-time type of objects in the graph. Hence, in general, we have to dynamically dispatch calls to serialization functions for pointer fields using the vtable mechanism.

To implement Ser-Tlm, we first serialize G to a contiguous buffer B in the off-chip memory of S. Now, we have to transfer B to B' in R's partition. We saw in Section 4.3.4 that the preferred data-transfer technique is via DMA transfers from TLM to a remote memory. Unfortunately, the current prototype (cf. Section 3.6) only supports TLM as the possible remote memory type. Hence, we copy B to a buffer B_S in the sending tile's TLM, transfer it to R's TLM using a DMA transfer, and then deserialize a copy G' (see Section 4.3.4 for details).

This process may seem overly redundant. However, suppose we have a library that implements message passing using DMA transfers between TLMs on invasive architectures. In this case, the X10 compiler is in charge of serializing the object graph G and passes the resulting buffer B to the library. On the receiving side, the library expects to be passed a buffer B' to store the received data, as the data cannot stay in TLM permanently. Hence, the X10 compiler would allocate a buffer B' on the receiving side as well. Thus, (possibly redundant) copying of data between off-chip memory and TLMs may happen due to library usage.

To implement Ser-Off-Opt, we first serialize G into a contiguous buffer B in the off-chip memory partition of S. Then we force a write-back of B's address range using the previously described process. We wait until all cache lines have been written to memory. Then, we notify the receiving tile using an *i*-let, which carries the address of B. We store the buffer size in B before the actual payload. On the receiving tile, the *i*-let invalidates B's address range. Then, it deserializes a copy G' of the original object graph from B.

To implement Clone-Off-Opt, we choose pull-style object cloning. Hence, we first force a write-back of each object in the object graph G. As with serialization, our X10 compiler generates type-specific write-back functions. We implement this write-back operation using a modified variant of Algorithms 1 and 2; see Algorithm 5. Hence, the generated functions perform a depth-first traversal of G and at each object write back its respective address range. We use the C-style syntax `p + x` with an address `p` and offset `x` for byte-wise address arithmetic. As with serialization, if the type of an object is statically known, we can directly invoke the matching writeback function. Otherwise, we use dynamic dispatch.

Algorithm 5 Type-specific write-back function.

```
procedure Writeback(O)
    if AlreadyVisited.contains(O) then
        return
    WritebackRange(O, O + O.size) // Automatic write-back
    AlreadyVisited[O] ← True
    for each F in O.type.fields do
        if F.type.pointer then
            Writeback(F)
```

Subsequently, we wait until all write-back operations have finished and then notify the receiver using the method discussed in Section 4.3.4. After that, the receiver clones G. We implement Algorithm 2. Again, our X10 compiler generates specialized clone functions per type. However, there is one crucial alteration we make to Algorithm 2: we adapt the code generation so that, before we access an object o from S's partition, we invalidate *o*'s address range. See line 6 in Algorithm 6. As every access to objects from *S*'s partition happens in compiler-generated clone functions, we issue invalidation commands for exactly the necessary memory regions. Other data from *R*'s own partition is accessed normally.

Algorithm 6 Object cloning with integrated cache invalidation.

```
 1  procedure CloneInv(O)
 2     if AlreadyCloned.contains(O) then
 3        return AlreadyCloned[O]
 4     O' ← Allocate(O.size)
 5     AlreadyCloned[O] ← O'
 6     InvalidateRange(O, O + O.size) // Automatic invalidation
 7     for each F in O.type.fields do
 8        if ¬F.type.pointer then O'.F ← O.F
 9        else O'.F ← CloneInv(O.F)
10     return O'
```

As we can only invalidate our L1 cache completely, we employ the following method to avoid invalidating the whole cache each time we read an object. We invalidate the relevant L2 cache lines before reading an object. However, we ignore the L1 cache and issue no operations. After visiting all objects of the graph, we thus have an L2 cache that does not contain valid lines from foreign memory partitions. Of course, our L1 cache can still contain such lines. We now issue *one* invalidation of the complete L1 cache. As our L1 cache is configured in write-through mode, this invalidates all cache lines that cache data from foreign memory partitions while not discarding local modifications. The following L1 cache misses are mostly compensated by the L2 cache.

There is one remaining complication that we need to discuss. The X10 runtime system handles statements of the form `at (B) async S` specially. Such a statement immediately terminates locally (cf. Section 3.4). Hence, this construct is the X10 equivalent of an active message [Eic+92]. When using a push-style approach, after preparing the object graph (either via serialization or cloning), we can spawn the activity on the remote place and require no further synchronization. With our pull-style approach, we have to wait for the remote tile to finish cloning the object graph.

4.5. Hardware Support

In this section, we propose a hardware extension to allow the invalidation, write-back, and flushing of address ranges. We first motivate this hardware extension and discuss the design space. We then describe our implementation and relate it to existing work. The complete hardware was implemented by Michael Mechler and Carsten Tradowsky [MT17; Mec16; Tra16].

We saw in Sections 2.4, 4.3.4 and 4.4.4 that software-managed coherence is a fundamental operation to implement efficient communication on non-cache-coherent architectures, regardless of the programming model used. However, the granularity with which we manage coherence may differ. An implementation of the shared-memory programming model may require managing the coherence of individual variables, i.e., the ability to operate on individual addresses and their respective cache lines or even words within cache lines (cf. Section 2.2.2.3). On the other hand, with a message-passing or PGAS programming model, we operate on a coarser granularity, e.g., contiguous buffers or contiguous objects. Here, we often require the ability to invalidate or write back address ranges.

It is easy to build such range-based cache operations using line-based variants. Assume that the hardware provides means to invalidate the cache line associated with a particular address. As mentioned before, we can then invalidate a whole address range $[S, E]$ using the following program, assuming a cache line size of L bytes:

```
for x := $S - (S \bmod L)$ to $E - (E \bmod L)$ step $L$:
  invalidate(x)
```

Again, we use mod to denote the modulus operation on integers. The term $A - (A \bmod L)$ rounds the address A down to the nearest multiple of L.

However, as communication operations using these range-based cache operations may be frequent, better hardware support is desirable. We are not the first to realize that fact. We found multiple instances of this insight in the literature on non-cache-coherent architectures:

- Peter et al. [Pet+11a, section IV.B] write:

  > In our message-passing implementations, we generally know precisely which addresses we wish to invalidate. Consequently, we would find more fine-grained cache control very useful. An instruction which would invalidate a region around a given address would be ideal for us.

- Rotta et al. [Rot+12, section VI] write:

  > The presented framework would benefit from a write-back and a write-back-invalidate instruction on logical address ranges.

- Christgau et al. [CS16, section 8] write:

  > Consequently, more fine-grained control is required to prevent unnecessary invalidation. Therefore, it would be beneficial to supply the starting (virtual) address and the size of the region to the invalidation instruction.

Thus, we see that ranged-based cache operations are generally considered useful on non-cache-coherent architectures.

4.5.1. Design Space

In the following, we discuss the design space of range-based cache operations. We assume a standard cache architecture (cf. right half of Figure 4.17), i.e., the cache is organized in *cache lines*. Each cache line saves a *tag* for identifying the cache line as well as two status bits *valid* and *dirty* to signal whether the data of the cache line is up-to-date (valid), or up-to-date but locally modified (dirty).

Conceptually, we desire instructions `invalidate` and `writeback`, which operate on address ranges, with the following semantics:

- `invalidate S E` invalidates all cache lines that hold data from the address range $[S, E]$. Hence, this instruction clears the valid bit of all affected cache lines. It performs invalidation regardless of whether the cached data is marked as modified or not.

- `writeback S E` writes back the contents of all cache lines that are marked as modified and hold data from the address range $[S, E]$. Hence, this instruction writes the data of the affected cache lines to the next component in the memory hierarchy (i.e., the next cache or the backing memory) and clears their modified (dirty) bits.

Ideally, these hypothetical instructions would run in one clock cycle. This is feasible in theory; however, the overhead would be significant. Essentially, such a fully parallel implementation of this concept would require two hardware comparators per cache line. These comparators would need to compare the address range given in the instruction against the tag of the respective cache line. In case the cached address is part of the address range, the respective action (invalidation or writeback) would be triggered.

The area overhead of having two comparators per cache line seems prohibitive. Typical addresses have 32 or 64 bits. We only need to compare the number of bits occupied by the tag of a cache line. While the tag typically has fewer bits, depending on the concrete cache structure, the required comparators for that number of bits would still cause massive area overhead.

Therefore, the designers of the Intel SCC chose a different approach. They introduced an additional status bit, “MPBT” for “message-passing buffer type”[19], for each cache line in the L1 cache. Software can mark memory regions as MPBT memory with page granularity. Reads from MPBT memory get cached in the L1 cache, but bypass the L2 cache. L1 cache lines holding data from MPBT memory have their respective MPBT bit set. The Intel SCC then provides the instruction `CL1INVMB` to invalidate all cache lines marked as MPBT in the L1 cache. This instruction runs in one clock cycle.

Unfortunately, there is no official documentation available about the internal implementation of this instruction. We suspect that it is implemented using 1-bit comparators, i.e., a single AND gate, per cache line. When a core executes the `CL1INVMB` instruction, the hardware checks the MPBT

[19] Named after the SCC’s on-chip memories (message-passing buffers, cf. Section 2.3.1). This memory type was designed to be used with message-passing buffer memory, but is not limited to it, i.e., off-chip memory can be marked as MPBT as well [Mat+10].

bit of all cache lines in parallel. For each cache line, if its MPBT bit is set, the hardware clears the valid bit of the respective cache line.

The Intel SCC does not provide a counterpart for writing back cache lines. Writes to MPBT memory do not get cached. However, the Intel SCC offers a "write-combine buffer" that aggregates writes. This buffer is flushed once a full cache line has been written, or a different cache line is written to. Thus, as long as applications are aware of this behavior, no explicit writeback functionality is required.

While `CL1INVMB` allows the efficient invalidation of many cache lines at the same time, it is imprecise. This may cause the unneeded invalidation of cache lines, resulting in unneeded memory accesses the next time these addresses are accessed. In fact, all statements from the authors quoted at the end of the previous section referenced the Intel SCC's imprecise invalidation instruction. Hence, this design sacrifices precision to achieve the goal of low running time (one clock cycle) with manageable area overhead.

In the following, we propose a different trade-off to approximate the ideal of range-based cache operations that complete in one clock cycle. Our proposal does not impede precision, i.e., it operates exactly on the cache lines belonging to a given address range. Instead, it compromises on the running time: our proposed instruction takes one clock cycle (from the view of the processor) only in the best case; in the worst case it takes n clock cycles to operate on an address range spanning n cache lines.

4.5.2. Concept and Implementation

In the following, we present our concept and implementation of non-blocking range-based cache operations (or range operations for short). Our range operations offload the work to an enhanced cache controller. The underlying processor for our implementation is a Gaisler LEON 3, which implements the SPARC V8 ISA (cf. Section 3.6). However, neither our concept nor our implementation are tied to this particular ISA or microarchitecture. We first present the instruction format we used and then discuss our implementation of the cache-controller logic.

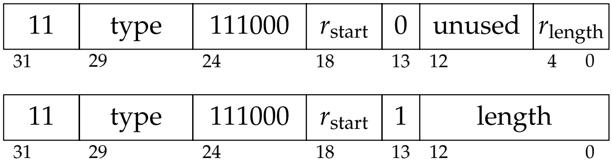


Figure 4.16: Instruction encoding of range operations. The type field encodes the operation type (invalidate, writeback, flush); r_{start}, r_{length} are register operands; length is a 13-bit immediate.

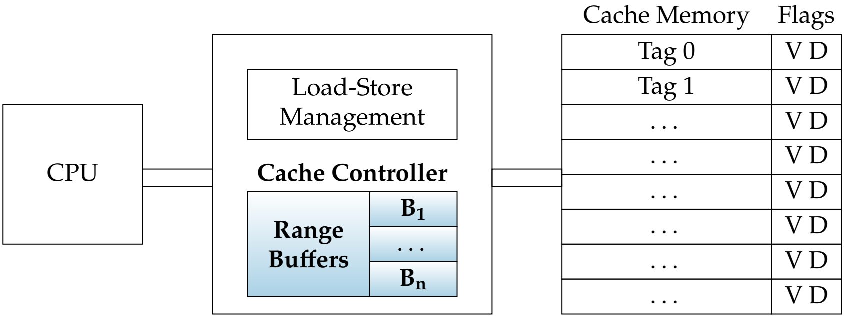


Figure 4.17: Schematic view of our modified cache architecture. The modified cache controller containing the range buffers is highlighted bold.

We develop new instructions compatible with the SPARC V8 ISA. Figure 4.16 shows the encoding used for the range operations. We use the instruction format for load/store instructions and describe the address range as a starting address and a length in bytes. The starting address must be supplied in a general-purpose register. The length can be given as either a 13-bit immediate or in a register. During the Execute stage of the pipeline, the accumulator calculates the end address (cf. Section 5.2 for details on the pipeline structure). During the Memory stage, start and end address are forwarded to the cache controller.

Figure 4.17 shows a schematic view of the modified cache architecture with changed parts of the cache controller highlighted bold. First, we add an interface to transfer the operation type as well as the affected address range from processor to cache controller. When the processor executes a range operation, the processor pipeline is halted and control is transferred to the cache controller.

Then, we extend the cache controller with the ability to invalidate, write back, or flush multiple cache lines. We implement our range operations as multi-cycle instructions. Thus, we enhance the cache controller with a simple state machine that iterates over the address range specified by the range operation. The cache controller can modify one cache line per clock cycle. Modification consists of performing an address lookup and, if a cache line is present, applying the respective operation. Hence, it takes n clock cycles to apply a range operation spanning n cache lines. Essentially, this implements the loop from our introduction of Section 4.5 in hardware.

However, our initial goal was an instruction that completes in one clock cycle. An obvious shortcoming of our current approach is that we halt the processor until the range operation has finished, i.e., our range operations are blocking. Can we drop this restriction and let the processor continue executing its program? For non-memory-related instructions, e.g., arithmetic or control-flow instructions, this is unproblematic[20], as these instructions do not require any cache functionality. However, load or store instructions must perform cache lookups, which interferes with the lookups performed by our range operation logic. Additionally, they might access an address that is part of the range the cache controller operates on. It is not obvious how to handle this situation correctly.

We propose the following design. In order to make our range operations non-blocking, we add *range buffers* B_i (cf. Figure 4.17) to the cache controller. Each range buffer holds a triple (s, e, t) of start address s, end address e, and operation type t (invalidation, writeback, or flush). Each time the processor executes a range operation on a range A, the cache controller stores A along with its operation type in a range buffer as follows:

(i) If there is no free range buffer, we halt the processor until a buffer becomes free.
(ii) If A overlaps with a range A' already stored in another buffer, we halt the processor until A' has been processed.
(iii) Otherwise, we store A and its type in a free range buffer.

[20] Assuming a load/store architecture.

Then, the processor continues executing the program. Every time it executes a load or store to an address D, the cache controller checks D against all stored ranges. We perform the checking in parallel using separate comparators for each range buffer. If $D \in A$ for a stored range A, we halt the processor until the operation on A has finished. Otherwise, we perform a cache lookup as usual.

We call a clock cycle, during which the processor does not execute a load or store instruction, a *spare cycle*. We observe that, from the view of the cache controller, the cache is idle in every clock cycle where the processor does not lookup an address. Our modified cache controller uses these spare cycles to work on range operations. Hence, during every spare cycle, as long as there is at least one range A stored in a range buffer, the cache controller applies the respective operation to the next cache line relevant for A, e.g., clearing a line's valid bit for an invalidation. The cache controller keeps track of its progress using an internal register.

Therefore, it takes n spare cycles to apply an operation to a range spanning n cache lines. In the best case, between the execution of two range operations, there (i) are at least n spare cycles, and (ii) we execute no interfering load or store instructions. Then, the first range operation takes only one clock cycle from the view of the processor.

In summary, our proposed instructions inhabit a new point in the design space of instructions that support software-managed coherence. They are precise and operate exactly on the given address range. To limit the area overhead, they compromise on the running time and give no hard guarantee: execution can take 1 clock cycle, but may take up to n clock cycles for an address range that spans n cache lines[21].

4.5.3. Related Work

As already mentioned in the introduction of Section 4.5, our range operations are related to the `CL1INVMB` instruction of the Intel SCC. The execution time of that instruction is guaranteed to be one clock cycle. However, it is imprecise as it invalidates all cache lines of a certain type.

[21] If the cache is shared between multiple cores, execution may take more than n cycles due to interference with cache accesses of other cores.

Multiple authors [Pet+11a; Rot+12; CS16] criticize this lack of fine-grained control.

Range-based cache operations have been implemented before. Certain processors from the ARM11 family, e.g., the ARM1136J(F)-S processors, can perform invalidation, writeback, and flushing of address ranges via a system control coprocessor [ARM09, section 3.3.17]. The range operations are blocking. In contrast, our concept only requires an enhanced cache controller instead of a full-blown coprocessor. Additionally, we provide non-blocking range operations.

4.6. Evaluation

In the following, we evaluate some of the presented data-transfer techniques on our hardware prototype. First, we investigate transfers of flat data structures and analyze the performance of Ac-Tlm and Ac-Off. We use a synthetic benchmark program. Then, we turn towards complex data structures. Here, we analyze the performance of Clone-Off-Opt compared to Ser-Tlm and Ser-Off-Opt. We first consider individual data transfers using a synthetic benchmark and then look at distributed benchmark programs from an existing test suite. We perform all experiments on the invasive hardware prototype. Finally, we investigate overhead and benefit of our cache controller extension.

4.6.1. Setup

We conducted all running time measurements on the invasive hardware prototype described in Section 3.6. Recall that the architecture consists of 4 tiles with 4 cores each. Each tile forms a coherence domain, where cache coherence is guaranteed by a classical bus snooping protocol. However, there is no cache coherence between tiles. The tiles are connected by the invasive network on chip [Hei+14] (NoC).

All cores are Gaisler SPARC V8 LEON 3 [Cob17b; SPA92] processors. Each processor has a private 16 KiB 2-way instruction cache with a cache line size of 32 bytes and a private 8 KiB 2-way write-through L1 data cache with a cache line size of 16 bytes. Additionally, the 4 cores of each tile share a 64 KiB 4-way write-back L2 cache with a cache line size of 32 bytes. Each tile has 8 MiB of SRAM-based on-chip memory. Tile 3 has 256 MiB of DDR3 memory, used as shared memory, attached to its internal bus. We do not execute any application code on this tile during our experiments. Hence, all cores used by our applications access the off-chip memory via the NoC.

We only use two compute tiles to increase stability in case of concurrent DMA transfers[22]. As two tiles still require inter-tile data transfers, two tiles are sufficient for our purposes. The hardware design was synthesized to a CHIPit Platinum system [Syn15], a multi-FPGA platform based on Xilinx Virtex 5 LX 330 FPGAs[23].

On the software side, we use X10 as our PGAS programming language. We use the modified X10 compiler (cf. Section 3.5) based on version 2.3[24]. We compiled all programs using the `-O3` flag. We use OctoPOS (cf. Section 3.3) as our operating system[25]. We use the `perf` variant of OctoPOS, which results in an optimized build with disabled assertions and without sanity checks. We used the `hwcpy` variant, which uses hardware-accelerated data transfers between TLMs.

We use a conservative stop-the-world garbage collector [BW88][26] for memory management on each tile. We compiled all C components of our software stack using the official SPARC toolchain provided by Gaisler [Cob15a], which is based on GCC 4.4.2. We used GRMON [Cob17a] version v2.0.69.1 to load and run binaries[27]. We used temci [Bec16] version 0.7.9 to analyze some of our benchmark data. As we work on custom hardware, we did not use any benchmark-data acquisition tools provided by temci; we only used it to visualize and analyze already collected benchmark data.

[22] At the time of writing, these problems have been fixed in current hardware revisions.

[23] We used hardware revision `2016_04_18` from April 18, 2016.

[24] We used Git revision `1faf26498de2eb3e25f85bdc0e74a5f9b816ab59`.

[25] We used Git revision `510073385ec96b75fafdd91c0aac894f99357315`. This revision is based on Git revision `741f34079a5e968d6002b7e8d3270a2b0f58fe07`, but adapts some parameters, such as using fewer but larger contexts, i.e., stacks, for *i*-lets.

[26] We used Git revision `5f1b891d30626dc4074686aa2ea061356c635b93`.

[27] Extended with `grmon_tools` Git revision `65189723b9cee70505a66ceff2244ed9bd826524`.

4.6.2. Establishing an Evaluation Environment

On our FPGA-based prototype, latency and bandwidth differences between TLMs and off-chip memory are not as pronounced as on a real ASIC. We measured latency and bandwidth of reading accesses to local TLM, remote TLM, and off-chip memory on the default hardware design. We use cycle-accurate performance counters provided by the NoC. We determined the cost to query the performance counter to be 6 cycles and substract it from all measurements. We performed the bandwidth measurements with a fully unrolled loop that uses double-word load instructions to read 256 bytes. We report minimum latency and maximum bandwidth numbers, as we are interested in the best case.

	Latency (in clock cycles)	Bandwidth (in MiB s^{-1})
Local TLM	13	19.0
Remote TLM	99	4.0
Off-chip	104	3.8

Table 4.1: Memory latency and bandwidth numbers on the default hardware prototype design.

Table 4.1 shows the resulting numbers. We see that reading from remote memory is significantly more expensive than accessing local memory, as all accesses to remote memory proceed via the NoC. However, the difference between remote TLM and off-chip memory is miniscule, which is unrealistic. The whole point of adding TLM to a non-cache-coherent system is to improve latency and bandwidth.

Hence, we would like to investigate the performance of data-transfer techniques on a system that is more realistic than our default FPGA design. So how do latency and bandwidth numbers look on other non-cache-coherent architectures, preferably ones available as an ASIC? We use the Intel SCC as a reference as it is comparatively well-documented.

First, let us look at the memory latencies. Let C_c, C_n, and C_d be the cycle lengths of core, network-on-chip, and DRAM. On the Intel SCC, the latencies L to read one 32-byte cache line are as follows [PN14].

- Reading from local on-chip memory: $L_{\text{local}} = 45 \cdot C_c + 8 \cdot C_n$.
- Reading from remote on-chip memory: $L_{\text{remote}} = 45 \cdot C_c + k \cdot 8 \cdot C_n$.
- Reading from DRAM: $L_{\text{DRAM}} = 40 \cdot C_c + k \cdot 8 \cdot C_n + 46 \cdot C_d$.

Here, k is the number of hops in the network-on-chip from source to destination ($0 \leq k \leq 8$).

However, computing these latencies is not straightforward, as the Intel SCC's cores, network-on-chip, and DRAM can be clocked at different frequencies. The possible frequencies (in MHz) are $f_c \in \{533, 800\}$, $f_n \in \{800, 1600\}$, and $f_d \in \{800, 1066\}$. We assume the recommended setting of $f_c = 533\,\text{MHz}$, $f_n = 800\,\text{MHz}$, and $f_d = 800\,\text{MHz}$, which is also used in most papers. The latencies compute to (assuming $k = 4$ as an average and rounding up the values):

- $L_{\text{local}} = 90\,\text{ns}$
- $L_{\text{remote}} = 110\,\text{ns}$
- $L_{\text{DRAM}} = 165\,\text{ns}$

Hence, accessing DRAM on the Intel SCC has a roughly 50% higher latency than accessing on-chip memory. However, the SCC is only a prototype chip as well. Its core clock of 533 MHz is low in comparison to current microprocessors. Therefore, on a non-prototype chip the core and network-on-chip frequencies would be several times higher than the DRAM frequency. This would likely increase the latency difference between accessing on-chip and off-chip memory even further.

Second, let us look at bandwidth on the Intel SCC. Van Tol et al. [Tol+11] report bandwidth measurements on the Intel SCC, also with the standard frequency settings. The Intel SCC has four memory controllers resulting in a theoretical peak transfer rate of $6.4\,\text{GiB}\,\text{s}^{-1}$. Van Tol et al. determine the maximum bandwidth load generated by one core to be about $107\,\text{MiB}\,\text{s}^{-1}$ when reading from off-chip memory (and not from a cache). For the case when 48 cores read from off-chip memory at the same time, van Tol et al. measure a peak bandwidth requirement of $5.9\,\text{GiB}\,\text{s}^{-1}$. Hence, even 48 cores cannot saturate the available memory bandwidth. Again, this is unrealistic, as current multi-core processors can easily saturate the memory bandwidth with far fewer than 48 cores. Hence, the Intel SCC can serve as a guideline, but is not an ideal example of realistic latency or bandwidth numbers.

Besides the lack of clear target numbers for latency and bandwidth, also emulating more realistic hardware on an FPGA is challenging. For example, we cannot arbitrarily reduce the clock frequency of our DRAM chips as they are very sensitive to timing, especially regarding the necessary DRAM refresh cycles. On the other hand, increasing the clock frequency of the cores is not possible either due to critical-path lengths and timing constraints.

Hence, we focus on one aspect that we can influence comparatively easily: latency. We modified the default hardware design to artifically increase off-chip memory latency by 1000 clock cycles[28]. We consciously chose the high penalty of 1000 clock cycles to clearly separate the default design from the modified design. In the modified design, additional logic has been inserted into the network adapter of each compute tile. If a core (or the L2 cache) issues a load or store request to an address that is backed by off-chip memory (i.e., DRAM), the network adapter artificially delays the serving of this request by 1000 clock cycles. All other requests proceed normally. Hence, the delay is implemented on the requesting side of memory accesses.

In the following, we call the default hardware variant Hw-Default[29], and the modified hardware variant with artificial off-chip delay Hw-Delay[30].

	Latency [clock cycles]	Bandwidth [MiB s^{-1}]
Local TLM	13	19.0
Remote TLM	99	4.0
Off-chip	1104 (+1000)	0.7 (−3.1)

Table 4.2: Memory latency and bandwidth numbers on Hw-Delay with artifical DRAM latency. We list the absolute change compared to Hw-Default in parentheses.

Table 4.2 shows latency and bandwidth numbers on Hw-Delay. We observe exactly the artificial latency penalty of 1000 clock cycles for off-

[28] Sven Rheindt provided the modified hardware design.

[29] We used hardware revision `2016_04_18` dated from April 18, 2016.

[30] We used hardware revision `2016_04_18_delay_ddr_ls` dated from December 7, 2017 based on the design from April 18, 2016.

chip accesses. At the same time, the increased latency also significantly reduces the bandwidth of off-chip accesses.

Now, we we have two designs that represent extremes: Hw-Default, where off-chip memory is almost as fast as remote TLMs, and Hw-Delay, where off-chip memory is significantly slower, even more so than on the Intel SCC. To investigate the behavior of our data transfer approaches at these two extremes, we perform all following benchmarks on both hardware designs. As the hardware designs are functionally equivalent, we can run the same binaries on both hardware variants. For each benchmark, this provides us with lower and upper bounds for the running times. The running time on a realistic system would likely fall somewhere in this range, depending on the parameters of its memory system.

4.6.3. Block-Based Data Transfers

We now compare our implementations of Ac-Tlm to Ac-Off on the invasive hardware prototype. We use a synthetic X10 benchmark program[31] that issues one-sided copy operations via `Rail.asyncCopy()` to copy data from the sender's off-chip memory partition to the receiver's off-chip memory partition. We vary the size of the transferred memory block from 2^5 bytes to 2^{18} bytes. We measure running times using cycle-accurate counters provided by the NoC and use 25 as the divisor to compute microseconds (as our cores run at 25 MHz). We repeat each experiment at least 50 times; for small transfer sizes we use 200 iterations to stabilize measurements. We verify the received data after the transfer; this verification step is not part of the measured running time.

We expect to see that Ac-Off is faster at least for large transfers. In this case, taking the detour via TLM should be slower. For small transfers, the situation is not as clear. Here, if data is served from cache to TLM, transferred to another TLM, and then again cached, blocking off-chip memory accesses are completely avoided. Hence, we might see a break-even point, i.e., Ac-Off is faster for memory blocks larger than some threshold.

[31] We used Git revision `0888071439f58a06c116fd56af46bd4c74b41905`.

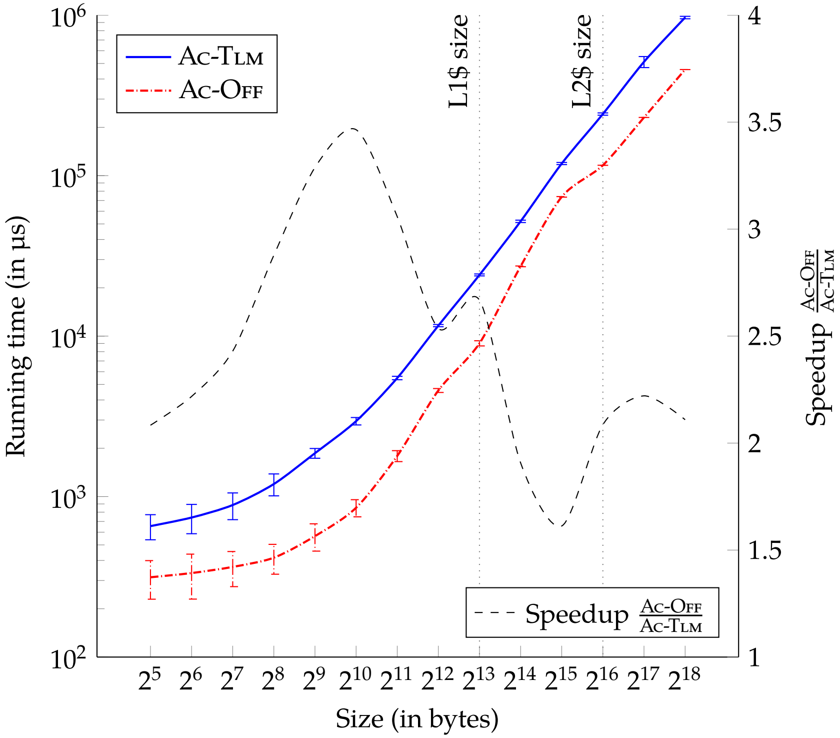


Figure 4.18: Running times (in microseconds) and speedup of AC-OFF over AC-TLM on HW-DEFAULT. We issue individual block-based transfers of sizes ranging from 2^5 bytes to 2^{18} bytes. We show standard deviations with error bars. Both axes are logarithmic.

Figure 4.18 shows the running times of both approaches, as well as the computed speedups of Ac-Off over Ac-Tlm using Hw-Default. Here, directly transferring the data to off-chip memory is always faster than going via TLMs. For small buffers, the standard error is quite high, but still allows us to state that Ac-Off is about 2× faster than Ac-Tlm. The speedup then reaches its maximum of about 3.5× for transfers of 1 KiB. After exceeding the size of the L1 cache, the speedup decreases significantly. Interestingly, as we reach the size of the L2 cache, the speedup increases. We do not have a convincing theory to explain this observation.

Figure 4.19 shows the running times of both approaches and the computed speedups on Hw-Delay. We see that for buffer sizes less than 2^8 bytes, the standard deviations are too high to declare one of the approaches superior. For buffers of size 2^9 bytes and higher, Ac-Off is about 2.5×–3× faster than Ac-Tlm. As we exceed the size of the L1 cache, the speedup declines sharply and stays between 1.5× and 2×. Suprisingly, we do not observe a break-even point. This may be due to the requirement of our platform to copy data to TLM before issuing DMA transfers.

These speedup numbers agree with the numbers reported by Christgau et al. [CS16] who measured a speedup of 2× to 5× on the Intel SCC using one-sided communication via off-chip memory relative to on-chip-based message passing. They performed their measurements in the context of MPI one-sided communication using `MPI_Put`. Just as in our experiment, the one-sided version was always faster, even for small transfers.

4.6.4. Transfers of Pointered Data Structures

In the following, we look at transfers of pointered data structures. We select a subset of the presented data-transfer techniques and analyze the performance of Clone-Off-Opt compared to Ser-Tlm and Ser-Off-Opt. Comparing to Ser-Tlm shows whether using on-chip memory is worthwhile. Comparing to Ser-Off-Opt shows whether it is worthwhile to avoid serialization.

We first consider individual data transfers using synthetic benchmarks and then look at distributed benchmark programs from an existing test

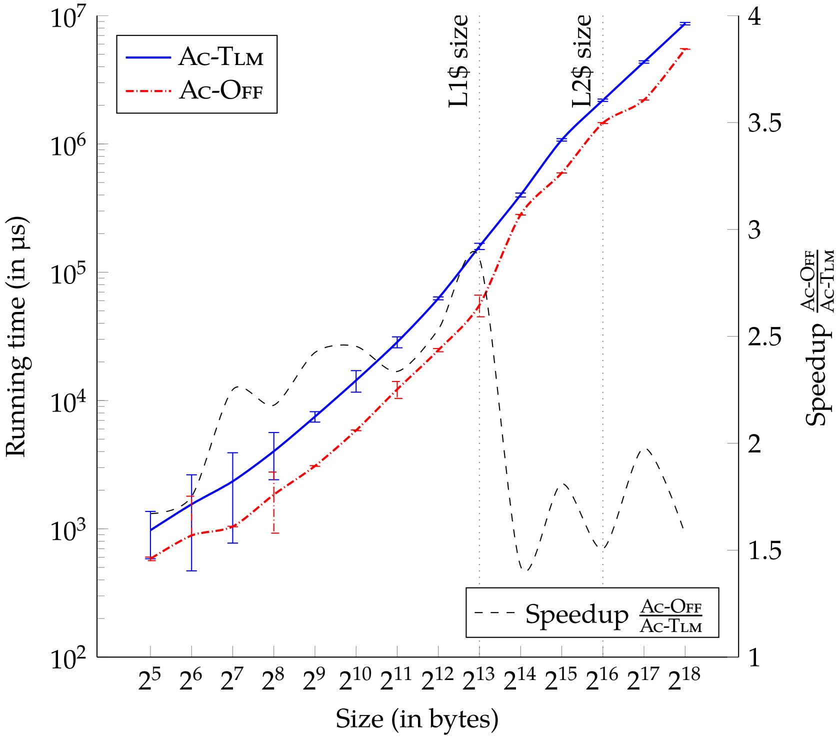


Figure 4.19: Running times (in microseconds) and speedup of AC-OFF over AC-TLM on HW-DELAY. We issue individual block-based transfers of sizes ranging from 2^5 bytes to 2^{18} bytes. We show standard deviations with error bars. Both axes are logarithmic.

suite. For execution on the invasive hardware prototype, we used the scripts developed as part of the `octopos-testsuite-infrastructure` project[32]. We repeated each experiment 50 times unless otherwise noted. As our software has full control over the hardware and there is no resource virtualization, the running time was highly deterministic. The standard deviation for all runs was below 0.1%, so we omit giving standard deviations and report minimum running times. The Ser-Tlm approach did not have to split messages in our experiments as our TLMs provide ample size.

4.6.4.1. Individual Data Transfers

First, we look at individual transfers using a synthetic benchmark[33]. We transfer a circular doubly linked list and vary two parameters: the number of list elements n and the size per list element E. We create a new data structure for each transfer. We compare Clone-Off-Opt to Ser-Tlm and Ser-Off-Opt. We first perform experiments on Hw-Default and then on Hw-Delay.

Regarding the presentation of our benchmark results, we decided to use a series of tables. We have a two-dimensional parameter space (n and E), running times for three approaches, and the computed speedups. We tried using a visualization as in Figures 4.18 and 4.19. However, the resulting three-dimensional plot was difficult to read and understand. Hence, we use four tables. Each has our two-dimensional parameter space as rows and columns, and as table entries shows the computed speedup numbers for Clone-Off-Opt over either Ser-Tlm or Ser-Off-Opt on either Hw-Default or Hw-Delay, resulting in four tables. In all following experiments, we measure speedups for lists from $n = 1$ to 256 elements with element sizes E ranging from 64 bytes up to 4 KiB.

On Hw-Default, we expect Clone-Off-Opt to be superior to Ser-Tlm, except maybe for certain medium-sized object graphs. For these graphs, their serialized representation would fit into the cache, hence no blocking off-chip memory access occur. Furthermore, the hardware-accelerated

[32] We used Git revision `4edfb2558b8c5bde5fe18ba76da86ac6cd50538c`.

[33] We used Git revision `1faf26498de2eb3e25f85bdc0e74a5f9b816ab59`.

DMA transfer between TLMs could outweigh the cost of serialization. Concerning Ser-Off-Opt, we expect Clone-Off-Opt to be strictly superior to Ser-Off-Opt as Clone-Off-Opt avoids constructing a temporary buffer. Additionally, for both comparisons, we expect to see a noticeable increase in speedup numbers as $n \cdot E$ exceeds the L2 cache size, i.e., 64 KiB. Then, a large part of the object graph is already in off-chip memory and not in the sender's cache. Hence, most of the write-back operation issued by Clone-Off-Opt are no-operations, whereas the serialization-based approaches must construct an expensive temporary buffer.

Table 4.3 shows the speedup of Clone-Off-Opt over Ser-Tlm on Hw-Default. Suprisingly, we see that Clone-Off-Opt is always at least as fast as Ser-Tlm and provides speedups of up to 8.39×. There is no object graph size where copying to TLM and using a DMA transfer is worthwhile.

Table 4.4 shows the speedup of Clone-Off-Opt over Ser-Off-Opt on Hw-Default. Again, Clone-Off-Opt is always at least as fast as Ser-Off-Opt and provides speedups of up to 7.45×.

In both comparisons, speedups increase with increasing element size and increasing element count. Interestingly, if the object graph consists of many small elements, Clone-Off-Opt provides little or no benefit over Ser-Tlm or Ser-Off-Opt. We suspect that here the overhead for traversing the object graph, which is needed in all approaches, dominates and whether we serialize or clone the data has little influence on the running time.

For object graphs that are significantly larger than the cache size we observe high speedups compared to serialization-based approaches. In these cases, serializing the object graph into a buffer puts heavy load on the memory subsystem, which is avoided by cloning. However, we notice a significant speedup increase at a total size of 2^{19} bytes = 512 KiB. This is eight times the size of our L2 cache, significantly higher than we expected. We do not have a convincing theory that explains this observation.

On Hw-Delay, we expect the situation to be not as clear. Here, using TLMs offers a real benefit as accessing TLM is now significantly faster than accessing off-chip memory. Hence, we expect Ser-Tlm to outperform Clone-Off-Opt at least for small object graphs. For larger object graphs, where some or most of the data is already contained in off-chip memory, Clone-Off-Opt should still be faster. The result of the comparison with

	Element size E (in bytes)						
n	2^{6}	2^{7}	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}
2^{0}	1.77×	1.71×	1.67×	1.56×	1.55×	1.49×	1.54×
2^{1}	1.61×	1.59×	1.57×	1.53×	1.50×	1.53×	1.85×
2^{2}	1.50×	1.54×	1.49×	1.47×	1.59×	1.84×	1.88×
2^{3}	1.43×	1.39×	1.45×	1.54×	1.80×	1.91×	2.18×
2^{4}	1.17×	1.29×	1.46×	1.67×	1.82×	2.14×	2.57×
2^{5}	1.10×	1.34×	1.46×	1.68×	2.00×	2.48×	2.79×
2^{6}	1.13×	1.34×	1.53×	1.86×	2.31×	2.67×	2.86×
2^{7}	1.08×	1.33×	1.63×	2.09×	2.48×	2.72×	6.63×
2^{8}	1.12×	1.41×	1.80×	2.19×	2.52×	6.15×	8.39×

Table 4.3: Speedup of Clone-Off-Opt over Ser-Tlm for individual data transfers on Hw-Default. We copy a circular doubly linked list with n elements of size E.

	Element size E (in bytes)						
n	2^{6}	2^{7}	2^{8}	2^{9}	2^{10}	2^{11}	2^{12}
2^{0}	1.32×	1.33×	1.34×	1.35×	1.39×	1.39×	1.40×
2^{1}	1.28×	1.30×	1.36×	1.38×	1.45×	1.42×	1.45×
2^{2}	1.26×	1.33×	1.36×	1.39×	1.40×	1.47×	1.52×
2^{3}	1.25×	1.31×	1.37×	1.38×	1.45×	1.51×	1.58×
2^{4}	1.13×	1.21×	1.31×	1.30×	1.44×	1.57×	1.77×
2^{5}	1.05×	1.22×	1.27×	1.36×	1.54×	1.73×	1.86×
2^{6}	1.01×	1.17×	1.30×	1.47×	1.68×	1.78×	1.84×
2^{7}	1.03×	1.16×	1.33×	1.54×	1.69×	1.77×	5.62×
2^{8}	1.04×	1.19×	1.36×	1.54×	1.70×	5.20×	7.45×

Table 4.4: Speedup of Clone-Off-Opt over Ser-Off-Opt for individual data transfers on Hw-Default. We copy a circular doubly linked list with n elements of size E.

Ser-Off-Opt is difficult to predict. Both approaches use off-chip memory to transfer data. The only difference is that Ser-Off-Opt writes a single contiguous block to memory, whereas Clone-Off-Opt writes (potentially many) smaller objects.

Table 4.5 shows the speedup of Clone-Off-Opt over Ser-Tlm on Hw-Delay. As expected, the situation is not as clear as before. For small object graphs of size less than $2^{14} = 16\,\text{KiB}$, Ser-Tlm is faster than Clone-Off-Opt by roughly a factor of 2. We observe the highest speedup of Ser-Tlm (or lowest speedup for Clone-Off-Opt) for $n = 32$ and $E = 64$. Here, the latency and bandwidth advantages of TLM seem to be most pronounced. For large object graphs, Clone-Off-Opt is still significantly faster than Ser-Tlm and provides speedups of up to 8.36×.

Table 4.6 shows the speedup of Clone-Off-Opt over Ser-Off-Opt on Hw-Delay. Now, Ser-Off-Opt is sometimes a bit faster than Clone-Off-Opt but not by a large margin. We again observe the highest speedup of Ser-Off-Opt (or lowest speedup for Clone-Off-Opt) for $n = 32$ and $E = 64$. We suspect that the speedup of Ser-Off-Opt relative to Clone-Off-Opt is due to more predictable access behavior on the receiver. For Ser-Off-Opt, the receiver reads the contiguous serialized representation from off-chip memory. Hence, as the access behavior is predictable, many load instructions are cache hits. In contrast, Clone-Off-Opt traverses the object graph on the receiver. The objects may be scattered across off-chip memory, which may lead to a higher number of cache misses. For large object graphs, Clone-Off-Opt significantly outperforms Ser-Off-Opt and provides speedups of up to 7.54×.

Again, in both comparisons speedups increase with increasing element size and increasing total data size. The significant speedup increase is again at a total size of 2^{19} bytes = 512 KiB, for which we cannot offer an explanation.

Our results show that, in general, cloning is beneficial for large object graphs. If the speed difference between TLM and off-chip memory is large enough, we observe a break-even point P. For object graphs larger than this P, cloning is the fastest approach. For object graphs smaller than P, exploiting TLM offers a performance advantage.

	Element size E (in bytes)						
n	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
2^0	0.77×	0.68×	0.64×	0.47×	0.53×	0.47×	0.48×
2^1	0.80×	0.60×	0.60×	0.50×	0.50×	0.52×	0.66×
2^2	0.62×	0.56×	0.51×	0.44×	0.52×	0.64×	0.76×
2^3	0.58×	0.42×	0.55×	0.53×	0.63×	0.79×	1.30×
2^4	0.40×	0.47×	0.55×	0.65×	0.79×	1.28×	2.07×
2^5	0.37×	0.57×	0.55×	0.81×	1.30×	2.10×	2.61×
2^6	0.47×	0.74×	0.91×	1.32×	2.04×	2.53×	2.65×
2^7	0.75×	0.96×	1.38×	1.98×	2.46×	2.69×	6.57×
2^8	0.94×	1.40×	1.93×	2.30×	2.58×	6.34×	8.36×

Table 4.5: Speedup of Clone-Off-Opt over Ser-Tlm for individual data transfers on Hw-Delay. We copy a circular doubly linked list with n elements of size E.

	Element size E (in bytes)						
n	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}
2^0	0.98×	0.93×	0.93×	1.00×	0.98×	1.06×	0.92×
2^1	1.05×	1.00×	0.95×	0.92×	0.96×	0.95×	0.99×
2^2	0.94×	0.90×	0.92×	0.97×	0.98×	1.06×	1.18×
2^3	0.83×	0.93×	0.99×	0.93×	1.05×	1.17×	1.42×
2^4	0.85×	0.88×	1.02×	1.04×	1.14×	1.41×	1.63×
2^5	0.75×	1.01×	1.00×	1.14×	1.34×	1.62×	1.74×
2^6	1.09×	1.08×	1.18×	1.34×	1.60×	1.77×	1.85×
2^7	0.94×	1.17×	1.39×	1.56×	1.71×	1.82×	5.64×
2^8	1.11×	1.34×	1.53×	1.68×	1.76×	5.46×	7.54×

Table 4.6: Speedup of Clone-Off-Opt over Ser-Off-Opt for individual data transfers on Hw-Delay. We copy a circular doubly linked list with n elements of size E.

4.6.4.2. Distributed Kernel Benchmarks

We now compare the running times of X10 applications using Ser-Tlm, Ser-Off-Opt, and Clone-Off-Opt. We use the X10 programs from the IMSuite benchmark suite [GN15] as our test inputs. IMSuite consists of 12 programs that implement popular, mostly graph-based distributed algorithm kernels. More specifically, the programs are:

- BF, an implementation of the Bellman-Ford algorithm;
- DST, which computes shortest routes according to Dijktra's method;
- BY, a solver for the Byzantine generals' problem;
- DR, which computes a routing table for a graph;
- DS, which finds a dominating set;
- KC, which partitions the nodes of a network into committees of size at most k;
- MIS, which computes a maximal independent set of a set of nodes;
- LCR, HS, DP, which all implement leader election algorithms with different graph constraints;
- MST, which computes a minimum spanning tree; and
- VC, which colors the nodes of a tree with three colors.

Being distributed in nature means that, when run on an invasive architecture, the programs must communicate between tiles. Hence, they are a good fit for assessing data-transfer performance. The sizes of the test programs range from 300 loc to 1000 loc.

We use the iterative X10-FA configuration of the benchmark programs with the input data set of size 64. We use the running time measurement infrastructure already present in the programs. We modified the programs so that they contain their input data as our prototype platform does not provide a file system. Input data is read during the initialization phase, which is not included in the running time measurements.

IMSuite contains implementations of each algorithm in two languages: X10 and Habanero Java [Cav+11]. Habanero Java extends Java with features very similar to those found in X10. As both languages aim to increase programmer productivity, the IMSuite authors decided against writing highly tuned implementations of the respective algorithms. Instead, IMSuite intentionally contains rather straightforward algorithm translations to the two target languages.

Unfortunately, the X10 implementations contain multiple instances of a common pitfall that can lead to serious performance degradations if the programs are executed on multiple places. The pitfall, also explicitly mentioned in the X10 language specification [Sar+16, §13.3.7], may cause X10's **at** construct to copy more values than necessary (cf. Section 4.4.4). To understand the problem, let us look at the following X10 program.

```
 1 class Foo {
 2   val large = new Large();
 3   val x = 42;
 4   public def get() { return x; }
 5
 6   public def test(p: Place) {
 7     at (p) x;
 8     at (p) get();
 9   }
10 }
```

Here, the method `test` uses **at** to evaluate the expressions `x` (in line 7) and `get()` (in line 8) on place `p`. At first glance, it seems that both **at** expressions (in lines 7 and 8) only capture the field `x`. Thus, both should be relatively lightweight operations.

However, both **at** expressions actually capture the implicit **this** reference as `x` is a field and `get()` is a non-static method. Therefore, both **at** expressions lead to the transfer of all objects transitively reachable from **this**. This includes the `Large` object referenced by the `large` field, which is potentially costly to transfer.

The X10 developers are aware of this pitfall [Sar+16, §13.3.7] and have proposed multiple possible solutions[34]. These proposals include "copy specifiers" that allow programmers to specify the variables they expect to be captured (which enables the compiler to warn if more variables

[34] See also X10 issue reports `https://xtenlang.atlassian.net/browse/XTENLANG-1913` and `https://xtenlang.atlassian.net/browse/XTENLANG-2466`.

are captured than expected), and allowing capturing individual fields (without also capturing the reference to their enclosing object).

However, at the time of writing, the X10 compiler does not implement any of these techniques. Hence, the pragmatic solution to avoid accidentally capturing too many variables is to manually apply the following two rewrite rules, which we describe in a semi-formal style.

1. For each field reference of the form `o.f` (where `o` can be an implicit **`this`** reference) inside an **`at`** block A, introduce a final local variable `f'` in A's enclosing scope, initialize `f'` with `o.f`, and replace all occurrences of `o.f` in A with `f'`. Hence, in the above example, rewrite

   ```
   at (p) x;
   ```

   to

   ```
   val x' = x;
   at (p) x';
   ```

2. For each non-static method of the form `m`$(p_1, \ldots, p_n)$ B (with parameters p_i and method body B), which is called inside an **`at`** block A, add a static copy `m'`$(p_1, \ldots, p_n, p'_{n+1}, \ldots, p'_k)$ B' of `m` in `m`'s scope. For each field that is referenced in the form **`this`**`.f` in B, add a parameter p'_j to the parameter list of `m'`, replace each occurrence of **`this`**`.f` in B' with p'_j, and add `o.f` to the argument list of each call to `o.m` in A. Finally, replace all calls to `m` in A with calls to `m'`. We must now rewrite the newly added method arguments of `o.m` according to rule 1.

   Hence, in the above example, we rewrite

   ```
   public def get() { return x; }
   public def test() {
     at (p) get();
   }
   ```

   to

```
  public def get() { return x; }
  public static def get'(x: Int) { return x; }
  public def test() {
    val x' = x;
    at (p) get'(x');
  }
```

As our proposed optimizations from Section 4.3 concern the **at** construct, we need to ensure a realistic usage pattern of this construct in our benchmarks. If the benchmarks spent an unrealistic portion of their running time performing data transfers, the impact of our optimization on an average program would be overestimated. Therefore, we manually adapted the programs from IMSuite using the presented rewrite rules. Figure 4.20 shows that the changes are purely mechanical.

We sent our adapted benchmark programs to the IMSuite authors, who acknowledged the problem, agreed that our fixes are correct, and stated that they planned to release a new version of IMSuite. However, at the time of writing of this dissertation, no new testsuite version has been released yet. We provide our adapted program versions as part of the software artifacts described in Appendix B. In the following, we always use the adapted benchmark programs[35].

Table 4.7 shows statistics about the object graphs that we observe during a full run of our IMSuite benchmark programs. We instrumented our runtime system to collect these numbers. We see that the benchmarks have distinct communication patterns. Some communicate little, e.g., BF only transfers a total of about 81 KiB between tiles, while others send more data, e.g., MST transfers more than 44 MiB. The same holds for the number of object graphs, which varies from few (1151 in the case of BF) to many (625104 for BY).

Interestingly, the average number of vertices per object graph does not vary that much and is roughly within the same order of magnitude (between about 3 and 13) across all benchmarks. However, the average size of object graphs differs significantly. Some benchmarks, such as DS, transfer lots of

[35] We used Git revision `100264ac8cbac654e6f57358bb13e654501f00cd`.

```
static def loadweight(weight: Long) {
  ... loadValue ...
}
def bfsForm() {
  finish for (i in D) async
    at (D(i)) {
      for (var j: Int = 0; j < nodeSet(i).tMH.size(); j++)
        nodeSet(i).mH.add(nodeSet(i).tMH.get(j));
      nodeSet(i).tMH.clear();
      if (loadValue != 0)
        nval(i) = loadweight(nval(i) + i(0));
    }
  // ...
}
```

(a) Original program code.

```
static def myloadweight(weight: Long, lv: Long) { ... }
def bfsForm() {
  finish for (i in D) async {
    val mynodeSet = nodeSet;
    val myloadValue = loadValue;
    val mynval = nval;
    at (D(i)) {
      for (var j: Int = 0; j < mynodeSet(i).tMH.size(); j++)
        mynodeSet(i).mH.add(mynodeSet(i).tMH.get(j));
      mynodeSet(i).tMH.clear();
      if (myloadValue != 0)
        mynval(i) = myloadweight(mynval(i) + i(0), myloadValue);
    }
  }
  // ...
}
```

(b) Adapted program code.

Figure 4.20: Excerpts from inner loop of benchmark program `bfsBellmanFord` before (top) and after (bottom) changes. The code has been reformatted and some identifiers have been shortened to improve readability.

	Benchmark					
	BF	DST	BY	DR	DS	MIS
Σ #object graphs	1 151	4 502	625 104	28 624	68 052	4 648
Σ #objects	7 486	29 322	3 141 072	96 696	229 992	28 757
ø #objects	6.50	6.51	5.02	3.38	3.38	6.19
Σ sizes (in bytes)	81 191	3 507 332	37 636 416	3 739 712	2 285 971	313 244
ø size (in bytes)	70.54	779.06	60.21	130.65	33.59	67.39

	Benchmark					
	KC	DP	HS	LCR	MST	VC
Σ #object graphs	31 107	22 558	25 858	12 545	32 034	1 303
Σ #objects	165 327	133 297	326 186	99 333	201 665	11 011
ø #objects	5.31	5.91	12.61	7.92	6.30	8.45
Σ sizes (in bytes)	2 000 704	2 585 623	3 088 640	1 055 796	44 124 424	114 340
ø size (in bytes)	64.32	114.62	119.45	84.16	1 377.42	87.75

Table 4.7: Object-graph properties from all programs in our test suite. We acquired all numbers through instrumentation of the runtime system during a full run of the IMSuite programs using the same input data as for our running time measurements. We list the total number of graphs, the total number of contained objects, the average number of objects per graph, the total size of all transferred objects, and the average size of the objects per graph.

small object graphs (about 32 bytes on average). Others, such as DST or MST, transfer larger object graphs in the order of 1 KiB on average.

From these numbers, we would suspect DST and MST to have the largest speedup potential as they transfer the largest object graphs. As an experiment, we take Tables 4.3 to 4.6 as "lookup tables" for the speedups we can expect in the best case and use the numbers from Table 4.7 as indices. Regarding indices, the closest match for DST (779 B in 6.51 objects, i.e., about 119 B per object) is $n = 2^3$ and $E = 2^7$. For MST, the closest match is $n = 2^3$ and $E = 2^{10}$. We retrieve the following best case speedups:

- On Hw-Default relative to Ser-Tlm: 1.39× for DST, and 1.80× for MST;
- On Hw-Default relative to Ser-Off-Opt: 1.31× for DST, and 1.45× for MST;
- On Hw-Delay relative to Ser-Tlm: 0.42× for DST, and 0.63× for MST;
- On Hw-Delay relative to Ser-Off-Opt: 0.93× for DST, and 1.05× for MST.

The upper three rows of Table 4.8 show the running times of all benchmarks on Hw-Default for the three tested variants Ser-Tlm, Ser-Off-Opt, and Clone-Off-Opt. First, we see clear differences in the running times between the three variants, which means that due to their distributed nature, the benchmarks spend a significant portion of their running time on communication. This supports our case that efficient data transfers are crucial for application performance on invasive architectures.

The middle two rows show the speedup of Clone-Off-Opt compared to Ser-Tlm and Ser-Off-Opt. On average, Clone-Off-Opt provides a speedup of 1.17× compared to Ser-Tlm. Compared to Ser-Off-Opt, Clone-Off-Opt achieves an average speedup of 1.05×. For every test case, Clone-Off-Opt is at least as fast as Ser-Tlm or Ser-Off-Opt.

We expected the highest speedups for DST and MST. Our running time measurements confirm our supicion. For example, for MST, compared to Ser-Off-Opt, the observed speedup of 1.24× is suprisingly close to the best-case speedup of 1.45× we approximated before. In general, speedups are somewhat lower due to interferences with program behavior unrelated to data transfers. For example, the invalidation of the complete L1 cache may

	Benchmark						
	BF	DST	BY	DR	DS	MIS	Geomean
Ser-Tlm	1.30	9.35	736.79	83.22	50.92	1.75	
Ser-Off-Opt	1.17	7.94	677.27	82.13	47.24	1.60	
Clone-Off-Opt	1.13	7.35	658.39	80.42	45.49	1.57	
Speedup$_{\text{Ser-Tlm}}$	1.15×	1.27×	1.12×	1.03×	1.12×	1.12×	
Speedup$_{\text{Ser-Off-Opt}}$	1.03×	1.08×	1.03×	1.02×	1.04×	1.02×	
Reduction$_{\text{Ser-Tlm}}$	33.7%	57.6%	28.2%	22.5%	22.0%	33.7%	
Reduction$_{\text{Ser-Off-Opt}}$	9.7%	28.4%	8.6%	15.0%	8.3%	7.7%	

	Benchmark						
	KC	DP	HS	LCR	MST	VC	Geomean
Ser-Tlm	27.10	36.59	43.86	14.24	69.82	1.60	
Ser-Off-Opt	25.86	34.14	34.81	11.92	62.87	1.30	
Clone-Off-Opt	25.84	32.61	34.00	11.88	50.70	1.26	
Speedup$_{\text{Ser-Tlm}}$	1.05×	1.12×	1.29×	1.20×	1.38×	1.27×	1.17×
Speedup$_{\text{Ser-Off-Opt}}$	1.00×	1.05×	1.02×	1.00×	1.24×	1.03×	1.05×
Reduction$_{\text{Ser-Tlm}}$	12.5%	35.2%	56.2%	49.0%	50.9%	50.1%	34.5%
Reduction$_{\text{Ser-Off-Opt}}$	0.3%	17.3%	9.9%	1.8%	39.8%	9.5%	8.1%

Table 4.8: Running and communication time for all test programs from IMSuite on Hw-Default. Upper rows: Running times (in seconds) for each of the three variants Ser-Tlm, Ser-Off-Opt, and Clone-Off-Opt. Middle and lower rows: Overall speedups and reduction of communication time of Clone-Off-Opt over Ser-Tlm and Ser-Off-Opt.

negatively affect following load operations if they cannot be compensated by the L2 cache.

In general, we see that exploiting off-chip memory for data transfers is beneficial on Hw-Default: for most benchmarks, there is a large gap between Ser-Tlm and the other two variants as Ser-Tlm transfers data via TLMs. We suspect that this is at least partly due to the current requirement of our DMA units of copying data to TLM instead of going directly to off-chip memory.

The lower two rows of the table show the reduction of the time spent on communication of Clone-Off-Opt over Ser-Tlm and Ser-Off-Opt. We instrumented our runtime system to determine the time spent on communication. To this end, we employ a global timestamp mechanism provided by the NoC. On average, Clone-Off-Opt provides a 34.5% reduction in communication time relative to Ser-Tlm. Compared to Ser-Off-Opt, Clone-Off-Opt achieves an average communication time reduction of 8.1%.

Table 4.9 shows the running times of all programs on Hw-Delay. We reduced to number of iterations for BY to 10 due to its high running time. In general, running times are significantly higher (by roughly a factor of 5×) compared to Hw-Default due to increased access latency and decreased bandwidth to off-chip memory. We observe an interesting effect: for some benchmarks, Ser-Tlm is now the fastest approach. This shows how important it is to evaluate data-transfer techniques with different memory parameters.

If we take a closer look at the programs for which Ser-Tlm is the fastest, we notice that all these programs transfer comparatively small object graphs. For such graphs, serialization is relatively cheap. In the best case, we never have to access off-chip memory during the whole transfer. Here, the object graph is still in a local cache, gets serialized and copied into the local TLM, transferred to the receiver's TLM via a DMA transfer, where the receiver then deserializes the object graph and again holds it in a local cache. On Hw-Default, this does not provide a significant advantage as TLM and off-chip memory are about equally fast.

For programs that transfer comparatively large object graphs, such as DST and MST, Clone-Off-Opt is still superior. In this case, serialization is

	Benchmark						
	BF	DST	BY	DR	DS	MIS	Geomean
Ser-Tlm	4.27	42.14	1 995.26	128.05	115.81	14.95	
Ser-Off-Opt	4.36	43.91	2 041.17	132.99	121.78	15.21	
Clone-Off-Opt	4.33	39.71	2 085.32	128.93	121.95	14.98	
Speedup$_{\text{Ser-Tlm}}$	0.99×	1.06×	0.96×	0.99×	0.95×	1.00×	
Speedup$_{\text{Ser-Off-Opt}}$	1.01×	1.11×	0.98×	1.03×	1.00×	1.01×	

	Benchmark						
	KC	DP	HS	LCR	MST	VC	Geomean
Ser-Tlm	104.47	120.52	114.67	36.70	280.31	3.93	
Ser-Off-Opt	106.88	122.75	118.73	37.79	286.57	4.01	
Clone-Off-Opt	108.01	121.22	107.81	36.03	233.65	3.85	
Speedup$_{\text{Ser-Tlm}}$	0.97×	0.99×	1.06×	1.02×	1.20×	1.02×	1.02×
Speedup$_{\text{Ser-Off-Opt}}$	0.99×	1.01×	1.10×	1.05×	1.23×	1.04×	1.04×

Table 4.9: Running times for all test programs from IMSuite on Hw-Delay. Upper three rows: Running times (in seconds) for each of the three variants Ser-Tlm, Ser-Off-Opt, and Clone-Off-Opt. Lower two rows: Overall speedups of Clone-Off-Opt over Ser-Tlm and Ser-Off-Opt.

so expensive that avoiding it is worth more frequent accesses to off-chip memory. The speedups of Clone-Off-Opt compared to Ser-Tlm are lower than on Hw-Default. This shows that the larger the gap of access speeds between off-chip and on-chip memory, the better it is to exploit TLM for transferring data.

Interestingly, Clone-Off-Opt is faster than Ser-Tlm for DST and MST although our previous considerations suggested otherwise. We suspect that this is due to Clone-Off-Opt being more cache-friendly. Our synthetic benchmark program from Section 4.6.4.1 only measured individual data transfers and did not consider the effect of cache pollution on further program execution. This effect might be what causes Clone-Off-Opt to be faster in this scenario.

In general, Clone-Off-Opt trades the need to serialize for more frequent off-chip memory accesses. On Hw-Default, where accessing off-chip memory is extremely cheap, this trade is always beneficial. As TLM and off-chip memory are almost equal in terms of latency and bandwidth, Clone-Off-Opt practically avoids some copies without increasing cost. Hence, it is not surprising that Clone-Off-Opt is strictly superior to Ser-Tlm. The situation changes on Hw-Delay. Now, more frequent copies can be worth avoiding costly accesses to off-chip memory. Then, depending on the size of the object graphs, Ser-Tlm can be faster than Clone-Off-Opt.

4.6.5. Hardware Overhead

Tradowsky et al. implemented our proposed range operations as an extension to the cache controller of the Gaisler LEON3 processor [Cob17b]. Table 4.10 shows that compared to the unmodified cache controller, about 15% of additional logic is necessary to implement non-blocking range operations with one range buffer on the Xilinx XUPV5 Virtex-5 FPGA.

Table 4.11 shows the overhead of implementing blocking range operations compared to the non-blocking variant. The numbers differ from Table 4.10 as Tradowsky et al. used a more recent version of our modified cache controller. Interestingly, adding range operations in the first place is far more expensive than making them additionally non-blocking. Making the operations non-blocking causes a slight increase of the number of slices

	Additional resources	
	absolute	relative
Slices	1489	15.2%
Register	623	14.6%
LUTs	1491	15.0%
BRAM	1	4.9%

Table 4.10: Additional resources for the implementation of non-blocking range operations compared to original cache controller.

	Blocking	Non-blocking	Relative change
Slices	680	734	+7.9%
Registers	775	775	0.0%
LUTs	1688	1672	−0.9%
BRAMs	0	0	0.0%

Table 4.11: Additional resources used for blocking range operations compared to non-blocking range operations. Numbers differ from Table 4.10 as a more recent version of the modified cache controller was used.

used, while the number of LUTs even decreases. We suspect that this decrease is due to heuristic optimizations that happen during hardware synthesis. The expensive part of adding support for range operations is the required state machine that implements the loop over all relevant cache lines, which is required independent of whether the operations are blocking or not.

As explained in Section 4.5, our implementation needs at most n spare cycles to execute a range operation on a range spanning n cache lines. We instrumented the programs from IMSuite and found that the average object graph size is 257.3 bytes. On our system, the minimum cache line size is 16 bytes. Hence, there must be at least 17 spare cycles between two range operations to avoid blocking. Analysis of the generated code for our cloning approach showed that this is fulfilled. For both write-back and the cloning operation itself, we use a resizable hash set to detect cycles in the object graph. Operating on the hash set involves enough arithmetic and control flow instructions to hide the range operation's latency. Therefore, executing a range operation during Clone-Off-Opt takes one cycle from the view of the processor for the average object graph.

So, is it worth the effort? Let us take the L2 cache of our prototype system as a concrete example. This cache offers means to invalidate a single cache line, identified by an address, but does not have the extensions that we proposed in Section 4.5. Hence, to invalidate an address range, the processor has to execute a software loop and issue one invalidation per relevant cache line. The loop boils down to an addition, a store (which triggers the invalidation via a memory-mapped register), a comparison, and a conditional branch. Additionally, the cache takes five clock cycles to process each invalidation [Cob16, section 74.3.3]. Hence, each invalidation takes roughly 10 cycles.

With an average object graph size of 17 cache lines, our loop takes in the order of 200 clock cycles, which we can reduce to 1 (from the view of the processor) in the best case using the non-blocking range operation extension. This sounds like an impressive speedup. However, compared to the latency of the memory accesses following the invalidation, this difference is negligible, as fetching a single cache line can easily take hundreds of cycles, depending on the structure of the memory subsystem.

Hence, we conclude that while adding range operations is feasible, it is not worth the additional hardware cost. The literature that requests such instructions (cf. Section 4.5) probably did not consider this cost. We agree that more fine-grained cache control is crucial for performance on non-cache-coherent architectures to support software-managed coherence. However, means to invalidate or write back individual cache lines are sufficient and cheap to implement. All remaining functionality should be implemented in software. However, our findings may be useful in the implementation of remote invalidation operations, which we discuss in Appendix A.1.

4.6.6. Threats to Validity

In this section, we try to list all limitations of our experiments as well as decisions that may have influenced our results.

The most important limitation of our prototype is that the difference concerning latency and bandwidth of TLMs compared to off-chip memory is far less pronounced than on real systems. We tried to alleviate this limitation by using a hardware variant with artificially increased DRAM latency. However, this does not capture all differences between our prototype and a real chip. Hence, behavior on a real ASIC with realistic clock frequencies and latencies may be significantly different.

Our prototype architecture is compute-bound. Our cores only run at 25 MHz, while our SRAM and DRAM are disproportionately faster. This is not realistic, as on a real chip this relationship would be reversed: the memory would be much slower than the cores. While this effect is not as pronounced on Hw-Delay as on Hw-Default, it may still have led us to overstate the cost of serialization. On a real chip it may thus not be worthwhile to use cloning instead of a serialization-based approach or it may only be worthwhile for object graphs of a certain minimum size, as we observed on Hw-Delay. On Hw-Default, such a break-even point is not measurable: our numbers show that cloning is always at least as fast as serialization-based approaches.

We did not reproduce the state of the art for message passing techniques using on-chip memories. Almost all existing work has been done in the context of the Intel SCC. As our hardware is sufficiently different, we could not reuse that work directly. Due to time constraints, we did not port existing projects to our hardware platform. Additionally, differences between on-chip and off-chip memory are hard to emulate on an FPGA (see above), thus porting these approaches is also problematic conceptually. Hence, it is possible that our numbers acquired for Ser-Tlm are not representative and thus must be interpreted carefully.

Our TLMs are too large compared to other system parameters. A size of 8 MiB per tile is unrealistic, e.g., the Intel SCC has 16 KiB of on-chip memory per tile, which contains two cores. This does not affect the comparison of Ser-Off-Opt with Clone-Off-Opt, as they do not use TLMs to transfer data. It gives Ser-Tlm an unfair advantage, as it does not need to split messages because every message fits completely into the TLM.

We did not integrate our hardware extension into the platform on which we performed our benchmarks. Hence, we did not have fine-grained control over the L1 cache; we could only flush it completely. This affects both Ser-Off-Opt and Clone-Off-Opt. Hence, their absolute running times are skewed; however, the relative comparison of the two approaches is fair.

Our proposed pull-style cloning approach is not one-sided as the receiver actively partakes in the data transfer. In contrast, serialization-based approaches are easy to implement in a one-sided fashion. In conjunction with possibly lower costs for serialization (see above) this could lead to serialization-based approaches being superior to cloning. For example, assume that our architecture was memory-bound, had larger caches, and our DMA units supported transfer to off-chip memory. Then, it is possible that serializing an object graph and then copying it using a DMA transfer would be more efficient.

We use only one I/O tile with external memory, hence our system does not have distributed off-chip memory. Additionally, our synthesized chip is quite small and every tile has a low maximum distance to off-chip memory of two hops.

Our cores have write-through L1 caches, which avoids some difficult problems. Suppose we had write-back L1 caches and core 1 wanted to write back a block of memory B. It is now possible that part of B is held in the L1 cache of a different core, say core 2. Hence, we either synchronize all cores and force the software to execute the necessary writeback operation on all cores of a tile. Or, we add hardware support for writing back or invalidating cache lines in caches other than the one of the current core. This could, for example, be realized by letting all cache controllers snoop such operations and execute them on their respective caches; see Appendix A.1 for details.

We do not support custom serialization formats in our cloning approach. X10 allows types to implement the interface `x10.io.CustomSerialization`. This signifies that these types do not use the default serialization methods generated by the compiler but specify their own custom serialization format. Supporting custom serializable types in the cloning approach is cumbersome. Our cloning approach hinges on the fact that all accesses to foreign memory partitions happen under control of the compiler. Hence, the necessary write-backs and invalidations can be inserted automatically. If we look at custom serialization formats, its counterpart in the cloning approach would be a user-implemented clone function. Hence, user-controlled code would need to access objects from foreign memory partitions, which requires a write-back on the sending side and a preceding invalidation on the receiving side. We see no other possibility than letting the user issue these cache operations, which is error-prone.

We did not consider cache architectures with more than two levels. If we assume a cache hierarchy consisting of L1, L2, and L3 cache, which is common on current processors, invalidations and write-backs must be effective on all cache levels. This can influence the cost and complexity of the necessary hardware support.

We do not evaluate the power impact of our proposed cache extension. Our extension causes the cache to become active during more cycles; specifically during spare cycles. This causes an increased power usage, which could make our extension unattractive. However, if an address range must be invalidated or written back, some component has to do this work. If the hardware does not support it directly, a general-purpose processor must take over, which probably requires significantly more power. Hence, depending on the frequency of range operations, a hardware-accelerated implementation could even lower power usage.

4.7. Relation to Invasive X10

So far, we have run standard X10 programs on our platform. However, as explained in Section 3.1, Invasive Computing also proposes a new programming paradigm. In the following, we first explain how we integrated this paradigm into X10. We call the resulting extended language Invasive X10 [Bra+14] to differentiate it from regular X10. Subsequently, we argue that our work on data transfers from the previous sections is especially important in the context of Invasive X10.

Invasive X10. The invasive paradigm focuses on exclusive resource allocation. Resources are partitioned into claims. Invasive applications can create (“invade”), resize (“reinvade”), use (“infect”), and destroy (“retreat”) claims.

When trying to integrate the idea of claims with existing X10 language semantics, one quickly notices the following problem. Suppose we run an invasive X10 application on an invasive architecture with four tiles. Assume that, initially, the application only has one claim containing cores on tile 0. How many X10 places does this application see if it queries, e.g., the number of places?

One possibility would be that it sees four places (as we represent each tile by one place). In regular X10, the number of places is fixed during a program run [Sar+16, section 13]. If we want to keep this rule, the answer to our question from before must be four; otherwise, we would never be able to use more than one tile. Hence, all places would be visible, but only some places should be usable for the application. Thus, if an application uses places not currently contained in its claim, we would need to report some kind of error, e.g., by throwing an exception. We could then provide means to query whether a place is contained in a given claim to enable the programmer to prevent these errors.

This basically adds the notion of “allowed” and “disallowed” places to X10. As existing X10 code does not know about this distinction, it has to be adapted. For example, all existing X10 standard library code that deals with multiple places must be changed. Thus, we conclude that this design is feasible in theory, but inelegant and impractical.

Instead, we propose to lift the concept of claims to a level above the concept of places. Now, regular X10 lives *inside* a claim: an X10 application only sees multiple places if the claim it currently runs in contains cores from multiple tiles. In general, the application’s view is restricted to the resources that are contained in its claim. Hence, in this design the answer to our question from before is one.

This design breaks X10’s rule that the number of places is fixed. Suppose a program changes its claim. It may acquire additional cores on a different tile, which we must, according to our new design, make available as a new place. Hence, the number of places may increase. Conversely, the application may free resources so that the number of places decreases.

In general, the number of places is now dynamic, and we call the resulting extended X10 language Invasive X10. Invasive X10 requires changes to the runtime system. For example, we have to convert the static field `Place.NUM_PLACES` to a method `Place.numPlaces()`; for details see [Bra+14].

The programmer deals with claims analogously to places. Hence, each claim is represented as a regular X10 object of type `Claim`, just like each place is an object of type `Place`. The class `Claim` offers a static method `invade()` to create new claims, and non-static methods `infect()`, `reinvade()`, and `retreat()` to operate on existing claims. Just like `at` allows changing between places, `infect` allows changing between claims.

From the viewpoint of an application running inside a claim, the regular X10 semantics hold. Therefore, regular X10 can be embedded naturally into Invasive X10: a regular X10 program behaves exactly like an Invasive X10 program that runs inside a claim containing all resources in the system. In fact, this is exactly how we performed all our experiments in Section 4.6. Here, we configured our X10 runtime system to create a single claim containing all system resources during initialization and destroyed it on program exit, never modifying it in the meantime. Hence, we could run regular unmodified X10 programs.

Data redistribution. If the number of places changes, the program has to adapt. As such resource changes only happen at well-defined program points, such as calls to `reinvade()`, the programmer does not have to deal with asynchronous events[36]. For details on how to handle appearing and disappearing places, see [Bra+14] and [Cun+14]. In the following, we focus on one aspect: data redistribution.

Suppose an invasive application has acquired additional cores and its claim has grown from cores on one tile to cores on three tiles. In order to exploit the processing power of the cores on the two new tiles, they

[36] Except for the special class of malleable applications, see [BMZ15] for details.

must be fed enough data. Hence, the application must redistribute its data from one place to three places.

As we have discussed before, the primary means for data transfers between places in X10 are `Array.asyncCopy()` and **`at`** for simple and complex data structures, respectively. We can accelerate both means to transfer data using the techniques we presented in Sections 4.3 and 4.4.

The invasive paradigm requires resource-aware applications that frequently adapt their resource needs. Otherwise, the system has little room to optimize resource usage for efficiency or predictability. In general, frequent resource changes induce frequent data transfers to redistribute data.

To support our argument, we perform the following experiment. We use an existing X10 application that exists in an invasive and a non-invasive variant. We use the multigrid [Bun+13] application[37]. This application stems from the high performance computing domain and is a numeric simulation of heat distribution on a metal plate. We instrumented the X10 runtime library to measure the amount of data transferred. We then ran the invasive and the non-invasive variant of the application with the same parameters[38] and determined the amount of data transferred.

We did not perform our experiments on the hardware prototype. Instead, we generated x86 code and used the `x86guest` variant of OctoPOS[39] , which runs as a guest operating system under Linux. The OctoPOS interface to applications and compiler is exactly the same as on the hardware prototype. As the sizes of all X10 data types are fixed and pointers have 32 bits on both x86 and SPARC, the number of transferred bytes is the same as on the hardware prototype. Using the x86 variant of OctoPOS allowed us to simulate three different invasive hardware configurations: a replica of the prototype with 4 tiles with 4 cores each; a variant with 6 tiles having 4 cores each; and a variant with 8 tiles having 6 cores each.

[37] We use `x10i` Git revision `31183335a89917f489046da746c5181174a7bdb3` and the multigrid application of Git revision `6bb6ef6ff5c260eb0391bd12b82f052184c3a097`.

[38] We set the number of simulated timesteps to 50 and used the defaults for all other parameters.

[39] We used Git revision `a0a23ef38fe2b6d7b9c9544a94c990cb2201ad57`.

	Non-invasive	Invasive	Relative change
4 tiles, 4 cores each	10 809 196	14 902 232	+37.9%
6 tiles, 4 cores each	17 851 708	24 872 572	+39.3%
8 tiles, 6 cores each	24 872 572	32 300 660	+29.9%

Table 4.12: Amount of data transferred (in bytes) during run of multigrid application. The table compares the amount for the non-invasive variant to the invasive variant on three different architecture configurations.

Table 4.12 shows the amount of data transferred (in bytes) during runs of the multigrid application and, for each simulated hardware design, compares this amount for the non-invasive to the invasive application variant. We see that due to resource adaptations and thus more frequent data redistribution, the invasive application transfers between 30% and 40% more data than the non-invasive variant. We suspect that the difference between variants with 4 cores per tile and the variant with 6 cores per tile is that with 6 cores per tile, more resource adaptations can happen inside the tiles without requiring inter-tile data transfers, hence the relative change is lower.

This experiment shows that efficient data transfers are even more important for invasive X10 programs than for regular X10 programs. Invasive X10 programs have been shown to improve resource efficiency and throughput [Bun+13]. However, in general, this comes at the price of more frequent data transfers, which should therefore be as efficient as possible.

Summary

- X10 maps naturally to invasive architectures by viewing each tile (i.e., coherence domain) as one place.
- Data transfers between off-chip memory partitions are central to the performance of X10 programs on invasive architectures.
- TLMs offer interesting trade-offs for implementing such data transfers.
- One-sided transfers of simple contiguous data structures should always proceed directly via off-chip memory.
- Transfers of complex pointered data structures should proceed via TLMs for small data structures, and via off-chip memory for larger data structures.
- Object cloning can be adapted to work in the presence of non-coherent caches through automatic compiler-directed software-managed coherence.
- It is worthwile to avoid serialization by using object cloning when transferring large pointered data structures.
- Means for coarse-grained cache control are important for the efficient execution of PGAS and message-passing programs.
- Implementing such cache functionality completely in hardware is feasible, but not worth the overhead.
- Data transfers are even more important for invasive programs than for regular programs due to more frequent data redistribution.

```
Values:  1 2 3 4 5
Values:  2 3 4 5 1
Values:  3 4 5 1 2
Values:  4 5 1 2 4
```

Output of a test program on an
early hardware prototype

5 Code Generation with Permutation Instructions

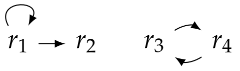
In this chapter, we investigate the use of permutation instructions to speed up the execution of shuffle code. We first present an instruction set extension and describe a possible hardware implementation. We then study different code-generation approaches and evaluate benefits and overheads. Parts of this chapter have been published in [Moh+13] and [BMR15b; BMR15a].

Motivation. During compilation of a program, register allocation is the task of mapping program variables to machine registers. During register allocation, the compiler often emits *shuffle code*, consisting of copy and swap operations, that transfers data between the registers. Three common sources of shuffle code are (i) conflicting register mappings at joins in the control flow of the program, e.g., due to if-statements or loops; (ii) the calling convention for procedures, which often dictates that input arguments or results must be placed in certain registers; and (iii) constrained machine instructions that only allow a subset of registers to occur as operands.

Figure 5.1 shows an example situation where the compiler needs to emit shuffle code. In Figure 5.1a, we assume that the compiler has mapped

```
a = 10; // in r1
b = 20; // in r3
c = 30; // in r4
...
foo(a, a, c, b);
```

(a) Source program.



(b) Necessary transfers.

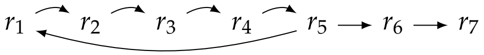
```
copy r1, r2
swap r3, r4
```

(c) An implementation.

Figure 5.1: Example of shuffle code. We assume that the platform requires passing function arguments in consecutive registers, i.e., the i-th argument in r_i.

variables a, b, and c to registers r_1, r_3, and r_4, respectively. Additionally, we assume that the calling convention requires us to put function arguments into consecutive registers, i.e., the first argument must go into r_1, the second into r_2, and so on. Hence, the compiler needs to emit code that shuffles register values before the function call to foo, so that the registers contain the values expected by the function.

Figure 5.1b illustrates the necessary value transfers using a *register-transfer graph*, which we study in more detail in Section 5.1.1. Every vertex represents a register and edges are transfer operations between registers. Semantically, all copy operations are supposed to happen in parallel. Here, r_1 needs to retain its value (hence the loop) but we also need to copy its value to r_2 as variable a is passed twice as an argument to foo. At the same time, we need to swap r_3 and r_4. Figure 5.1c shows a possible implementation to achieve the required data redistribution between registers. Depending on the quality of register allocation, such shuffle code may be frequent. Additionally, it may potentially involve many registers and thus may be expensive to implement. For example, the following register transfer graph is also possible, with a possible implementation shown on the right.



```
copy r6, r7   swap r4, r3
copy r5, r6   swap r3, r2
swap r5, r4   swap r2, r1
```

We now take a look at the hardware that executes this shuffle code. Modern processors often support out-of-order execution and rename

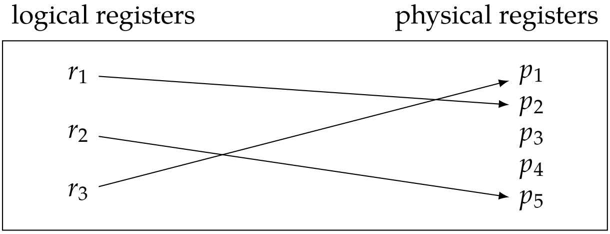


Figure 5.2: A register alias table. It holds the current mapping from logical registers r_i (visible in the instruction set) to physical registers p_j. There are usually more physical than logical registers.

registers to exploit instruction level parallelism. Such a processor executes instructions in an order consistent with the data dependencies between the instructions, but not necessarily in the program order. To remove so-called false dependencies, these processors employ register renaming.

```
i₁: r₁ ← add r₂, r₂
i₂: r₃ ← add r₁, r₁
i₃: r₁ ← add r₄, r₄
i₄: r₅ ← add r₁, r₁
```

For example, in this instruction sequence we have two instructions i_1, i_3 that do not require each other's computed results but write to the same destination register r_1. By renaming r_1 in i_3 (and in i_4) to some temporary register r_t, we can execute both i_1 and i_3 in parallel[40]. To this end, these processors have more physical registers than there are logical registers visible in the instruction set. Hence, we can choose a free physical register as our temporary register r_t.

To implement register renaming, the processor maintains a mapping from logical to physical registers. A popular way to implement this mapping is a *register alias table* (RAT). Figure 5.2 illustrates the concept. The RAT is indexed by a logical register r_i and provides the mapping to the corresponding physical registers p_j. Using our previous example, we could in i_3 rename r_1 to another free physical register, say p_3, by

[40] Of course, we have to take care that we later make the changes to r_1 visible according to the program order of instructions; see also Section 5.5.1.

adapting the RAT accordingly. Subsequent instructions all access their input registers via the RAT. Hence, instruction i_4 would read its inputs from p_3.

We can also use register-renaming techniques to efficiently rearrange register contents. For example, by exchanging the targets of two entries in the RAT, we can effectively swap the register contents. Note that we do not physically move the register contents; all we do is modify an indirection table that reroutes subsequent register uses. As a table entry is usually just a register index, it is fairly small and can therefore be modified efficiently. In contrast, typical sizes of actual register contents are 32 or 64 bits, hence they are more costly to modify. In principle, there is no limit on the number of RAT entries that we modify at the same time, hence we can also imagine performing operations involving more than two registers.

We now observe the following: on the hardware side, we have register-renaming units providing the ability to efficiently rearrange register contents for multiple registers at once. On the software side, we have the compiler needing exactly this functionality to implement shuffle code. Currently, however, the compiler cannot directly use this hardware. The register renaming is purely controlled by hardware and is transparent to software. Hence, the compiler has to use what usual instruction sets provide: copy instructions and, if available, exchange instructions on two registers. This results in potentially long sequences of copy and swap instructions.

At this point, we pose the following questions:

1. How can we eliminate the detour induced by the instruction set and give the compiler direct access to the underlying hardware's register-renaming capabilities?
2. How can the compiler leverage the new functionality to implement shuffle code more efficiently?
3. Is it worthwhile?

Contribution. In this chapter, we investigate these questions. We propose novel permutation instructions that allow to permute the contents of small sets of registers, develop code-generation approaches that exploit the new instructions to implement shuffle code more concisely, and evaluate when using the new instructions is advantageous. We base our presentation on an in-order architecture extended with renaming capabilities similar to the RAT mentioned before. We later discuss how our findings may carry over to an out-of-order architecture already incorporating register renaming.

Structure. The structure of the following chapters is as follows:

- In Section 5.1, we give an introduction to the problem setting. We cover relevant work related to register allocation and explain the origin of shuffle code. This motivates our concept of permutation instructions that permute up to five registers, which we introduce in the following. We describe how to extend an existing RISC ISA with the new instructions.
- In Section 5.2, as an instruction set extension must always undergo a feasibility study, we describe a prototype implementation of the permutation-instruction concept in an existing RISC microarchitecture.
- In Section 5.3, we study code-generation approaches for the implementation of shuffle code exploiting the new instructions. After first formalizing the problem statement, we propose two algorithms: a fast heuristic and a dynamic-programming-based technique. We formally prove the optimality of the dynamic-programming approach, i.e., we show that its solutions always have minimal length.
- In Section 5.4, we implement both code-generation approaches in a compiler and extensively evaluate compile times as well as code quality using a comprehensive testsuite. We collect precise dynamic instruction counts and validate these numbers by measuring actual running times on an FPGA-based prototype implementation of the permutation hardware.
- In Section 5.5, we argue that the proposed permutation instructions should be cheap to implement on current out-of-order processors that already support register renaming. We describe register-renaming hardware techniques in more detail and discuss how our permutation instructions fit into this scenario.

Acknowledgments. This part of this dissertation is joint work with Lars Bauer, Sebastian Buchwald, Artjom Grudnitsky, Sebastian Hack, Jörg Henkel, Tobias Modschiedler, and Ignaz Rutter [Moh+13; BMR15b; BMR15a]. The complete microarchitecture extension we describe in Section 5.2 as well as the FPGA-based prototype we use in Section 5.4 were designed and implemented by Lars Bauer, Artjom Grudnitsky, Tobias Modschiedler, and Jörg Henkel [Mod13; Moh+13]. The hardware extensions described in Section 5.2 as well as their overhead analysis in Section 5.4.6 are thus not a contribution of this dissertation.

The formal foundations of the code-generation problem and the optimal code-generation algorithm are joint work with Sebastian Buchwald and Ignaz Rutter [BMR15b; BMR15a]. Concerning the theoretical groundwork in Section 5.3, it is difficult to determine exactly who contributed what. The author did contribute significantly to the proofs and proof structure for the greedy algorithm (Section 5.3.3). The optimal dynamic program and its proofs (Section 5.3.5) are almost exclusively Rutter's work. In both cases, the formal rigor with which we present the material is exclusively Rutter's contribution. We thus do not consider Section 5.3.5 a contribution of this dissertation. However, as the author was closely involved, throughout all sections we use "we" instead of "Rutter" and "they".

5.1. Introduction

In this section, we give a brief explanation of the origin of shuffle code in the context of SSA-based register allocation. Then, we present our concept of permutation instructions to permute register contents. We do not build our presentation from first principles. Instead, we restrict ourselves to briefly introducing the most important terms and refer to appropriate literature where needed.

5.1.1. Parallel Copies and Register Transfer Graphs

Static Single Assignment Form (SSA form) [ASU86, section 6.2.4] has become a key property of modern compiler intermediate representations.

```
x = ...;
y = ...;
if (C) {
  t = x;
  x = y;
  y = t;
}
a = x;
b = y;
```

(a) Source program.

```
x_1 = ...;
y_1 = ...;
if (C) {
  t_1 = x_1;
  x_2 = y_1;
  y_2 = t_1;
}
a_1 = φ(x_1, x_2);
b_1 = φ(y_1, y_2);
```

(b) In SSA form.

```
x_1 = ...;
y_1 = ...;
if (C) {



}
a_1 = φ(x_1, y_1);
b_1 = φ(y_1, x_1);
```

(c) After copy propagation.

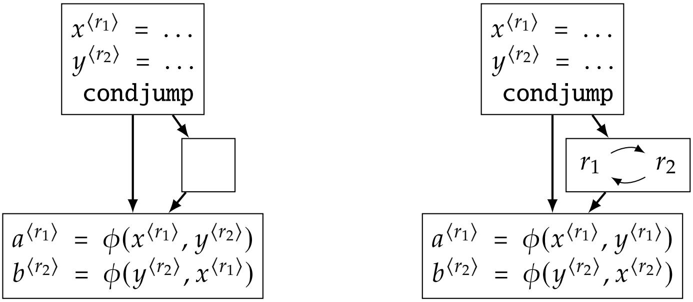
Figure 5.3: Example of conversion to SSA form.

In programs in SSA form, every variable is textually defined exactly once. We can convert a program to SSA form by renaming multiple definitions of each variable x to subscripted versions x_i of that variable. Figure 5.3a shows a simple program that defines two variables x and y, swaps them if some non-constant condition holds, and subsequently uses x and y. Figure 5.3b shows the resulting program in SSA form.

At control-flow joins, we must merge multiple subscripted versions z_i of the same original variable z. For this purpose, SSA form introduces the concept of ϕ-functions. These ϕ-functions are virtual functions[41] that are placed at the beginning of a basic block. They have as many arguments as their containing basic block has predecessors in the control-flow graph. A ϕ-function selects one of its arguments depending on the control-flow path that was taken to reach the current basic block.

After eliminating redundant variables via copy propagation, we get the program shown in Figure 5.3c. We see that the conditional swapping of x and y in the source program is completely encoded in the ϕ-functions. For example, the first ϕ-function selects its first argument x_1 if the condition C is false, and selects the second argument y_1 if the condition C is true, which complies with the semantics of the original program.

[41] Here used in the sense of "imaginary" and unrelated to the object-oriented term.


$x^{\langle r_1 \rangle} = \dots$
$y^{\langle r_2 \rangle} = \dots$
condjump
$a^{\langle r_1 \rangle} = \phi(x^{\langle r_1 \rangle}, y^{\langle r_2 \rangle})$
$b^{\langle r_2 \rangle} = \phi(y^{\langle r_2 \rangle}, x^{\langle r_1 \rangle})$
$x^{\langle r_1 \rangle} = \dots$
$y^{\langle r_2 \rangle} = \dots$
condjump
r_1
r_2
$a^{\langle r_1 \rangle} = \phi(x^{\langle r_1 \rangle}, y^{\langle r_1 \rangle})$
$b^{\langle r_2 \rangle} = \phi(y^{\langle r_2 \rangle}, x^{\langle r_2 \rangle})$

(a) After register allocation. **(b)** Inserted shuffle code.

Figure 5.4: Example of SSA-based register allocation We use $x^{\langle R \rangle}$ to denote that value x is kept in register R.

While the semantics of ϕ-functions is precisely defined, ϕ-functions are a theoretical construct and must be translated into primitive machine operations during code generation. This process is often called "SSA elimination", "SSA destruction", or "translating out of SSA".

Traditionally, SSA form is destructed before register allocation to make the resulting intermediate code compatible with non-SSA-aware register allocators. However, premature SSA destruction unnecessarily constrains register allocation [Hac07]. Research in SSA-based register allocation has led to register allocators that directly work on intermediate code in SSA form [Bri+06; Bou+07; HGG06]. These allocators sustain the SSA property until after register allocation. Hence, the ϕ-functions are still present in the register allocated program.

Figure 5.4a shows the control flow graph of the program from Figure 5.3c after SSA-based register allocation. We use $x^{\langle R \rangle}$ to denote that the value x is kept in register R at this program point. The ϕ-functions now choose between values held in different registers. As no regular processor directly offers ϕ-instructions, the ϕ-functions must be implemented using *shuffle code* that compensates for register mismatches. In the example from Figure 5.4a, this means that the compiler has to insert shuffle code that swaps the contents of registers r_1 and r_2 in the second basic block (see Figure 5.4b). The semantics of ϕ-functions dictates that all ϕ-functions

$\texttt{x}^{\langle r_4\rangle}$ = ...
$\texttt{y}^{\langle r_5\rangle}$ = ...

$\texttt{foo(x}^{\langle r_1\rangle}\texttt{, y}^{\langle r_2\rangle}\texttt{)}$

(a) Function call.

$\texttt{x}^{\langle r_4\rangle}$ = ...
$\texttt{y}^{\langle r_5\rangle}$ = ...
$r_4 \longrightarrow r_1 \qquad r_5 \longrightarrow r_2$
$\texttt{foo(x}^{\langle r_1\rangle}\texttt{, y}^{\langle r_2\rangle}\texttt{);}$

(b) With parallel copy.

Figure 5.5: Adding a parallel copy to satisfy register constraints.

in a basic block must be evaluated simultaneously. Hence, shuffle code consists of *parallel copy* operations.

In general, we may also need to insert such parallel copies before instructions with register constraints. Additionally, the calling convention may enforce certain registers for function call arguments, which we can treat as a special case of register constraint. For example, assume that an architecture requires to pass function arguments in registers with ascending numbers, i.e., r_1, r_2, etc. Figure 5.5 shows an example where we insert a parallel copy before the function call to ensure that the requirements are satisfied.

We see that parallel copies represent the set of mismatching register assignments and at the same time express the necessary copy operations to fix up these mismatches [BC13]. In the examples from Figures 5.4 and 5.5 we have already intuitively visualized parallel copies using *register-transfer graphs* [Hac07, page 56], which we define formally in the following.

Definition 2 A *register-transfer graph* (RTG) is a directed graph, where vertices represent registers and edges represent parallel copy operations between registers. Every vertex has at most one incoming edge, so each register contains an unambiguous value after all copy operations have taken place. □

In our example from Figure 5.4b, the RTG states that r_1 must be transferred to r_2 and, in parallel, r_2 must be transferred to r_1, effectively swapping r_1 and r_2. In Figure 5.5b, the RTG states that we must copy r_4 to r_1 and, independently, copy r_5 to r_2.

The size and shape of RTGs in the program directly depends on the quality of the copy coalescing that has been performed during register allocation. Copy coalescing tries to reduce the cost for copying values between registers as much as possible, i.e., in general tries to reduce the size and number of RTGs. As copy coalescing is NP-complete [BDR07], this reduction comes at great cost in terms of compilation time. Therefore, in certain scenarios, such as just-in-time compilation, we sometimes cannot avoid many potentially large RTGs.

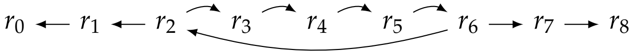


Figure 5.6: A more complex register transfer graph.

Figure 5.6 shows such a large RTG. On regular processor architectures, RTGs must be implemented using register-register copies and, if available, register-register swaps. For large RTGs, this can lead to a substantial amount of code being generated. Hence, it is desirable to be able to implement RTGs more concisely, ideally with a single instruction. This would require fewer instructions, and thus increase performance and decrease code size.

5.1.1.1. Related Work

The most influential approach to register allocation is graph coloring, introduced by Chaitin [Cha82]. Here, program variables are abstracted to nodes in the so-called interference graph. The interference graph is an undirected graph. Two nodes are connected by an edge if a liveness analysis [ASU86, section 9.2.5] determined that the two corresponding variables are live at the same time. A coloring of the interference graph then yields a correct register allocation. Chaitin also showed that for every undirected graph there exists a program, which has that graph as its interference graph [Cha82]. Hence, graph coloring register allocation is NP-hard.

Register allocation is always a trade-off between coalescing and live-range splitting. We say that two variables (or their respective nodes in the interference graph) are *copy-related* if the two variables are involved in a copy instruction. Coalescing aims to assign the same register to two copy-related variables, thereby eliminating the copy altogether. In the interference graph, this corresponds to merging two copy-related nodes. In general, coalescing decreases the number of copies but may increase the register pressure and therefore may cause additional spill code. Spill code saves registers to memory (usually in the current stack frame) and later reloads the spilled value to a register, which is potentially costly.

On the other hand, splitting the live range of a variable means creating a new definition of this variable and inserting a copy instruction between the old definition and the new definition. In the interference graph, this corresponds to splitting a single node into two distinct nodes. The copy instruction gives the register allocator additional freedom as the variable can now effectively change its register after the copy instruction (as represented by the two unconnected nodes in the interference graph). However, this flexibility is not for free: too many copy instructions can slow down execution significantly. In general, live-range splitting may reduce the number of spills, but may increase the number of copies.

Chaitin's original approach always merges two copy-related nodes in the interference graph. This can increase the register pressure of the program and, in turn, can lead to additional spill code. Therefore, this coalescing approach is called *aggressive* coalescing.

Since Chaitin's fundamental work, various improved coalescing techniques have been proposed. Briggs et al. [BCT94] derived criteria for *conservative* coalescing, which means that coalescing never trades a copy for a spill. Park and Moon [PM04] proposed *optimistic* coalescing, which is a conservative technique that tries to undo aggressive coalescing in case a spill was introduced because of a coalesced copy. In general, as the gap between processor speed and memory speed steadily increased, splitting live ranges more often and thereby trading spills for more copies became more attractive.

In 2006, different articles independently proposed performing register allocation on programs in SSA form [Bri+06; Bou+07; HGG06]. In contrast to traditional graph-coloring allocation, the ϕ-functions are still present *after* register allocation [HGG06]. For programs in SSA form, the contained ϕ-functions provide implicit live-range splits. The interference graph of a program in SSA form is *chordal*. This means it is optimally colorable in polynomial time.

In SSA-based register allocation, it is up to the register assignment or a later coalescing pass to find an assignment that involves as few copies as possible. This problem is again NP-hard even on SSA-form programs [BDR07; Hac07]. Various coalescing techniques for SSA-based register allocation have been proposed. Pereira et al. [PP05] and Bouchez et al. [BDR08] proposed novel conservative criteria for node coalescing. Hack and Goos [HG08] introduced recoloring to improve a previously found coloring by trying to assign two copy-related nodes the same color. Grund and Hack [GH07] presented an efficient ILP-based algorithm.

Braun et al. [BMH10] and Colombet et al. [Col+11] presented biasing techniques for the register-assignment phase. Usually, the allocator chooses one register out of a list of free registers. By biasing this choice, those allocators try to pick the same registers for copy-related variables in the first place instead of relying on a post pass, such as recoloring. Biasing usually produces colorings of inferior quality compared to more heavyweight techniques like recoloring or even optimal ILP-based ones. However, biasing techniques are in general more efficient. Wimmer et al. [WF10] adapted the linear-scan register allocator to work directly on SSA form, which simplifies the algorithm. Buchwald et al. [BZB11] presented an approach that integrates register assignment and coalescing by mapping it to the Partitioned Boolean Quadratic Problem.

5.1.2. Permutation Instructions

We first motivate our chosen instruction format and argue why the restriction to permutations is sensible. The instruction format is important as it heavily influences code generation, which we discuss in Section 5.3. The instruction format, as the interface between software and hardware, was developed in collaboration with Lars Bauer, Artjom Grudnitsky, Jörg Henkel, and Tobias Modschiedler.

Our overall goal is to implement RTGs more concisely, therefore increasing performance and decreasing code size. This requires rearranging register contents, which is an extension to the base processor capabilities. The two most common ways to access a processor extension are (i) via a new instruction, i.e., extending the ISA of the CPU, or (ii) by connecting the processor extension to the system bus and using memory-mapped access. Method (ii) can be a good choice for operations that take a long time to complete (e.g., offloading tasks to a co-processor). However, it is unsuitable for an instruction that needs to complete without delay, such as rearranging register contents, due to the inherent latency when accessing the system bus. Therefore, we choose alternative (i).

Ideally, we could implement an arbitrary RTG using a single instruction. However, this raises practical problems, as we must encode the RTG in the instruction. The hardware must be able to decode the instruction quickly, so the encoding scheme must be simple. Moreover, space inside the instruction word is severely limited as well. For most RISC instruction sets, instructions have a fixed size, e.g., 32 bit. CISC instruction sets typically offer instructions with variable length, but also CISC instructions should be as short as possible.

A simple but general encoding scheme encodes each edge of the RTG separately. For example, encoding edges (r_i, r_j) as a pair of register numbers i and j is simple. However, this approach wastes a lot of space. Assuming a standard 32-bit RISC architecture with 32-bit instruction words and 32 registers, we need 5 bits to encode a register index. Hence, we would need 20 bits to encode a small RTG with just two edges.

On the other hand, we could envision a more sophisticated encoding scheme. As we only have a finite (and small) set of registers available, we could enumerate all possible RTGs in some fixed way. We could then identify an RTG with its number according to this enumeration scheme and encode this number in the instruction. However, the matching decoder hardware would be difficult to implement.

Hence, the three goals of (i) encoding arbitrary RTGs, (ii) a compact instruction format, and (iii) efficient decoding hardware are incompatible. Thus, we have to compromise on at least one of these goals. As we extend a RISC architecture, we must adhere to a compact instruction format. Moreover, we are interested in improving performance, so the decoding hardware must be efficient. Thus, we relax our requirement (i) and restrict ourselves to a subset of all possible RTGs.

In this work, we restrict ourselves to *permutations* of registers. Our new permutation instructions must carry with them the permutation that we want to execute. Hence, we must first choose a suitable encoding for permutations. We can express permutations using different notations. An intuitive way of representing a permutation is the two-line notation, where for a permutation σ of a set S, we list the elements x of S in the first row and their images $\sigma(x)$ in the second row. The left hand side of Equation (5.1) shows an example of a particular permutation σ_1 of the set $\{1, 2, 3, 4, 5, 6\}$.

$$\sigma_1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 1 & 2 & 6 & 5 & 4 \end{pmatrix} \equiv (1\,3\,2)(4\,6) \equiv (2\,1\,3)(6\,4) \tag{5.1}$$

However, we can express permutations more concisely using the cycle notation. The cycle notation uses the fact that we can write every permutation as a product of cycles. To build the cycle notation, we start with some element $x \in S$ and repeatedly apply σ, resulting in a sequence $(x\ \sigma(x)\ \sigma(\sigma(x))\ \ldots)$. We stop as soon as we reach the initial element x again, do not append x a second time and call the resulting sequence a cycle. We repeat this process for each element of S that is not part of a cycle yet. The product of all cycles constructed this way is equal to the original permutation, as shown by the right-hand side of Equation (5.1). Note that there are multiple cycle notations for the same permutation depending on which starting element we choose.

We choose the cycle notation for our permutation instructions as it is compact as well as easy to encode and decode. Thus, our permutation instructions take a permutation in cycle form as an argument.

We call the number of elements that a permutation affects its size. For register-file permutation, the maximum size of a permutation is the number of logical registers. In our work, we extend the SPARC V8 ISA [SPA92] with permutation instructions. The SPARC V8 ISA has 32 logical registers. However, instruction width limits the size of a permutation that we can encode in a single instruction. The opcode uses o bits of the instruction word, leaving $n - o$ bits for encoding a permutation, with n being the instruction width. For 32 visible registers, $\lceil \log_2 32 \rceil = 5$ bits are required to identify one register (i.e., encode one element of the permutation). In our implementation for SPARC V8 we need 7 bits for the opcode, leaving us with 25 bits for encoding the permutation. This allows us to encode permutations with a size of up to 5 elements as the immediate of the permutation instruction.

Alternatively, we could store the permutation in a register instead of using an immediate. However, for a 32-bit register, this would only increase the maximum permutation size to $\lfloor 32/\lceil \log_2 32 \rceil \rfloor = 6$. Moreover, we now need two instructions to load the permutation into the register, as it does not fit into an immediate, and an additional instruction to actually execute the permutation. Furthermore, we increase the register pressure by 1. Thus, we decided that this alternative is too expensive and provides too little benefit to be worthwhile. Additionally, as we will see in Section 5.4, small permutations are far more common than large permutations.

Hence, we encode our permutations in cycle notation as immediates. We have extended the SPARC V8 ISA with two instructions for permuting the register file:

(i) `permi5` applies a permutation consisting of a single cycle of size up to 5, and
(ii) `permi23` applies a permutation that is the product of a 2-cycle and a cycle of size up to 3.

The instructions always have five operands. We encode permutations smaller than 5 elements by repeating the last member of the respective cycle. For example, the permutation instruction

```
permi5 r2, r3, r3, r3, r3
```

encodes swapping registers r_2 and r_3. We can encode two 2-cycles in a `permi23`, i.e., a "double swap", using the same technique. Hence,

```
permi23 r2, r3, r4, r5, r5
```

swaps r_2 and r_3 as well as r_4 and r_5. In the remainder of this dissertation we will use `permi` when referring to either permutation instruction.

Both instructions use the same format shown in Figure 5.7, where we refer to the five operands as a, b, c, d, and e. Due to limitations of the free opcode space, we cannot encode a as 5 consecutive bits, but we have to

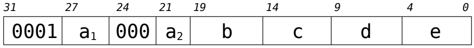


Figure 5.7: Permutation instruction format implemented for the SPARC V8 ISA.

split it into the upper 3 bits a_1 and the lower 2 bits a_2. For `permi5`, each argument corresponds to a member of the 5-cycle. For `permi23`, a and b encode the 2-cycle, while c, d, and e encode the 3-cycle.

The hardware discerns `permi5` and `permi23` instructions by comparing the first two operands. If the register numbers are in ascending order (i.e., $a < b$), the instruction is interpreted as `permi5`, otherwise as `permi23`. Here, we exploit that there are multiple cycle notations for the same permutation. The compiler chooses the cycle representation that results in the correct instruction, i.e., `permi5` or `permi23`, as the assembler only knows one `permi` instruction. We found it simpler to add this functionality to the compiler instead of the assembler.

5.1.2.1. Related Work

Some SIMD extensions of existing instruction sets support value permutation, e.g., Intel x86 [Int17] and PowerPC [Fre16]. The x86 instruction set offers the `PSHUFB` (Packed Shuffle Bytes) instruction as part of SSE3, which permutes bytes in a 256-bit register. It expects the permutation to be passed in a second operand register. Moreover, Advanced Vector Extension (AVX) introduces the `VPERM*` instructions, which do not perform in-place permutations, but write the permuted values into a destination register. In addition, the `VPERM*` instructions allow value duplication. The PowerPC AltiVec extension offers the `vperm` instruction, which extracts bytes from two 128-bit source registers and arranges them according to a user-definable mask into a 128-bit destination register. As for `VPERM*`, value duplication is permitted.

Both ISAs allow permutation only on values within one (or two) registers, but not between registers. Furthermore, the instructions are limited to special vector registers reserved for SIMD processing. Hence, they are unsuitable for implementing RTGs. In contrast, our proposed permutation instructions work on general-purpose registers.

Instructions of VLIW architectures include one operation per functional unit of the processor. Hence, on a 4-way VLIW architecture it may be possible to encode 4 swaps or 4 copies in one instruction. In contrast, our proposed permutation instructions target a non-VLIW architecture.

5.2. Hardware Implementation

This section describes the hardware implementation of our permutation instructions. The details of the hardware implementation are not necessary to follow the discussion of code-generation approaches in Section 5.3. Hence, the reader may skip this section. Still, we provide an overview of the hardware implementation for the sake of completeness.

The hardware implementation we describe in this section is not a contribution of this dissertation. The complete hardware was implemented by Lars Bauer, Artjom Grudnitsky, Tobias Modschiedler, and Jörg Henkel [Moh+13]. The content in this section is based on an unpublished extended version of [Moh+13]. Modschiedler [Mod13] gives the most extensive description of the hardware implementation available.

5.2.1. Fundamental Pipeline Modifications

The underlying processor for the implementation is a Gaisler LEON 3 [Cob17b]. The LEON 3 uses an in-order 7-stage pipeline. As an example, Figure 5.8 shows the processing of an add instruction.

The pipeline stages have the following tasks:

Fetch Retrieve the instruction word from the instruction cache.

Decode Extract instruction type as well as operand and destination registers from the instruction word. Use the operand registers as address inputs to the register file (one cycle latency for read or write access).

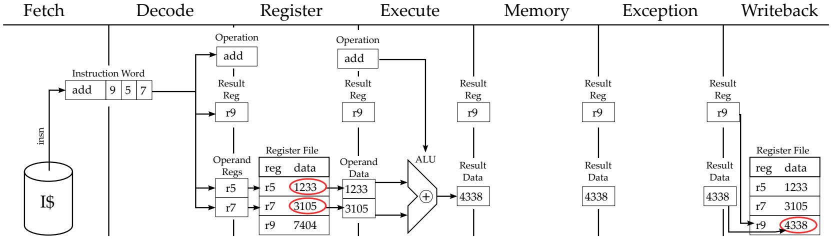


Figure 5.8: 7-stage RISC pipeline of the base architecture executing an add instruction.

Register Read operand data requested in the Decode stage from the register file and write it into operand-data pipeline registers.

Execute Execute arithmetic and branch operations. For an arithmetic operation, the ALU uses the contents of the operand-data pipeline registers as inputs and stores the result in the result-data register.

Memory Perform load and store operations.

Exception Handle traps and interrupts. This requires the following steps:

1. save the program counter (PC) and next program counter (NPC),
2. annul all instructions before the Exception stage,
3. execute the trap-handler routine,
4. restore the saved PC and NPC, and
5. resume execution of the original program.

Writeback If the instruction has a result, write the result data to the register file at the address specified by the result register.

SPARC V8 architectures organize the register file in multiple *register windows*. While the register file holds 136 entries in an implementation with 8 register windows, only 32 registers are visible at a time, defined by the *current-window pointer*. Certain SPARC V8 instructions, such as, e.g., `restore` or `save`, modify the current-window pointer. We refer to the SPARC V8 standard [SPA92] for details about register windows.

To support register file permutation, Bauer et al. introduce the distinction between *logical* and *physical* register addresses. All instructions only refer to logical register addresses, whereas actual register file accesses

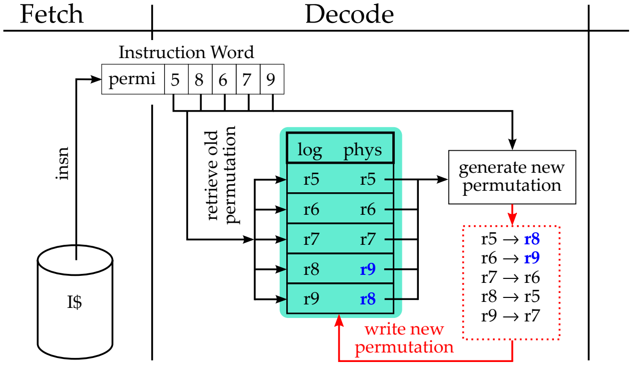


Figure 5.9: Applying the permutation (5 8 6 7 9) using the `permi5` instruction.

use physical register addresses. Bauer et al. add the *permutation table* to translate logical to physical register addresses. The permutation table stores the current logical-to-physical mapping of register addresses for all registers (i.e., it has 136 entries for 8 register windows used in their implementation). This corresponds to a permutation written in two-line notation (see left part of Equation (5.1) in Section 5.1.2).

Figure 5.9 shows how a `permi5` instruction applies the permutation (58679) to the register file. We only show the entries for registers r_5 to r_9 of the permutation table. The application of a `permi23` works analogously.

The execution of a permutation instruction consists of four steps, all of which happen in the Decode stage:

1. The instruction decoder recognizes a `permi5` or `permi23` instruction and extracts the five operands that define the permutation π_i carried by the instruction word.
2. As we have to apply π_i to an already existing permutation π_t saved in the permutation table, we first read π_t from the table.
3. Permutation instructions define a permutation on the current window, thus we use the current window pointer to filter the entries of the current window from π_t.

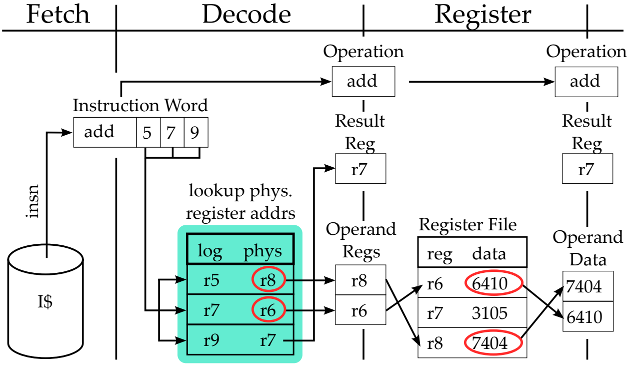


Figure 5.10: Executing the instruction add r_5, r_7, r_9 on a permuted register file. Logical registers r_5 and r_7 are operands, r_9 is the logical destination register.

4. We compute $\pi_t' = \pi_t \circ \pi_i^{-1}$ and write π_t' back to the permutation table. Intuitively, for some given register r, we look up the logical source register s of r (with π_i^{-1}) and then look up which physical register p we currently map s to (with π_t). This p is the new physical register for r, thereby implementing "copying" s to r.

In the example from Figure 5.9, the permutation table initially contains the cycle $\pi_t = (8\,9)$. The permi5 instruction encodes the cycle $\pi_i = (5\,8\,6\,7\,9)$. We compute $\pi_t' = (8\,9) \circ (5\,8\,6\,7\,9)^{-1} = (8\,9) \circ (5\,9\,7\,6\,8) = (5\,8)(6\,9\,7)$ and write π_t' back to the permutation table.

To execute a regular instruction with the modified pipeline, the important difference happens in the Decode stage. Figure 5.10 shows the pipeline activities of an add instruction immediately following the permi5 instruction from the example shown in Figure 5.9. It shows how, in the Decode stage of the pipeline, we first translate the logical register addresses from the instruction to physical register addresses. The translation is equivalent to the application of the permutation π_t from the table to the registers of the instruction. We then use the resulting physical register addresses to access the register file.

Permutation instructions do not induce read-after-write hazards in the pipeline, thus Bauer et al. do not have to extend the pipeline-forwarding logic. This is because permutation instructions *commit* their changes in the Decode stage. Hence, once the following instruction (add in the example) is in the Decode stage one cycle later, the permutation table has already been updated with the new permutation. Bauer et al. call this characteristic *early committing*.

At system reset, they initialize the permutation table with the identity permutation, i.e., the physical address of each register is the same as its logical address. Subsequent permutation instructions modify the permutation table. Permutations are transparent to the operating system (OS), so Bauer et al. do not need to modify OS code for context switches. For instance, if the OS wants to save r_5 from a task (to restore it later), it will actually access the physical register that currently holds the value of r_5 (which is r_8 in Figure 5.10). When restoring r_5 later, it may be written to a different register; however, an access to r_5 will provide the same data that was initially saved.

5.2.2. Exception Handling

The architecture outlined in the previous section can execute programs that use permutation instructions, unless traps occur during execution. The SPARC V8 standard specifies three categories of traps:

1. Precise traps are induced by particular instructions, e.g., unknown instructions, trap-on-condition instructions or instructions causing a register-window overflow or underflow.
2. Deferred traps are caused by floating-point and co-processor instructions and become visible after the instruction that caused them has committed.
3. Interrupting traps are caused by external interrupts, e.g., timer interrupts or I/O components notifying the processor that a buffer is full.

A program that runs directly on the hardware (without an OS), does not use any I/O components, and uses the register windows in a way that incurs neither window overflows nor underflows[42] will not cause any traps. However, when executing on a multi-tasking OS, the program is likely to be interrupted, e.g., by the timer used to periodically invoke the OS scheduler, by the page fault handler or by interrupts caused by peripherals.

The underlying architecture handles traps in the Exception stage. After the trap-handler code has finished, the program counter and next program counter are restored and regular program execution continues. Instructions must not be executed twice, i.e., instructions already in the pipeline before the trap is detected must not first proceed through the pipeline at the start of the trap handler and then be executed again after the old program counter is restored and the corresponding instruction is reloaded.

Therefore, at the time of trap detection, the pipeline automatically annuls all instructions that are currently in pipeline stages Exception or earlier. This ensures that they are executed only once: after trap handling has finished, the old program counter is restored and the instructions are loaded into the pipeline again.

However, Bauer et al. cannot simply annul permutation instructions as described above, as, due to their early committing characteristic, these instructions already modify the permutation table in the Decode stage. Hence, they differentiate between three cases for annulling permutation instructions during a trap:

1. They flag permutations in the Fetch stage with an annul bit. Permutations marked this way do not update the permutation table in the Decode stage.

2. They notify permutations in the Decode stage with a cancel signal. They send the cancel signal at the time they detect a trap in the Exception stage. If they see the cancel signal in the Decode stage, they create the new permutation π'_t but do not write it back to the permutation table.

[42] This is possible by compiling the program in a way that does not change the register window during function calls. This is often called a “flat” register model. For GCC, the option `-mflat` enables the flat register model.

Register	Execute	Memory	Exception
permi π_4	permi π_3	permi π_2	permi π_1

Figure 5.11: Traps require reversal of up to four permutations, depending on the pipeline state.

3. Permutations in the Register, Execute, Memory, and Exception stages have already passed the Decode stage. Hence, they have already modified the permutation table. Thus, Bauer et al. cannot annul or cancel them anymore. Instead, they need to *revert* their change to the permutation table.

It is imperative to handle all three cases. Otherwise, a permutation could modify the permutation table more than once, possibly leading to wrong register contents and a violation of program semantics.

In order to revert the change of a permutation π, Bauer et al. compute its inverse π^{-1} and apply it to the permutation table. As $\pi^{-1} \circ \pi = \mathrm{id}$, this reverts the change of π.

At the time of a trap, up to four pipeline stages might hold permutation instructions that we cannot annul or cancel: Register, Execute, Memory, and Exception. To invert multiple permutations, we compute the inverse of each of them and apply the inverse permutations in the reverse order we have applied the original permutations. Assuming, at the time of a trap, the pipeline state shown in Figure 5.11, we need to apply four inverse permutations to restore the original state of the permutation table. As $\pi_1^{-1} \circ \pi_2^{-1} \circ \pi_3^{-1} \circ \pi_4^{-1} \circ (\pi_4 \circ \pi_3 \circ \pi_2 \circ \pi_1) = \mathrm{id}$, this reverts the permutation table to its previous state.

The implementation of this concept requires that Bauer et al. propagate the permutations carried by permutation instructions through the pipeline up to the Exception stage. This requires four additional 25-bit wide pipeline registers and four corresponding 1-bit registers, which indicate whether the instruction was a permutation.

Figure 5.12 shows the updated pipeline structure. Bauer et al. extend the Exception stage to check whether at least one of the instructions in the Register, Execute, Memory, or Exception stages is a permutation. If they

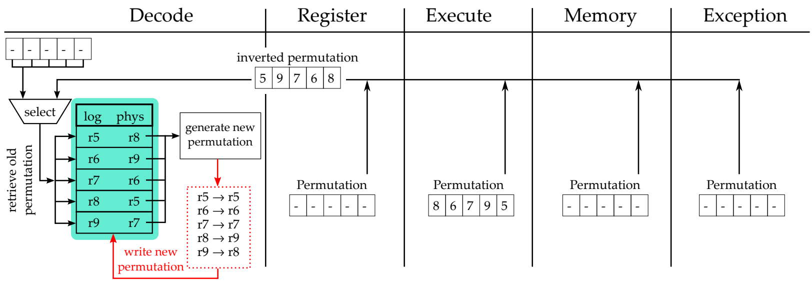


Figure 5.12: Implementation of permutation reversion during trap detection. Here, only one permutation from the Execute stage needs to be reverted.

detect no permutations in these stages, exception handling continues as usual.

Otherwise, they halt the pipeline and check each of the mentioned stages for a permutation. Bauer et al. check in the reverse order of application, i.e., Register, Execute, Memory, and finally Exception. For each permutation they detect, they compute an inverse permutation and apply it to the permutation table like a regular permutation. They can compute and apply one inverse permutation per cycle. Thus, permutation inversion can take up to four cycles per trap.

Implementing inversion as a multi-cycle operation is necessary to prevent increasing critical path length (and thus reducing processor frequency). Bauer et al. generate inverse permutations by reversing the cycle(s) of the original permutation (i.e., reversing the order of the arguments of the permutation).

5.3. Code Generation

As we now know what our new instructions look like and work they work, we discuss code-generation approaches for RTGs in the following. First, we briefly describe how we can implement RTGs on regular machines without permutation instructions, i.e., just using `copy` and `swap`. Then, we study how to exploit our new permutation instructions. Hence, our goal is to implement a given RTG with a minimal number of instructions using `copy`, `permi23`, and `permi5`.

Let us restate that a register-transfer graph (RTG) is a directed graph, where each vertex represents a register and an edge (u, v) means that the content of register u before the execution of the RTG must be in v after the execution. All copy operations in an RTG are assumed to be performed in parallel. Therefore, each vertex in the graph has at most one incoming edge, because the register content would be undefined if multiple concurrent copy operations wrote to the same destination register. However, a vertex can have multiple outgoing edges, which means that the register value is duplicated, and even loops (u, u), indicating that the register contents must be preserved.

We call a sequence of register-transfer instructions, such as `copy`, `swap`, `permi23`, and `permi5`, a *shuffle code*. A shuffle code *implements* an RTG if after the execution of the shuffle code, every register whose corresponding vertex has an incoming edge has the correct content. We call a shuffle code *optimal* regarding a certain RTG if the shuffle code has minimal length and implements the RTG.

Additionally, we introduce two special types of RTGs. First, *outdegree-1 RTGs* are RTGs where the maximum out-degree of every vertex is 1. Hence, outdegree-1 RTGs do not allow value duplication. And second, *PRTGs* (for *permutation* RTGs), where the in-degree and out-degree of every vertex are exactly 1. Every PRTG is an outdegree-1 RTG. We call an RTG *trivial* if it has only self-loops. In this case, it needs no shuffle code as every register already contains the right value.

In the rest of this chapter, we will identify registers r_i with their numbers i to improve readability and simplify dealing with permutations. Figure 5.13 shows simple RTGs, presented in the style we will use in the following.

5.3.1. Implementing RTGs on Regular Machines

We first look at how we can implement RTGs on traditional machines, i.e., using copy and swap instructions on registers. Implementing a given RTG G works as follows [Hac07, p. 56–57]. We assume that registers that

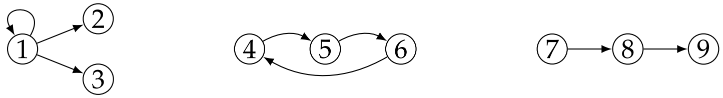


Figure 5.13: Example RTGs. On the left an RTG containing a loop; in the middle a PRTG; on the right an outdegree-1 RTG.

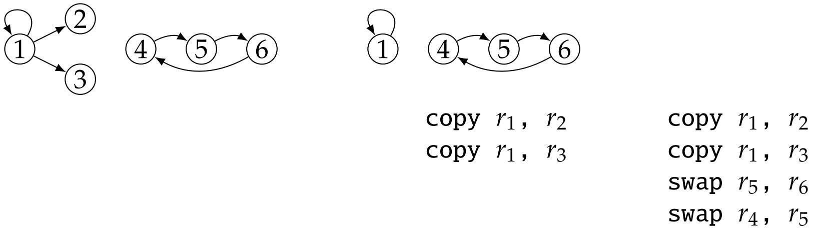
do not take part in the shuffle code, i.e., whose respective vertices in the RTG would have no incident edges, have been removed from the input RTG. Denote with F the set of free registers (see [Hac07, p. 56] on how to determine F).

1. If there is a vertex n with no outgoing edges, there must be exactly one edge (n', n) with $n \neq n'$. Emit a register-register copy $n' \longrightarrow n$. We may overwrite the value in n, as we do not need it anymore (as n has no outgoing edges). Remove the edge (n', n) from G's edge set.
   Then, replace each edge (n', m) (except for self-loops (n', n')) with an edge (n, m). This is correct as, after the copy, n and n' contain the same value. Replacing self-loops as well would be correct, but would lead to unnecessary copy instructions. Put n' into the set of free registers F. Repeat step 1.
2. Now G is a (possibly empty) PRTG. Cycles of length 1 (self-loops) do not require any instructions. We can implement cycles of length 2 or greater as follows:

   - If there is a free register, i.e., $F \neq \emptyset$, let $r_t \in F$. Implement a cycle $(r_1, \ldots, r_k)$ by k copies following the scheme $r_k \longrightarrow r_t, r_{k-1} \longrightarrow r_k, \ldots, r_t \longrightarrow r_1$.
   - If there is no free register, decompose a cycle of length k into $k - 1$ transpositions, and implement these using $k-1$ register-register swap instructions. If the instruction set does not offer a swap instruction, we can use arithmetic or bitwise operations to achieve the same effect [War02, section 2–19].

Figure 5.14 shows the implementation that the algorithm generates for a simple input RTG under the condition that $F = \emptyset$[43].

[43] Technically, we could use registers r_2 or r_3 as temporary registers, implement the cycle first, and then copy r_1 to r_2 and r_3.



```
copy r1, r2
copy r1, r3
```

```
copy r1, r2
copy r1, r3
swap r5, r6
swap r4, r5
```

Figure 5.14: Implementation of an example RTG using copy and swap instructions. On the left the input RTG before step 1 of the algorithm; in the middle the RTG after step 1 with the code emitted up to this point; on the right the finished implementation after step 2. We omit registers without incident edges for presentation reasons.

We now turn to the question of optimality. Note that each edge (except self-loops) in an RTG expresses a transfer operation and thus some instruction must perform that transfer. A copy instruction can implement an arbitrary RTG edge. A swap instruction can implement two RTG edges at the same time; however, the edges must be of the form (a, b) and (b, a).

Step 1 of the above algorithm is clearly optimal as we use copies to implement edges (u, v) whose target vertex v does not have further outgoing edges—in particular, there is no edge (v, u). Hence, we would not gain anything by using a swap and thus implementing (u, v) with a copy instruction is optimal. Note that, as long as u has no self-loop, it would be correct to use a swap instruction; it is just not better than using a copy.

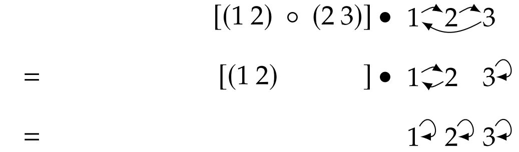
Step 2 of the algorithm does, in general, not lead to an optimal shuffle code (according to our definition of optimality) as it prefers k copy instructions over $k - 1$ swap instructions for a cycle of size k. This is because modern processors often handle copy instructions specially, as they are far more common than swap instructions; see Section 5.5 for details. However, a slightly modified algorithm that always chooses swap instructions for cycles is optimal.

5.3.2. Reformulation as a Graph Problem

We will now look at generating code RTGs with permutation instructions. As our set of instructions, we use `permi23`, `permi5`, and `copy`. As `permi5` can also encode the swapping of two registers, it is strictly more powerful than `swap`. Hence, we do not consider `swap` in the following.

Before we discuss the actual code-generation scheme, we will first rephrase our problem statement as a graph problem. If we can define the effect of an instruction on an RTG, we can view our problem as finding a shuffle code that, applied to an RTG, makes it trivial.

It is easy to define the effect of a permutation on an RTG. Let G be an RTG and let π be an arbitrary permutation that is applied to the contents of the registers. We define $\pi \bullet G = \pi G = (V, \pi E)$, where $\pi E = \{(\pi(u), v) \mid (u, v) \in E\}$. This models the fact that if v should receive the data contained in u, then after π moves the data contained in u to some other register $\pi(u)$, the data contained in $\pi(u)$ should end up in v. By applying this definition, we can now, formally explain why the permutation (1 2 3) resolves a cyclic shift of registers r_1, r_2, r_3:



Unfortunately, it is not possible to directly define the effect of a copy operation on an RTG. There are often multiple RTGs that could be the result of applying a copy operation to an RTG. Figure 5.15 shows an example where we apply a copy operation to a simple RTG. After the copy operation 1 ⟶ 2, there are two possible sources for the value of r_3 and it is unclear how to choose one.

Even more problematic are copy operations that do not work along an existing edge in the RTG. For example, how should we define the effect of the copy operation 2 ⟶ 1 on the input RTG from Figure 5.15? Applying this copy operation would overwrite the value in r_1. This value is then lost, and it is not recoverable. Hence, we are somehow stuck, and the value transfers described by the original RTG are now impossible (as we lost one of the required values). This asymmetry between permutation and copy operations reflects a fundamental difference in their semantics: permutations only redistribute values, but never duplicate or destroy them, wheras copy operations can do that.

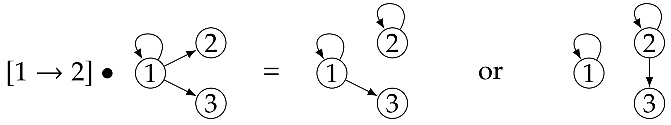


Figure 5.15: Attempt at defining the effect of copy operations on RTGs. After the copy operation 1 → 2, there are two possible sources for the value of r_3 and it is unclear how to choose one.

Therefore, instead of trying to define the semantics of applying an arbitrary copy operation to an RTG, we rely on the following observation. Consider an arbitrary shuffle code that contains a copy $a \longrightarrow b$ with source a and target b that is followed by a transposition $\tau = (c\ d)$ of the contents of registers c and d. We can replace this sequence with the same transposition $(c\ d)$ and a copy $\tau(a) \longrightarrow \tau(b)$. Thus, given a sequence of operations, we can successively move the copy operations to the end of the sequence without increasing its length. Hence, for any RTG there exists a shuffle code that consists of a pair of sequences $((\pi_1, \dots, \pi_p), (c_1, \dots, c_t))$, where the π_i are permutation operations and the c_i are copy operations.

We now strengthen our assumption on the copy operations. The following proofs of Lemmas 1 and 2 are the work of Rutter and thus not a contribution of this dissertation.

Lemma 1 *Every instance of the shuffle code generation problem has an optimal shuffle code $((\pi_1, \dots, \pi_p), (c_1, \dots, c_t))$ such that*

(i) No register occurs as both a source and a target of copy operations.
(ii) Every register is the target of at most one copy operation.
(iii) There is a bijection between the copy operations c_i and the edges of πG that are not loops, where $\pi = \pi_p \circ \pi_{p-1} \circ \cdots \circ \pi_1$.
(iv) If u is the source of a copy operation, then u is incident to a loop in πG.
(v) The number of copies is $\sum_{v \in V} \max\{\deg^+_G(v) - 1, 0\}$. □

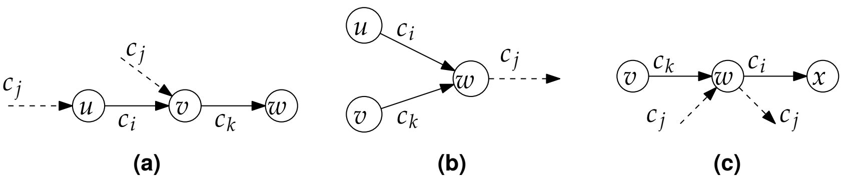


Figure 5.16: Illustration of the proof of Lemma 1. The copies c_j with $i < j < k$ along the dashed edges would contradict the choice of i or k.

Proof Consider an optimal shuffle code of the form $((\pi_1, \ldots, \pi_p), (c_1, \ldots, c_t))$ as above and assume that the number t of copy operations is minimal among all optimal shuffle codes.

Suppose there exists a register that occurs as both a source and a target of copy operations or a register that occurs as the target of more than one copy operation. Let k be the smallest index such that in the sequence $c_1, \ldots, c_k$ there is a register occurring as both a source and a target or a register that occurs as a target of two copy operations. We show that we can modify the sequence of copy operation such that the length of the prefix without such registers increases. Inductively, we then obtain a sequence without such registers.

Let v and w denote the source and target of c_k, respectively. Let i denote the largest index such that c_i is a copy operation that has w as a source or target or such that c_i is a copy operation with target v. We distinguish three cases based on whether c_i has target v, target w, or source w.

Case 1: The target of c_i is v; see Figure 5.16a. Let u denote the source of operation c_i. The sequence first copies a value from u to v and from there to w. Then, we replace c_k with a copy with source u and target w. (If $u = w$, we omit the operation altogether.) This only changes the outcome of the shuffle code if the value contained in u or v is modified between operations c_i and c_k, i.e., if there exists a copy operation c_j with $i < j < k$ whose target is either u or v. But then already the smaller sequence $c_1, \ldots, c_j$ has u occur as both a source and a target or v as a target of two operations, contradicting the minimality of k.

Case 2: The target of c_i is w; see Figure 5.16b. In this case, the copy operation c_i copies a value to w and later this value is overwritten by the operation c_k. Note that by the choice of i there is no operation c_j with $i < j < k$ with source w. Thus, omitting the copy operation c_i does not change the outcome of the shuffle code. A contradiction to optimality.

Case 3: The source of c_i is w; see Figure 5.16c. Let x denote the target of operation c_i. In this case, first a value is copied from w to x and later the value in v is copied to w. We claim that no copy operation c_j with $i < j < k$ involves x or w. If x occurs as the source of c_j, then x occurs as a source and target in the sequence $c_1, \dots, c_j$. If x occurs as the target of c_j, then x occurs twice as a target in $c_1, \dots, c_j$. In both cases, this contradicts the minimality of k. If w is the target of c_j, then w occurs as a source and a target in the sequence $c_1, \dots, c_j$, contradicting the choice of k. If w is the source of c_j we have a contradiction to the choice of i. This proves the claim.

We can thus, without changing the outcome of the shuffle code, move the operation c_i immediately before the operation c_k. Then, our sequence contains consecutive copy operations $w \to x$ and $v \to w$. Replace these two operations with a cyclic shift of $\{ v, w, x \}$ and a copy operation $w \to v$. This decreases the number of copy operations by 1 and thus contradicts the minimality of t.

Altogether, in each case, we have either found a contradiction to the optimality of the shuffle code, to the minimality of the number of copy operations, or we have succeeded in producing a shuffle code that has a longer prefix satisfying properties (i) and (ii). Inductively, we obtain a shuffle code satisfying both (i) and (ii). Fix such a code. Since no register is both source and target of a copy operation, the copy operations are commutative and can be reordered arbitrarily without changing the result.

For property (iii) first observe that the only way to transfer a value from u to v is via a copy operation $u \to v$. This is due to the facts that the shuffle code is correct, that no node occurs as both a source and a target of copy operations, and that π only permutes the values in the initial registers but does not duplicate them. Thus, for every edge there must be a corresponding copy operation. Conversely, this number of copy operations certainly suffices for a correct shuffle code for πG.

For property (iv) consider a copy operation from u to v such that u is not incident to a loop. If the indegree of v in πG were 1, then there would be an incoming edge, which would correspond to a copy operation with target u, which is not possible by property (i). Thus, u has indegree 0. But then, the contents of u are irrelevant and we can replace the copy from u to v by an operation that swaps the contents of u and v, resulting in a shuffle code with fewer copy operations.

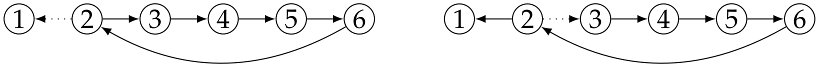
By property (iv) every vertex that is the source of an edge in πG is incident to a loop. Hence $\sum_{v \in V} \max\{\deg^+_{\pi G}(v) - 1, 0\}$ is the number of non-loop edges in πG, which is the same as the number of copy operations by property (iii). Note that by definition π only permutes the outdegrees of the vertices, and hence $\sum_{v \in V} \max\{\deg^+_{\pi G}(v) - 1, 0\} = \sum_{v \in V} \max\{\deg^+_{G}(v) - 1, 0\}$. This shows property (iv) and finishes the proof. ■

We call a shuffle code satisfying the conditions of Lemma 1 *normalized*. Observe that the number of copy operations used by a normalized shuffle code is a lower bound on the number of necessary copy operations since permutations, by definition, only permute values but never create copies of them.

Consider now an RTG G together with a normalized optimal shuffle code and one of the shuffle code's copy operations $u \longrightarrow v$. Since the code is normalized, the value transferred to v by this copy operation is the one that stays there after the shuffle code has been executed. If v had no incoming edge in G, then we could shorten the shuffle code by omitting the copy operation. Thus, v has an incoming edge (u', v) in G, and we associate the copy $u \longrightarrow v$ with the edge (u', v) of G. In fact, $u' = \pi^{-1}(u)$, where $\pi = \pi_p \circ \cdots \circ \pi_1$. In this way, we associate every copy operation with an edge of the input RTG. In fact, this is an injective mapping by Lemma 1 (ii). We define $G - C := (V, E \setminus C)$ for an RTG and an edge set C.

Lemma 2 *Let $((\pi_1, \ldots, \pi_p), (c_1, \ldots, c_t))$ be an optimal shuffle code S for an RTG $G = (V, E)$ and let $C \subseteq E$ be the edges that are associated with copies in S. Then*

(i) Every vertex v has $\max\{\deg^+_G(v) - 1, 0\}$ *outgoing edges in C.*
(ii) $G - C$ is an outdegree-1 RTG.
(iii) $\pi_1, \ldots, \pi_p$ is an optimal shuffle code for $G - C$. □



(a) With $C = \{(1,2)\}$, we need one permutation and one copy operation.

(b) With $C = \{(2,3)\}$, we need two permutation operations and one copy.

Figure 5.17: The choice of the copy set is crucial for obtaining an optimal shuffle code. We show edges in the copy set as dotted lines. With $C = \{(1,2)\}$, the RTG obtains the normalized optimal shuffle code (π_1, c_1), where $\pi_1 = (2\,3\,4\,5\,6)$ and $c_1 = 3 \rightarrow 1$. However, after putting the edge $(2,3)$ (instead of $(1,2)$) into the copy set, we cannot achieve an optimal solution anymore.

Proof For property (i) observe that, since permuting the register contents does not duplicate values, it is necessary that at least $\max\{\deg^+_G(v) - 1, 0\}$ of the edges of v are implemented by copy operations and thus are in C. By property (v) of Lemma 1, the number of copy operations is exactly the sum of these values, which immediately implies that equality holds at every vertex.

Property (ii) follows immediately from property (i).

Finally, for property (iii), suppose there is a shorter optimal shuffle code $\pi'_1, \ldots, \pi'_{p'}$ with $p' < p$ for $G - C$. Let $\pi' = \pi'_{p'} \circ \cdots \circ \pi'_1$. Then $\pi' G$ has $|C|$ edges that are not loops and by creating a copy operation for each of them we obtain a shorter shuffle code. This is a contradiction to the optimality of the original shuffle code. Hence property (iii) holds. ■

Lemma 2 shows that we can find an optimal shuffle code for an RTG G by first picking for each vertex one of its outgoing edges (if it has any) and removing the remaining edges from G; second finding an optimal shuffle code for the resulting outdegree-1 RTG; and finally creating one copy operation for each of the previously removed edges. We call the set of edges that we implement by copies a *copy set* and will denote it with C by default. We will study copy sets further in Section 5.3.4. Figure 5.17 shows that the choice of the copy set is crucial to obtain an optimal shuffle code.

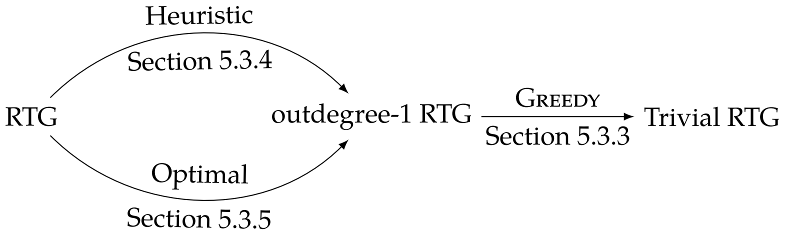


Figure 5.18: Structure of the following sections. We first study a greedy approach to generate shuffle code for an outdegree-1 RTG. We then present two approaches for picking a copy set: a heuristic and an optimal approach.

Overview. Hence, to generate code for an RTG, we first pick a copy set and then generate shuffle code for the resulting outdegree-1 RTG. Figure 5.18 shows an overview of the following sections. We start with the last step: generating shuffle code for an outdegree-1 RTG. Section 5.3.3 shows how to compute an optimal shuffle code for an outdegree-1 RTG using a greedy algorithm. Afterwards, we present two techniques for choosing a copy set:

(i) a simple heuristic with linear running time in Section 5.3.4, and
(ii) an optimal algorithm with running time $O(n^4)$ in Section 5.3.5.

In this case optimal means that, for an input RTG G, the algorithm chooses a copy set C such that the resulting outdegree-1 RTG $G - C$ (where we removed all edges in C from G) still admits a shuffle code with the smallest number of operations.

5.3.3. Optimal Shuffle Code for Outdegree-1 RTGs

In this section we propose a greedy algorithm to generate code for outdegree-1 RTGs. Furthermore, we prove its optimality.

Before we formulate the algorithm, let us look at the effect of applying a transposition $\tau = (u\ v)$ to contiguous vertices of a k-cycle $K = (V_K, E_K)$ in a PRTG G, where k-cycle denotes a cycle of size k. Hence, $u, v \in V_K$ and

$(u, v) \in E_K$. Then, in τG, the cycle K is replaced by a $(k-1)$-cycle and a vertex v with a loop. We say that τ has reduced the size of K by 1. If τK is trivial, we say that τ *resolves* K. It is easy to see that `permi5` reduces the size of a cycle by up to 4 and `permi23` reduces the sizes of two distinct cycles by 1 and up to 2, respectively.

We can now formulate Greedy as follows.

1. Complete each directed path of the input outdegree-1 RTG into a directed cycle, thereby turning the input into a PRTG.
2. While there exists a cycle K of size at least 4, apply a `permi5` operation to reduce the size of K as much as possible.
3. While there exist a 2-cycle and a 3-cycle, resolve them with a `permi23` operation.
4. Resolve pairs of 2-cycles by `permi23` operations.
5. Resolve triples of 3-cycles by pairs of `permi23` operations.

Figure 5.19 shows how Greedy generates code for the example RTG with 9 vertices shown in Figure 5.19a. First, we complete the right component, a path, into a cycle (Figure 5.19b) and obtain a PRTG. Then, the right component is a cycle of size 6, so, as shown in Figure 5.19c, we apply step 2 of Greedy, generating a `permi5`. We thereby reduce the right component to a 2-cycle. Now, as Figure 5.19d shows, step 3 of Greedy resolves the remaining 2-cycle and 3-cycle with one `permi23` operation. Hence, Greedy has transformed our input outdegree-1 RTG into a trivial RTG using two permutation operations.

We claim that Greedy computes an optimal shuffle code. Let G be an outdegree-1 RTG and let Q denote the set of paths and cycles of G. For a path or cycle $\sigma \in Q$, we denote by $\mathrm{size}(\sigma)$ the number of vertices of σ. We define $X = \sum_{\sigma \in Q} \lfloor \mathrm{size}(\sigma)/4 \rfloor$ and $a_i = |\{\sigma \in Q \mid \mathrm{size}(\sigma) = i \bmod 4\}|$ for $i = 2, 3$. We call the triple $\mathrm{sig}(G) = (X, a_2, a_3)$ the *signature* of G.

Lemma 3 *Let G be an outdegree-1 RTG with* $\mathrm{sig}(G) = (X, a_2, a_3)$*. The number* Greedy$(G)$ *of operations in the shuffle code produced by the greedy algorithm is* Greedy$(G) = X + \max\{\lceil (a_2 + a_3)/2 \rceil, \lceil (a_2 + 2a_3)/3 \rceil\}$. □

Proof After the first step we have a PRTG with the same signature as G. Clearly Greedy produces exactly X operations for reducing all cycle

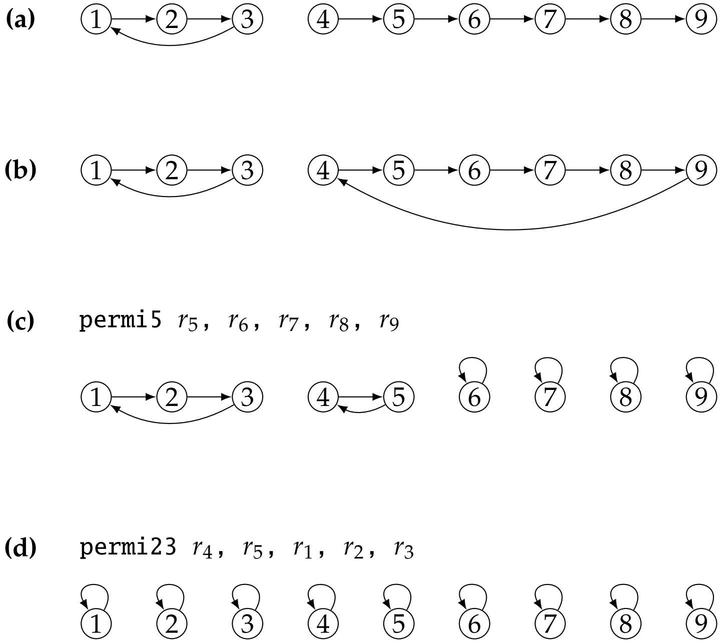

(a)
1
2
3
4
5
6
7
8
9
(b)
1
2
3
4
5
6
7
8
9
(c)
permi5 r_5, r_6, r_7, r_8, r_9
1
2
3
4
5
6
7
8
9
(d)
permi23 r_4, r_5, r_1, r_2, r_3
1
2
3
4
5
6
7
8
9

Figure 5.19: Example illustrating how Greedy generates code for an outdegree-1 RTG. In (a), we show the input RTG; (b) shows the resulting PRTG; (c) shows the permi5 operation generated by Greedy and the resulting modified PRTG; and (d) shows the second operation issued by Greedy as well as the final trivial RTG.

$$\begin{tikzcd}
G \arrow[rr, "\textsc{Greedy}(G)"] \arrow[dr, "1\ (\pi)"] & & \text{Trivial RTG} \\
& \pi G \arrow[ur, "\textsc{Greedy}(\pi G)"] &
\end{tikzcd}$$

Figure 5.20: Idea behind the strategy to prove the optimality of Greedy. G is an outdegree-1 RTG, π is a permutation instruction.

sizes below 4. Afterwards, only `permi23` operations are used to resolve the remaining cycles of size 2 and 3.

If $a_2 \geq a_3$, then first a_3 operations are used to resolve pairs of cycles of size 2 and 3. Afterwards, the remaining $a_2 - a_3$ cycles of size 2 are resolved by using $\lceil (a_2 - a_3)/2 \rceil$ operations. In total, these are $\lceil (a_2 + a_3)/2 \rceil$ operations.

If $a_3 \geq a_2$, then first a_2 operations are used to resolve pairs of cycles of size 2 and 3. Afterwards, the remaining $a_3 - a_2$ cycles of size 3 are resolved by using $\lceil 2(a_3 - a_2)/3 \rceil$ operations. In total, these are $\lceil (a_2 + 2a_3)/3 \rceil$ operations.

We observe that $(a_2 + a_3)/2 \leq (a_2 + 2a_3)/3$ holds if and only if $a_2 \leq a_3$ and that equality holds for $a_2 = a_3$. Since $\lceil \cdot \rceil$ is a monotone function, this implies that the total cost produced by the last part of the algorithm is $\max\{\lceil (a_2 + a_3)/2 \rceil, \lceil (a_2 + 2a_3)/3 \rceil\}$. ■

In particular, the length of the shuffle code computed by Greedy only depends on the signature of the input RTG G. In the remainder of this section, we prove that Greedy is optimal for outdegree-1 RTGs and therefore the formula in Lemma 3 actually computes the length of an optimal shuffle code.

Before we turn to the actual proof, we give an intuition of our proof strategy. Figure 5.20 depicts the idea behind the proof. Suppose we have an arbitrary outdegree-1 RTG G. We can now apply our Greedy algorithm, which will use Greedy(G) operations to transform G into a trivial RTG.

Alternatively, we can apply some arbitrary permutation instruction π to G, resulting in πG. Note that π does not have to be the permutation instruction that Greedy would choose next. Transforming πG into a trivial RTG using Greedy takes $\textsc{Greedy}(\pi G)$ instructions.

Now, if we can show that, regardless of which permutation instruction π we choose, directly applying Greedy is never worse than first using π and then Greedy, Greedy is optimal. Hence, we have to show that, for an arbitrary π, it is $\textsc{Greedy}(G) \leq \textsc{Greedy}(\pi G) + 1$. Or, equivalently, $\textsc{Greedy}(G) - \textsc{Greedy}(\pi G) \leq 1$.

Thus, it is crucial that we formally study the cost *difference* of Greedy for two given RTGs. With the following lemma we will do just that. As they are easier to handle, we will first look at PRTGs and then later generalize our findings to outdegree-1 RTGs in a straightforward manner.

Lemma 4 *Let G, G' be PRTGs with $\operatorname{sig}(G) = (X, a_2, a_3)$, $\operatorname{sig}(G') = (X', a_2', a_3')$ and $\textsc{Greedy}(G) - \textsc{Greedy}(G') \geq c$, and let $(\Delta_X, \Delta_2, \Delta_3) = \operatorname{sig}(G) - \operatorname{sig}(G')$. If $a_2 \geq a_3$, then $2\Delta_X + \Delta_2 + \Delta_3 \leq -2c + 1$. If $a_3 > a_2$, then $3\Delta_X + \Delta_2 + 2\Delta_3 \leq -3c + 2$.* □

Proof We assume that $\textsc{Greedy}(G) - \textsc{Greedy}(G') \geq c$ and start with the case that $a_2 \geq a_3$. By Lemma 3 and basic calculation rules for $\lceil\cdot\rceil$, we have the following.

$$\begin{aligned} \textsc{Greedy}(G) &= X + \lceil (a_2 + a_3)/2 \rceil \leq X + (a_2 + a_3 + 1)/2 \\ \textsc{Greedy}(G') &\geq X' + \lceil (a_2' + a_3')/2 \rceil \geq X + \Delta_X + (a_2 + a_3 + \Delta_2 + \Delta_3)/2 \end{aligned}$$

Therefore, their difference computes to

$$\begin{aligned} \textsc{Greedy}(G) - \textsc{Greedy}(G') &\leq -\Delta_X - (\Delta_2 + \Delta_3 - 1)/2 \\ &= -(2\Delta_X + \Delta_2 + \Delta_3 - 1)/2. \end{aligned}$$

By assumption, we thus have $-(2\Delta_X + \Delta_2 + \Delta_3 - 1)/2 \geq c$, or equivalently $2\Delta_X + \Delta_2 + \Delta_3 \leq -2c + 1$.

Now consider the case $a_3 > a_2$. By Lemma 3, we have the following.

$$\begin{aligned} \textsc{Greedy}(G) &= X + \lceil (a_2 + 2a_3)/3 \rceil \leq X + (a_2 + 2a_3 + 2)/3 \\ \textsc{Greedy}(G') &\geq X' + \lceil (a_2' + 2a_3')/3 \rceil \geq X + \Delta_X + (a_2 + 2a_3 + \Delta_2 + 2\Delta_3)/3 \end{aligned}$$

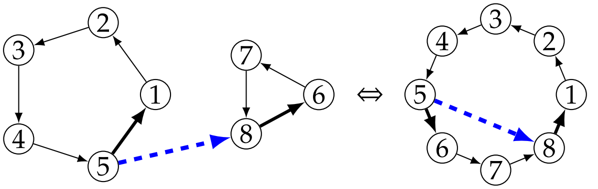


Figure 5.21: The transposition $\tau = (5\,8)$ acting on PRTGs. Affected edges are drawn thick. Read from left to right, the transposition is a merge; read from right to left, it is a split.

Similar to above, their difference computes to

$$\begin{aligned}\textsc{Greedy}(G) - \textsc{Greedy}(G') &\leq -\Delta_X - (\Delta_2 + 2\Delta_3 - 2)/3 \\ &= -(3\Delta_X + \Delta_2 + 2\Delta_3 - 2)/3.\end{aligned}$$

Similarly as above, by assumption we have $-(3\Delta_X + \Delta_2 + 2\Delta_3 - 2)/3 \geq c$, which is equivalent to $3\Delta_X + \Delta_2 + 2\Delta_3 \leq -3c + 2$. ■

Lemma 4 gives us necessary conditions for when the GREEDY solutions of two RTGs differ by some value c. These necessary conditions depend only on the difference of the two signatures. To study them more precisely, we define $\Psi_1(\Delta_X, \Delta_2, \Delta_3) = 2\Delta_X + \Delta_2 + \Delta_3$ and $\Psi_2(\Delta_X, \Delta_2, \Delta_3) = 3\Delta_X + \Delta_2 + 2\Delta_3$.

Next, we study the effect of a single transposition on these two functions. Let $G = (V, E)$ be a PRTG with $\text{sig}(G) = (X, a_2, a_3)$ and let τ be a transposition of two elements in V. We distinguish cases based on whether the swapped elements are in different connected components or not. In the former case, we say that τ is a *merge*, in the latter we call it a *split*; see Figure 5.21 for an illustration.

Merges. We start with the merge operations as they are a bit simpler. When merging two cycles of size s_1 and s_2, respectively, they are replaced by a single cycle of size $s_1 + s_2$. Note that removing the two cycles may decrease the values a_2 and a_3 of the signature by at most 2 in total. On the other hand, the new cycle can potentially increase one of these values

	0	1	2	3
0	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$	$(0,0,0)$
1		$(0,1,0)$	$(0,-1,1)$	$(1,0,-1)$
2			$(1,-2,0)$	$(1,-1,-1)$
3				$(1,1,-2)$

(a) Signature change $(\Delta_X, \Delta_2, \Delta_3)$.

	0	1	2	3
0	0	0	0	0
1		1	0	1
2			0	0
3				1

	0	1	2	3
0	0	0	0	0
1		1	1	1
2			1	0
3				0

(b) Values of Ψ_1 (left) and Ψ_2 (right).

Table 5.1: Signature changes and Ψ values for merges. Row and column are the cycle sizes modulo 4 before the merge.

by 1. The value X never decreases, and it increases by 1 if and only if $s_1 \bmod 4 + s_2 \bmod 4 \geq 4$.

Table 5.1a shows the possible signature changes $(\Delta_X, \Delta_2, \Delta_3)$ resulting from a merge. The entry in row i and column j shows the result of merging two cycles whose sizes modulo 4 are i and j, respectively. Table 5.1b shows the corresponding values of Ψ_1 and Ψ_2. Only entries with $i \leq j$ are shown, the remaining cases are symmetric.

Lemma 5 *Let G be a PRTG with* $\text{sig}(G) = (X, a_2, a_3)$ *and let τ be a merge. Then* $\textsc{Greedy}(G) \leq \textsc{Greedy}(\tau G)$. □

Proof Suppose we have $\textsc{Greedy}(\tau G) < \textsc{Greedy}(G)$. Then it is $\textsc{Greedy}(G) - \textsc{Greedy}(\tau G) \geq 1$ and by Lemma 4 either $\Psi_1 \leq -1$ or $\Psi_2 \leq -1$. However, Table 5.1b shows the values of Ψ_1 and Ψ_2 for all possible merges. In all cases it is $\Psi_1, \Psi_2 \geq 0$. A contradiction. ■

In particular, the lemma shows that merges never decrease the cost of the greedy solution, even if they were for free.

Splits. We now perform a similar analysis for splits. It is, however, obvious that splits indeed may decrease the cost of greedy solutions. In fact, we can always split cycles in a PRTG until it is trivial.

First, we study again the effect of splits on the signature change $(\Delta_X, \Delta_2, \Delta_3)$. Since a split is the inverse of a merge, we can essentially reuse Table 5.1a. If merging two cycles whose sizes modulo 4 are i and j, respectively, results in a signature change of $(\Delta_X, \Delta_2, \Delta_3)$, then, conversely, we can split a cycle whose size modulo 4 is $i + j$ into two cycles whose sizes modulo 4 are i and j, respectively, such that the signature change is $(-\Delta_X, -\Delta_2, -\Delta_3)$, and vice versa. Note that given a cycle whose size modulo 4 is s we have to look at all cells (i, j) with $i + j \equiv s \pmod 4$ to consider all the possible signature changes. Since Ψ_1, Ψ_2 are linear, negating the signature change also negates the corresponding value. Thus, we can reuse Table 5.1b for splits by negating each entry.

Lemma 6 *Let $G = (V, E)$ be a PRTG and let π be a cyclic shift of c vertices in V. Let further $(\Delta_X, \Delta_2, \Delta_3)$ be the signature change affected by π. Then $\Psi_1(\Delta_X, \Delta_2, \Delta_3) \geq -\lceil (c-1)/2 \rceil$ and $\Psi_2(\Delta_X, \Delta_2, \Delta_3) \geq -\lceil (3c-3)/4 \rceil$.* □

Proof We can write $\pi = \tau_{c-1} \circ \cdots \circ \tau_1$ as a product of $c - 1$ transpositions such that any two consecutive transpositions τ_i and τ_{i+1} affect a common element for $i = 1, \ldots, c - 1$.

Each transposition decreases Ψ_1 (or Ψ_2) by at most 1, but a decrease happens only for certain split operations. However, it is not possible to reduce Ψ_1 (or Ψ_2) with every single transposition since for two consecutive splits the second has to split one of the connected components resulting from the previous split.

To get an overview of the sequences of splits that reduce the value of Ψ_1 (or of Ψ_2) by 1 for each split, we consider the following transition graphs T_k for Ψ_k ($k = 1, 2$) on the vertex set $S = \{0, 1, 2, 3\}$. In the graph T_k there is an edge from i to j if there is a split that splits a component of size $i \bmod 4$ such that one of the resulting components has size $j \bmod 4$ and this split decreases Ψ_k by 1. The transition graphs T_1 and T_2 are shown in Figure 5.22.

For Ψ_1 the longest path in the transition graph has length 1. Thus, the value of Ψ_1 can be reduced at most every second transposition and thereby $\Psi_1(\Delta_X, \Delta_2, \Delta_3) \geq -\lceil (c-1)/2 \rceil$.

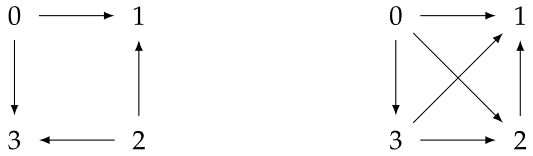


Figure 5.22: Transition graphs for Ψ_1 (left) and Ψ_2 (right).

For Ψ_2 the longest path has length 3 (vertex 1 has outdegree 0). Therefore, after at most three consecutive steps that decrease Ψ_2, there is one that does not. It follows that at least $\lfloor (c-1)/4 \rfloor$ operations do not decrease Ψ_2, and consequently at most $\lceil (3c-3)/4 \rceil$ operations decrease Ψ_2 by 1. Thus, $\Psi_2(\Delta_X, \Delta_2, \Delta_3) \geq -\lceil (3c-3)/4 \rceil$. ■

Since `permi5` performs a single cyclic shift and `permi23` is the concatenation of two cyclic shifts, Lemmas 4 and 6 can be used to show that no such operation may decrease the number of operations Greedy has to perform by more than 1.

Corollary 1 *Let G be a PRTG and let π be an operation, i.e., either a* `permi23` *or a* `permi5`*. Then* $\text{Greedy}(G) \leq \text{Greedy}(\pi G) + 1$. □

Proof Assume for a contradiction that $\text{Greedy}(G) > \text{Greedy}(\pi G) - 1$. By Lemma 4 we have that either $\Psi_1(\Delta_X, \Delta_2, \Delta_3) \leq -3$ or $\Psi_2(\Delta_X, \Delta_2, \Delta_3) \leq -4$.

We distinguish cases based on whether π is a `permi5` or a `permi23`. If π is a `permi5`, then it is a c-cycle with $c \leq 5$. By Lemma 6, we have that $\Psi_1(\Delta_X, \Delta_2, \Delta_3) \geq -2$ and $\Psi_2(\Delta_X, \Delta_2, \Delta_3) \geq -3$. This contradicts the above bounds from Lemma 4.

If π is a `permi23`, then it is a composition of a 2-cycle and a c-cycle with $c \leq 3$. According to Lemma 6, both cycles contribute at least -1 to Ψ_1, and at least -1 and -2 to Ψ_2. Therefore, we have $\Psi_1(\Delta_X, \Delta_2, \Delta_3) \geq -2$ and $\Psi_2(\Delta_X, \Delta_2, \Delta_3) \geq -3$. This is again a contradiction. ■

Using this corollary and an induction on the length of an optimal shuffle code, we show that Greedy is optimal for PRTGs. If no operation reduces the number of operations Greedy needs by more than 1, why not use the operation suggested by Greedy?

Theorem 1 *Let G be a PRTG. An optimal shuffle code for G takes* Greedy(G) *operations. Algorithm* Greedy *computes an optimal shuffle code in linear time.*□

Proof The proof is by induction on the overall length of an optimal shuffle code. Clearly, Greedy computes optimal shuffle codes for all instances that have a shuffle code of length 0.

Assume that G admits an optimal shuffle code of length $k + 1$. We show that Greedy$(G) = k + 1$. First of all, note that Greedy$(G) \geq k + 1$ as it computes a shuffle code of length Greedy(G). Let $\pi_1, \ldots, \pi_{k+1}$ be a shuffle code for G. Then obviously $\pi_{k+1}G$ admits an optimal shuffle code of length k, and therefore Greedy$(\pi_{k+1}G) = k$ by our inductive assumption. Corollary 1 implies Greedy$(G) \leq$ Greedy$(\pi_{k+1}G) + 1 = k + 1$; the induction hypothesis is proved. Hence, algorithm Greedy indeed computes a correct, and thus optimal, shuffle code.

Also, it computes this optimal shuffle code in linear time, as we can see as follows. The first step (completing directed paths into cycles) is clearly linear. In each iteration of Greedy, one of the steps 2–5 is active and Greedy generates one instruction. This instruction creates at least one loop. As Greedy never touches vertices with loops again and stops when the RTG is trivial, the number of vertices n is an upper bound for the number of iterations. As splitting a cycle only takes constant time, each iteration takes constant time as well. Hence, Greedy runs in linear time. ■

Moreover, since merge operations may not decrease the cost of Greedy and any PRTG that can be formed from the original outdegree-1 RTG G by inserting edges can be obtained from the PRTG G' formed by Greedy and a sequence of merge operations, it follows that the length of an optimal shuffle for G is Greedy(G').

Lemma 7 *Let G be an outdegree-1 RTG and let G' be the PRTG formed by completing each directed path into a directed cycle. Then the length of an optimal shuffle code of G is* Greedy(G'). □

Proof Assume $\pi_1, \ldots, \pi_k$ is an optimal shuffle code for G. Of course, applying $\pi = \pi_k \circ \cdots \circ \pi_1$ to G maps every value of G somewhere, that is, $\pi_1, \ldots, \pi_k$ is actually an optimal shuffle code for some instance G'' that consists of a disjoint union of directed cycles and contains G as a subgraph. It is not hard to see that G'' can be obtained from G' by a sequence of merge operations $\tau_1, \ldots, \tau_t$, i.e., $G'' = \tau_t \circ \cdots \circ \tau_1 G'$. Lemma 5 implies that $\textsc{Greedy}(G') \leq \textsc{Greedy}(\tau_1 G') \leq \cdots \leq \textsc{Greedy}(\tau_t \circ \cdots \circ \tau_1 G') = \textsc{Greedy}(G'') = k$, where the last equality follows from Theorem 1, the optimality of Greedy for PRTGs. ■

By combining Theorem 1 and Lemma 7, we obtain the main result of this section.

Theorem 2 *Let G be an outdegree-1 RTG. Then an optimal shuffle code for G requires* $\textsc{Greedy}(G)$ *operations.* Greedy *computes such a shuffle code in linear time.* □

Remark. Recall that the combination of `permi5` and `permi23` enables us to express any permutation of up to five elements. We define the size of a permutation to be the number of elements affected by the permutation. Viewed this way, we can use Greedy to solve a more general problem than computing shuffle code: it decomposes a given permutation into a shortest product of permutations of maximum size 5.

5.3.4. A Heuristic for Finding Copy Sets

We now turn to the general case, i.e., our input is not an outdegree-1 RTG, but an arbitrary RTG. Following the idea from Section 5.3.2, we must now pick a copy set. To recap: the idea behind a copy set is that, at each vertex with more than one outgoing edge, we pick one outgoing edge, remove the other outgoing edges from the graph and put them into the copy set. By doing that, the graph becomes an outdegree-1 RTG and is suitable for Greedy. We then implement all edges in the copy set with copy operations, adapting their source vertices as described in Section 5.3.2.

More formally, a *copy set* of an RTG $G = (V, E)$ is a set $C \subseteq E$ such that $G - C = (V, E - C)$ is an outdegree-1 RTG. It is always $|C| = \sum_{v \in V} \max\{\deg^+(v) - 1, 0\}$. We denote by $\mathcal{C}(G)$ the set of all copy sets of G.

Once we have chosen a copy set $C \in \mathcal{C}(G)$ we can, by Theorem 2, compute an optimal shuffle code for $G - C$ with the greedy algorithm and we can compute its length according to Lemma 3. We now propose a simple heuristic for finding a good copy set fast. After that, in Section 5.3.5, we propose an approach to find an *optimal* copy set $C \in \mathcal{C}(G)$, i.e., a copy set such that the outdegree-1 RTG $G - C$ admits a shortest shuffle code.

Our heuristic is based on two ideas: (i) as our permutation instructions are good at handling cycles, we always preserve existing cycles in the RTG, and (ii) as our permutation instructions are most useful when implementing large outdegree-1 RTGs, we try to choose our copy set so that the resulting outdegree-1 RTG is as large as possible.

We directly translate these two ideas into the following two simple rules:

1. We always keep cycles. Hence, at each vertex that is part of a cycle, we keep the outgoing edge that is part of the cycle, and put all other outgoing edges into the copy set.
2. We prefer creating large outdegree-1 RTGs. Hence, at each vertex that is not part of a cycle, we keep the edge that is part of the longest path starting at that vertex.

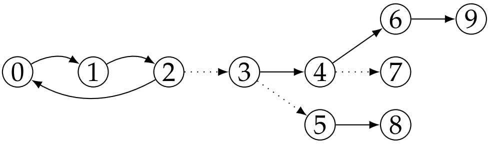


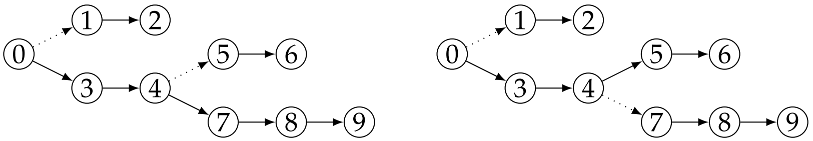
Figure 5.23: An RTG after the heuristic has chosen a copy set (depicted as dotted edges).

Figure 5.23 shows an example RTG, for which the heuristic has chosen a copy set (shown as dotted edges). Because of rule 1, we put edge $(2, 3)$ into the copy set as edge $(2, 0)$ is part of the existing cycle (0 1 2). After doing this for all cycles, the remaining components are either cycles (which we can keep) or trees.

For the tree-shaped RTGs, we apply rule 2. Hence, at vertex 3 we keep the edge $(3, 4)$ as the path from 3 to 9 is longer than the path from 3 to 8 (rule 2). Rule 2 also applies for vertex 4, hence we keep $(4, 6)$, whereas $(4, 7)$ becomes part of the copy set.

Time Complexity. Our heuristic has running-time complexity $O(n)$ for an RTG with n vertices. As each vertex has at most one incoming edge, n is also an upper bound for the number of edges. For step 1 of our heuristic, we need to find all cycles in the RTG. We can do this in $O(n)$ time using, e.g., Tarjan's SCC algorithm [Tar72]. Additionally, for the tree-shaped RTGs, we need to determine the longest path starting at each vertex, which we can do in linear time using a depth-first search. In total, we have linear worst-case complexity.

Quality. We will analyze the quality of the code generated by the heuristic empirically in Section 5.4. However, as the following examples show, both rules of the heuristic can lead to finding non-optimal copy sets.



(a) Heuristic solution, requires 5 instructions.

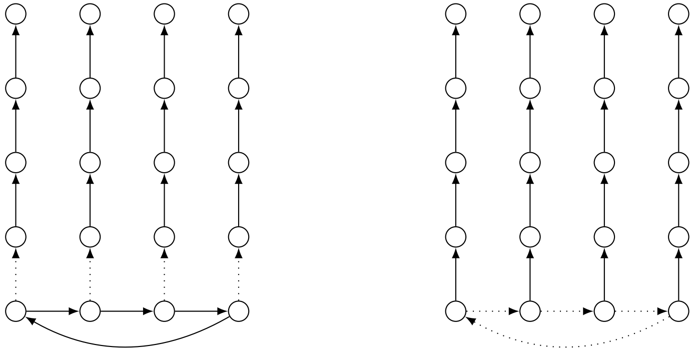
(b) Optimal solution, requires 4 instructions.

Figure 5.24: Comparison of copy set chosen by heuristic with optimal copy set. We show copy sets with dotted edges.

Figure 5.24 shows the smallest RTG known to the author, for which the heuristic computes a non-optimal copy set. As depicted in Figure 5.24a, due to rule 2, the heuristic puts edges $(0, 1)$ and $(4, 5)$ into the copy set in order to preserve the longest path 0 to 9 in the RTG. The remaining three components require three instructions, hence, including two copies, we require five instructions in total.

However, as shown in Figure 5.24b, an optimal copy set, e.g., $\{(0, 1), (4, 7)\}$, can reduce the number of required instructions to four. Here, the remaining components (paths of lengths 2, 3 and 5) fit perfectly into a `permi23` and a `permi5` instruction. Hence, rule 2 of the heuristic can lead to a non-optimal choice for the edges in the copy set.

Figure 5.25 shows that also rule 1 of the heuristic can lead to a non-optimal copy set. Due to the size of the RTG, we omit vertex numbers. In



(a) Heuristic solution, requires 9 instructions.

(b) An optimal solution, requires 8 instructions.

Figure 5.25: Comparison of copy set chosen by heuristic with an optimal copy set.

Figure 5.25a, we see how rule 1 of the heuristic preserves the cycle, hence we put the 4 edges leaving the cycle into the copy set. This leaves us with 5 components of size 4, for which Greedy needs 5 `permi5` instructions. Thus, we need 9 instructions in total.

However, as Figure 5.25b shows, it is beneficial to break the cycle. If we put all edges that are part of the cycle into the copy set, we can implement the remaining 4 paths of size 5 using just 4 `permi5` instructions. Hence, in total, we need 8 instructions, one instruction less than with the copy set found by the heuristic.

This raises the question of how to find optimal copy sets for RTGs. The presented examples suggest that small changes to the heuristic will probably not be sufficient to achieve optimality, as both ideas that we based our heuristic on can lead to suboptimal solutions. Hence, an algorithm to find optimal copy sets will likely have an entirely different structure.

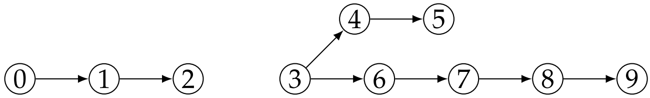


Figure 5.26: Example where a locally optimal copy set is not globally optimal. If we just look at the right component, the copy set $C_1 = \{(3,6)\}$ is locally optimal; we then need 3 instructions to implement the RTG, which is minimal. However, C_1 is not globally optimal: if we must implement a path of length 3 (shown on the left side) at the same time, we need 4 instructions. In this case, $C_2 = \{(3,4)\}$ is a globally optimal copy set for the right component; we then need 3 instructions for the whole RTG.

5.3.5. Finding Optimal Copy Sets

We now want to find an optimal copy set. Thus, for an RTG G, we seek a copy set $C \in \mathcal{C}(G)$ that minimizes the cost function $\textsc{Greedy}(G - C) = X + \max\{\lceil(a_2 + a_3)/2\rceil, \lceil(a_2 + 2a_3)/3\rceil\}$, where (X, a_2, a_3) is the signature of $G - C$. We call such a copy set *optimal*.

Before we study this problem formally, we give an intuition for the idea behind our approach. We will find optimal copy sets using dynamic programming. The idea is that for some RTG G, we compute optimal copy sets for progressively larger subgraphs of G until we have found an optimal copy set for G. For example, suppose G is a tree-shaped RTG with root vertex v, we would like to compute optimal copy sets for all tree RTGs rooted at the children of v and then combine them to get an optimal copy set for G.

Unfortunately, it is, in general, not possible to determine an optimal copy set locally. This is because the cost function $\textsc{Greedy}(G - C)$ strongly depends on a_2 and a_3, the number of 2-cycles and 3-cycles, of the *complete* RTG $G - C$.

Figure 5.26 shows an example where a locally optimal copy set is not globally optimal. Here, we have a disconnected RTG G consisting of a path of length 3 (shown on the left) and a tree-shaped component G' (shown on the right). An optimal copy set for G' in isolation is $C_1 = \{(3,6)\}$. Then,

$G' - C_1$ leaves a 4-cycle and a 3-cycle, so we need 3 instructions in total, which is minimal.

However, if we look at the complete RTG, it is better to choose $C_2 = \{(3,4)\}$ as a copy set for G'. Locally, it does not make a difference: we would still need 3 instructions for G' alone. Yet, globally, we have a surplus of 3-cycles because of the path of size 3. Hence, it is beneficial to choose C_2 as copy set for G' to create a local surplus of 2-cycles. Globally, this results in an equal number of 2-cycles and 3-cycles, so Greedy can then match the pair and our overall costs are minimal.

Hence, we keep track of optimal copy sets for all possible combinations of numbers of remaining 2-cycles and 3-cycles. This guarantees that, at the end with a global view, we can choose the optimal copy set for the whole RTG. In the following, we will formalize this idea.

Minimizing Greedy$(G - C)$ is equivalent to minimizing the function Greedy$'$ where we drop the rounding expressions:

$$\textsc{Greedy}'(G-C) = X+\max\{\frac{a_2+a_3}{2}, \frac{a_2+2a_3}{3}\} = \begin{cases} X + \frac{a_2}{2} + \frac{a_3}{2} & \text{if } a_2 \geq a_3 \\ X + \frac{a_2}{3} + \frac{2a_3}{3} & \text{if } a_2 < a_3 \end{cases}$$

To keep track of which case is used for evaluating Greedy$'$, we define $\mathrm{diff}(G - C) = a_2 - a_3$ and compute for each of the two function parts and every possible value d a copy set C_d with $\mathrm{diff}(G - C_d) = d$ that minimizes that function.

More formally, we define $\mathrm{cost}^1(G - C) = X + \frac{1}{2}a_2 + \frac{1}{2}a_3$ and $\mathrm{cost}^2(G - C) = X + \frac{1}{3}a_2 + \frac{2}{3}a_3$. We then seek two tables $T^1_G[\cdot], T^2_G[\cdot]$, such that $T^i_G[d]$ is the smallest cost $\mathrm{cost}^i(G - C)$ that can be achieved with a copy set $C \in \mathcal{C}(G)$ with $\mathrm{diff}(G - C) = d$.

We observe that $T^i_G[d] = \infty$ for $d < -n$ and for $d > n$. The following lemma shows how to compute the length of an optimal shuffle code from these two tables.

Lemma 8 *Let $G = (V, E)$ be an RTG. The length of an optimal shuffle code for G is* $\sum_{v\in V} \max\{\deg^+(v) - 1, 0\} + \min\{\min_{d\geq 0}\lceil T^1_G[d]\rceil, \min_{d<0}\lceil T^2_G[d]\rceil\}$. □

Proof Let $m = \sum_{v \in V} \max\{\deg^+(v) - 1, 0\}$. Consider an optimal normalized shuffle code for G, which, according to Lemma 2, consists of a copy set $C \subseteq E$ and a sequence of k permutation operations, i.e., the length of the shuffle code is $m + k$. Let (X, a_2, a_3) denote the signature of $G - C$ and let $d = a_2 - a_3$.

If $a_2 \geq a_3$, or equivalently $d \geq 0$, then according to Theorem 2, we have $k = \textsc{Greedy}(G-C) = X + \lceil (a_2+a_3)/2 \rceil = \lceil X + (a_2+a_3)/2 \rceil = \lceil \mathrm{cost}^1(G-C) \rceil$, and therefore the length of the shuffle code is at most $m + \lceil T^1_G[d] \rceil$.
If $a_2 < a_3$, i.e., if $d < 0$, then we have $k = \textsc{Greedy}(G - C) = X + \lceil (a_2 + 2a_3)/3 \rceil = \lceil X + (a_2 + 2a_3)/3 \rceil = \lceil \mathrm{cost}^2(G - C) \rceil$, and therefore the length of the shuffle code is at most $m + \lceil T^2_G[d] \rceil$.
In either case the length of the shuffle code is bounded by the expression given in the statement of the theorem.

Conversely, assume that the minimum of the expression is obtained for some value $T^i_G[d]$.

If $d \geq 0$, there exists a copy set C such that $\mathrm{sig}(G - C) = (X, a_2, a_3)$ and $\textsc{Greedy}(G - C) = \lceil \mathrm{cost}^1(G - C) \rceil$ is at most $\lceil T^1_G[d] \rceil$. Then, the shuffle code defined by C and Greedy applied to $G - C$ has length at most $m + \lceil T^1_G[d] \rceil$.
If $d < 0$, there exists a copy set C such that $\mathrm{sig}(G - C) = (X, a_2, a_3)$ and $\textsc{Greedy}(G - C) = \lceil \mathrm{cost}^2(G - C) \rceil$ is at most $\lceil T^2_G[d] \rceil$. Then, the shuffle code defined by C and Greedy applied to $G - C$ has length at most $m + \lceil T^2_G[d] \rceil$. ■

In the following, we show how to compute for an RTG G a table $T_G[\cdot]$ with

$$T_G[d] = \min_{\substack{C \in \mathcal{C}(G) \\ \mathrm{diff}(G-C)=d}} \mathrm{cost}(G - C)$$

for an arbitrary cost function $\mathrm{cost}(G - C) = c(\mathrm{sig}(G - C))$, where c is a linear function. We do this in several steps depending on whether G is disconnected, is a tree, or is connected and contains a cycle. Before we continue, we introduce several preliminaries to simplify the following calculations. We denote by P_s a directed path on s vertices.

Definition 3 A map f that assigns a value to an outdegree-1 RTG is *signature-linear* if there exists a linear function $g \colon \mathbb{R}^3 \to \mathbb{R}$ such that $f(G) = g(\mathrm{sig}(G))$ for every outdegree-1 RTG G. For a signature-linear function f, $\Delta_f(s) = f(P_{s+1}) - f(P_s)$ is the *correction term*. □

Note that both cost $= c \circ \mathrm{sig}$ and diff $= d \circ \mathrm{sig}$ with $d(X, a_2, a_3) = a_2 - a_3$ are signature-linear. The correction term $\Delta_f(s)$ describes the change of f when the size of one connected component is increased from s to $s + 1$.

Lemma 9 *Let f be a signature-linear function. Then the following hold:*

(i) $f(G_1 \cup G_2) = f(G_1) + f(G_2)$ *for disjoint outdegree-1 RTGs G_1, G_2,*

(ii) Let $G = (V, E)$ be an outdegree-1 RTG and let $v \in V$ with in-degree 0. *Denote by s the size of the connected component containing v and let $G^+ = (V \cup \{u\}, E \cup \{(u, v)\})$ where u is a new vertex. Then $f(G^+) = f(G) + \Delta_f(s)$.* □

Proof For Statement (i) observe that $\mathrm{sig}(G_1 \cup G_2) = \mathrm{sig}(G_1) + \mathrm{sig}(G_2)$; then the statement follows from the signature-linearity of f.

For Statement (ii) observe that by adding u, we replace a connected component of size s with one of size $s + 1$. Thus $\mathrm{sig}(G^+) = \mathrm{sig}(G) - \mathrm{sig}(P_s) + \mathrm{sig}(P_{s+1})$. The statement follows from the signature-linearity of f and the definition of $\Delta_f(s)$. ■

Note that $\Delta_f(s) = \Delta_f(s + 4)$ for all values of s and hence it suffices to know the size of the enlarged component modulo 4.

The main idea for computing table $T_G[\cdot]$ by dynamic programming is to decompose G into smaller edge-disjoint subgraphs $G = G_1 \cup \cdots \cup G_k$ such that the copy sets of G can be constructed from copy sets for each of the G_i.

We call such a decomposition *proper partition* if for every vertex v of G there exists an index i such that G_i contains all outgoing edges of v. Let $G_1, \ldots, G_k$ be a proper partition of G and let $\mathcal{C}_i \subseteq \mathcal{C}(G_i)$ for $i = 1, \ldots, k$. We define $\mathcal{C}_1 \otimes \cdots \otimes \mathcal{C}_k = \{C_1 \cup \cdots \cup C_k \mid C_i \in \mathcal{C}_i, i = 1, \ldots, k\}$. It is not hard to see that $\mathcal{C}(G_1 \cup \cdots \cup G_k) = \mathcal{C}(G_1) \otimes \cdots \otimes \mathcal{C}(G_k)$.

5.3.5.1. Disconnected RTGs

We start with the case that G is disconnected and consists of connected components $G_1, \ldots, G_k$, which form a proper partition of G. Our intuition

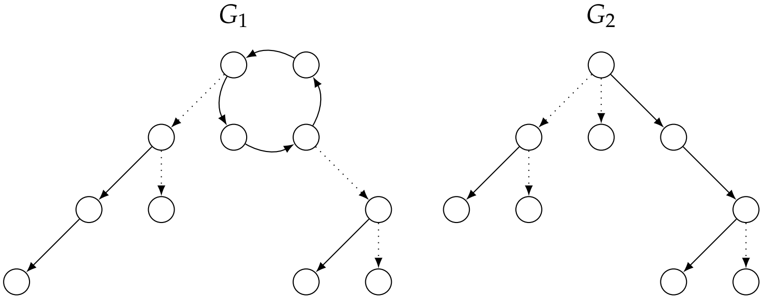


Figure 5.27: A disconnected RTG with 2 components. Dotted edges are in the copy set. If we have copy sets C_1 and C_2 for the components, $C_1 \cup C_2$ is a copy set for the complete RTG.

is that if we have a copy set C_i for each of the components G_i, their union $\bigcup_i C_i$ forms a copy set for G. Figure 5.27 illustrates this idea. We can then find an optimal copy set for the overall RTG by choosing the best combination of copy sets of its components. In the following we will see that this intuition is indeed correct.

When studying this problem formally, the main issue is to keep track of diff and cost. For an RTG G, we define $\mathcal{C}(G; d) = \{C \in \mathcal{C}(G) \mid \text{diff}(G - C) = d\}$. By Lemma 9 (i) and the signature-linearity of diff, if $C_i \in \mathcal{C}(G_i; d_i)$ for $i = 1, 2$, then $C_1 \cup C_2 \in \mathcal{C}(G_1 \cup G_2; d_1 + d_2)$. This leads to the following lemma.

Lemma 10 *Let G be an RTG and let G_1, G_2 be vertex-disjoint RTGs. Then*

(i) $\mathcal{C}(G) = \bigcup_d \mathcal{C}(G; d)$ *and*
(ii) $\mathcal{C}(G_1 \cup G_2; d) = \bigcup_{d'} (\mathcal{C}(G_1; d') \otimes \mathcal{C}(G_2; d - d'))$. □

Proof Equation (i) follows immediately from the definition of $\mathcal{C}(G; d)$.

For Equation (ii) observe that if $C_1 \in \mathcal{C}(G_1; d')$ and $C_2 \in \mathcal{C}(G_2; d - d')$, then $C = C_1 \cup C_2$ is a copy set of G and by Lemma 9 (i) $\text{diff}(G - C) = \text{diff}((G_1 - C_1) \cup (G_2 - C_2)) = \text{diff}(G_1 - C_1) + \text{diff}(G_2 - C_2) = d' + d - d' = d$, and hence $C_1 \cup C_2 \in \mathcal{C}(G; d)$.

Conversely, if $C \in \mathcal{C}(G; d)$, define $C_i = C \cap E_i$ where E_i is the edge set of G_i for $i = 1, 2$. Let $d' = \mathrm{diff}(G_1 - C_1)$. As above, it follows from Lemma 9 (i) that $d = \mathrm{diff}(G - C) = \mathrm{diff}(G_1 - C_1) + \mathrm{diff}(G_2 - C_2) = d' + \mathrm{diff}(G - C)$, and hence $\mathrm{diff}(G - C) = d - d'$. Thus $C \in \mathcal{C}(G_1; d') \otimes \mathcal{C}(G_2; d - d')$. ■

By further exploiting the signature-linearity of cost, we also get $\mathrm{cost}((G_1 \cup G_2) - (C_1 \cup C_2)) = \mathrm{cost}(G_1 - C_1) + \mathrm{cost}(G_2 - C_2)$, allowing us to compute the cost of copy sets formed by the union of copy sets of vertex-disjoint graphs.

Lemma 11 *Let G_1, G_2 be two vertex-disjoint RTGs and let $G = G_1 \cup G_2$. Then $T_G[d] = \min_{d'}\{T_{G_1}[d'] + T_{G_2}[d - d']\}$.* □

Proof Applying the definition of $T_G[\cdot]$ as well as Lemma 10 (ii) and Lemma 9 (i) yields

$$\begin{aligned}
T_G[d] &= \min_{C \in \mathcal{C}(G;d)} \mathrm{cost}(G - C) \\
&= \min_{C \in \bigcup_{d'}(\mathcal{C}(G_1;d') \otimes \mathcal{C}(G_2;d-d'))} \mathrm{cost}(G - C) \\
&= \min_{d'} \left\{ \min_{C \in \mathcal{C}(G_1;d') \otimes \mathcal{C}(G_2;d-d')} \mathrm{cost}(G - C) \right\} \\
&= \min_{d'} \left\{ \min_{C_1 \in \mathcal{C}(G_1;d')} \mathrm{cost}(G_1 - C_1) + \min_{C_2 \in \mathcal{C}(G_2;d-d')} \mathrm{cost}(G_2 - C_2) \right\} \\
&= \min_{d'} \{T_{G_1}[d'] + T_{G_2}[d - d']\}.
\end{aligned}$$
■

By iteratively applying Lemma 11, we compute $T_G[\cdot]$ for a disconnected RTG G with an arbitrary number of connected components.

Lemma 12 *Let G be an RTG with n vertices and connected components $G_1, \ldots, G_k$. Given the tables $T_{G_i}[\cdot]$ for $i = 1, \ldots, k$, the table $T_G[\cdot]$ can be computed in $O(n^2)$ time.* □

Proof Let n_i denote the number of vertices of G_i. For two graphs H_1 and H_2 with h_1 and h_2 vertices, respectively, computing $T_{H_1 \cup H_2}[\cdot]$ according to Lemma 11 takes time $O(h_1 \cdot h_2)$ and the table size is $O(h_1 + h_2)$. Thus, iteratively combining the table for G_{i+1} with the table for $\bigcup_{j=1}^{i} G_j$ takes time $O(\sum_{i=1}^{k-1} n_{i+1} \sum_{j=1}^{i} n_j)$. It is $\sum_{i=1}^{k-1} n_{i+1} \sum_{j=1}^{i} n_j \leq \sum_{i=1}^{k-1} n_{i+1} n = n \sum_{i=1}^{k-1} n_{i+1} \leq n^2$. Hence, the running time is $O(n^2)$. ■

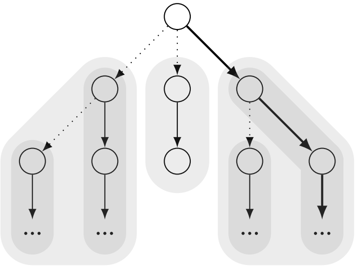


Figure 5.28: Finding a copy set for a tree RTG G. At each inner vertex, we can keep exactly one outgoing edge; all others must go into the copy set C. Hence, in each component of $G - C$, there is a path, which we call *root path*, from the root vertex of the component to one of its leaves. We draw the root path of G with thick edges.

5.3.5.2. Tree RTGs

For a tree RTG G, our overall strategy is to compute $T_G[\cdot]$ in a bottom-up fashion. Hence, we start at the leaves and at each inner vertex v we compute the table for the subtree rooted at v by combining the already computed tables of v's children.

Figure 5.28 illustrates our idea of how to find a copy set for a tree RTG G. The main insight is that at each inner vertex, we can keep exactly one of the outgoing edges and we must put all others into the copy set C. In the example shown, we choose to keep the rightmost edge.

This construction implies that for a tree RTG G with root vertex r and matching copy set C, each component of $G - C$ contains exactly one path, which we call *root path*, from the component's root vertex to one of the leaves. If there were a path from the root to another leaf, there would have to exist a vertex with multiple outgoing edges, which would mean C is not a valid copy set.

As we use a bottom-up approach, we assume that we have already found a copy set for each of the subtrees rooted at the children of r. Choosing an outgoing edge for r prolongs the root path of the respective component

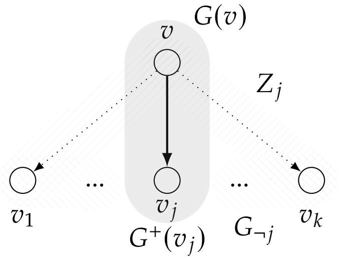


Figure 5.29: Nomenclature used for tree RTGs in the formalization.

by 1. By trying out all possible outgoing edges of the root vertex r, we find the optimal copy set for G.

The direction of the edges naturally defines a unique root vertex r that has no incoming edges and we consider G as a rooted tree. Figure 5.29 illustrates the nomenclature used in the following. For a vertex v, we denote by $G(v)$ the subtree of G with root v. Let v be a vertex with children $v_1, \ldots, v_k$. What does a copy set C of $G(v)$ look like?

Clearly, $G(v) - C$ contains precisely one of the outgoing edges of v, say (v, v_j). Then $Z_j = \{(v, v_i) \mid i \neq j\} \subseteq C$. The graph $G(v) - Z_j$ has connected components $G(v_i)$ for $i \neq j$, whose union we denote $G_{\neg j}$, and one additional connected component $G^+(v_j)$ that is obtained from $G(v_j)$ by adding the vertex v and the edge (v, v_j). This forms a proper partition of $G(v) - Z_j$.

As above, we decompose the copy set $C - Z_j$ further into a union of a copy set $C_{\neg j}$ of $G_{\neg j}$ and a copy set C_j of $G^+(v_j)$. Graph $G_{\neg j}$ is disconnected and can be handled as above. Note that the only child of the root of $G^+(v_j)$ is v_j and hence C_j is a copy set of $G(v_j)$.

For expressing the cost and difference measures for copy sets of $G^+(v_j)$ in terms of copy sets of $G(v_j)$, we use the correction terms Δ_{cost} and Δ_{diff}. By Lemma 9 (ii), $\text{diff}(G^+(v_j) - C_j) = \text{diff}(G(v_j) - C_j) + \Delta_{\text{diff}}(s)$, where s is the size of the *root path* $P(v_j, C_j)$ of $G(v_j) - C_j$, i.e., the size of the connected component of $G(v_j) - C_j$ containing v_j. An analogous statement holds for cost. More precisely, it suffices to know s modulo 4.

Therefore, we further decompose our copy sets as follows, which allows us to formalize our discussion.

Definition 4 For a tree RTG G with root v and children $v_1, \dots, v_k$, we define $\mathcal{C}(G; d, s) = \{C \in \mathcal{C}(G; d) \mid |P(v, C)| \equiv s \pmod 4\}$. We further decompose these by $\mathcal{C}(G; d, s, j) = \{C \in \mathcal{C}(G; d, s) \mid (v, v_j) \notin C\}$, according to which outgoing edge of the root is not in the copy set. □

Lemma 13 *Let G be a tree RTG with root v and children $v_1, \dots, v_k$ and for a fixed vertex v_j, $1 \leq j \leq k$, let $G^+(v_j)$ be the subgraph of G induced by the vertices in $G(v_j)$ together with v. Let further $G_{\neg j} = \bigcup_{i=1, i \neq j}^{k} G(v_i)$ and $Z_j = \{(v, v_i) \mid i \neq j\}$. Then*

(i) $\mathcal{C}(G; d) = \bigcup_{s=0}^{3} \mathcal{C}(G; d, s)$ *and* $\mathcal{C}(G; d, s) = \bigcup_{j=1}^{k} \mathcal{C}(G; d, s, j)$.

(ii) $\mathcal{C}(G^+(v_j); d, s) = \mathcal{C}(G(v_j); d - \Delta_{\text{diff}}(s), s - 1)$.

(iii) $\mathcal{C}(G; d, s, j) = \bigcup_{d'} \big(\mathcal{C}(G_{\neg j}; d') \otimes \mathcal{C}(G^+(v_j); d - d', s) \otimes \{Z_j\}\big)$. □

Proof Statements (i) follow immediately from the definitions of $\mathcal{C}(G; d, s)$ and $\mathcal{C}(G; d, s, j)$.

We continue with Statement (ii). Since v in $G^+(v_j)$ has only one child v_j, the edge (v, v_j) is not in any copy set of $G^+(v_j)$. Therefore, the copy sets of $\mathcal{C}(G^+(v_j))$ and $\mathcal{C}(G(v_j))$ are in one-to-one correspondence.

We need to understand how the partition into copy sets with difference measure d and root path length s (modulo 4) respects this bijection. Let s be the root path size of $G^+(v_j) - C$ for a copy set $C \in \mathcal{C}(G^+(v_j))$. Obviously, $|P(G(v_j) - C)| = |P(G^+(v_j) - C)| - 1 = s - 1$. Moreover, going from $G^+(v_j) - C$ to $G(v_j) - C$ replaces a connected component of size s with one of size $s - 1$. Therefore $\text{sig}(G(v_j) - C) = \text{sig}(G^+(v_j) - C) - \text{sig}(P_s) + \text{sig}(P_{s+1})$.

By the signature-linearity of diff, we have $\text{diff}(G(v_j) - C) = \text{diff}(G^+(v_j) - C) - \Delta_{\text{diff}}(s)$. Note further that $\Delta_{\text{diff}}(s) = \Delta_{\text{diff}}(s + 4)$ for every value of s, and hence it suffices to know $s \mod 4$. Overall, it follows that a copy set $C \in \mathcal{C}(G^+(v_j); d, s)$ is a copy set of $G(v_j)$ with difference measure $\text{diff}(G^+(v_j) - C) - \Delta_{\text{diff}}(s)$ and root path size modulo 4 being $s - 1$. Thus $C \in \mathcal{C}(G(v_j), d - \Delta_{\text{diff}}(s), s - 1)$. And conversely $C \in \mathcal{C}(G(v_j), d - \Delta_{\text{diff}}(s), s - 1)$ satisfies $C \in \mathcal{C}(G^+(v_j); d, s)$.

Next, we consider Statement (iii). First observe that the copy sets C of G whose root path starts with (v, v_j) are exactly those copy sets of G that contain all edges in Z_j. These sets correspond bijectively to copy sets of $G - Z_j$. Thus $\mathcal{C}(G; d, s, j) = \mathcal{C}(G - Z_j; d, s) \otimes \{Z_j\}$.

Observe that $G - Z_j = G_{\neg j} \cup G^+(v_j)$ is a proper partition of $G - Z_j$. Furthermore, the root path of any copy set of this graph lies in $G^+(v_j)$. Therefore Lemma 10 (ii) implies that $\mathcal{C}(G - Z_j; d, s) = \bigcup_{d'}(\mathcal{C}(G_{\neg j}; d') \otimes (\mathcal{C}(G(v_j)^+; d - d', s)$. Combining this with the previously derived description of $\mathcal{C}(G; d, s, j)$ yields Statement (iii). ■

To make use of this decomposition of copy sets, we extend our table T with an additional parameter s to keep track of the size of the root path modulo 4. We call the resulting table $\tilde{T}$. More formally, $\tilde{T}_v[d, s] = \min_{C \in \mathcal{C}(G(v); d, s)} \text{cost}(G(v) - C)$. It is not hard to see that $T_G[\cdot]$ can be computed from $\tilde{T}_r[\cdot, \cdot]$ for the root r of a tree RTG G.

Lemma 14 *Let G be a tree RTG with root r. Then $T_G[d] = \min_s \tilde{T}_r[d, s]$.* □

Proof Using the definitions of $T_G[\cdot]$ and $\tilde{T}_r[\cdot, \cdot]$, we obtain

$$T_G[d] = \min_{C \in \mathcal{C}(G;d)} \text{cost}(G - C) = \min_{s \in \{0,\dots,3\}} \min_{C \in \mathcal{C}(G;d,s)} \text{cost}(G - C) = \min_{s \in \{0,\dots,3\}} \tilde{T}_r[d, s].$$

■

To compute $\tilde{T}_v[\cdot, \cdot]$ in a bottom-up fashion, we exploit the decompositions from Lemma 13 and the fact that we can update the cost function from $G(v_j) - C_j$ to $G^+(v_j) - C_j$ using the correction term Δ_{cost}. The proof is similar to that of Lemma 11 but more technical.

Lemma 15 *Let G be a tree RTG, let v be a vertex of G with children $v_1, \dots, v_k$, and let $G(v_i) = (V_i, E_i)$ for $i = 1, \dots, k$. Then with $G_{\neg j} = (V_{\neg j}, E_{\neg j}) = \bigcup_{i=1, i \neq j}^{k} G(v_i)$ it is*

$$\begin{aligned}\tilde{T}_v[d, s] = \min_{j \in \{1,\dots,k\}} \min_{d'} T_{G_{\neg j}}[d'] + \tilde{T}_{v_j}[d - d' - \Delta_{\text{diff}}(s), (s - 1) \bmod 4] \\ + \Delta_{\text{cost}}(s).\end{aligned}$$

□

Proof According to the definition of $\tilde{T}_v[d,s]$ and Lemma 13 (i), we find that

$$\tilde{T}_v[d,s] = \min_{C \in \mathcal{C}(G;d,s)} \operatorname{cost}(G - C) = \min_j \min_{C \in \mathcal{C}(G;d,s,j)} \operatorname{cost}(G - C) \tag{5.2}$$

Using Lemma 13 (iii) yields

$$\min_{C \in \mathcal{C}(G;d,s,j)} \operatorname{cost}(G - C) = \min_{d'} \min_{\substack{X \in \mathcal{C}(G_{\neg j};d') \\ Y \in \mathcal{C}(G^+(v_j);d-d',s)}} \operatorname{cost}(G - X - Y - Z_j). \tag{5.3}$$

Note that $G - Z_j = G_{\neg j} \cup G^+(v_j)$. By Lemma 10, we have that for $X \in \mathcal{C}(G_{\neg j};d'), Y \in \mathcal{C}(G^+(v_j);d-d',s)$, it is $\operatorname{cost}(G - X - Y - Z_j) = \operatorname{cost}(G_{\neg j} \cup G^+(v_j) - X - Y) = \operatorname{cost}(G_{\neg j} - X) + \operatorname{cost}(G^+(v_j) - Y)$. Therefore,

$$\begin{aligned} &\min_{\substack{X \in \mathcal{C}(G_{\neg j};d') \\ Y \in \mathcal{C}(G^+(v_j);d-d',s)}} \operatorname{cost}(G - X - Y - Z_j) \\ &= \min_{X \in \mathcal{C}(G_{\neg j};d')} \operatorname{cost}(G_{\neg j} - X) + \min_{Y \in \mathcal{C}(G^+(v_j);d-d',s)} \operatorname{cost}(G^+(v_j) - Y). \end{aligned} \tag{5.4}$$

By definition $\min_{X \in \mathcal{C}(G_{\neg j};d')} \operatorname{cost}(G_{\neg j} - X) = T_{G_{\neg j}}[d']$. Furthermore, $G^+(v_j)$ is a tree RTG whose root v has the single child v_j. Hence, by Lemma 13 (ii) and Lemma 9 (ii), we find

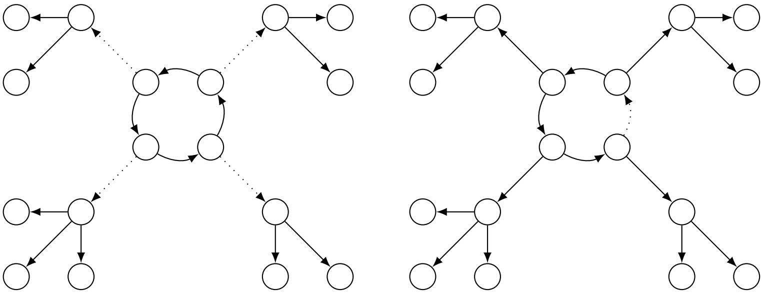
$$\begin{aligned} &\min_{Y \in \mathcal{C}(G^+(v_j);d-d',s)} \operatorname{cost}(G^+(v_j) - Y) \\ &= \min_{Y \in \mathcal{C}(G(v_j);d-d'-\Delta_{\text{diff}}(s),s-1)} \operatorname{cost}(G(v_j) - Y) + \Delta_{\text{cost}}(s) \\ &= \tilde{T}_{v_j}[d - d' - \Delta_{\text{diff}}(s), s-1] + \Delta_{\text{cost}}(s) \end{aligned} \tag{5.5}$$

Combining Equations 5.2–5.5 yields the claim. ■

For leaves v of a tree RTG G, $\tilde{T}_v[0,1] = 0$ and all other entries are ∞. We compute $T_G[\cdot]$ by iteratively applying Lemma 15 in a bottom-up fashion, using Lemma 14 to compute $T[\cdot]$ from $\tilde{T}[\cdot,\cdot]$ in linear time when needed.

Lemma 16 *Let $G = (V,E)$ be a tree RTG with n vertices and root r. The tables $\tilde{T}_r[\cdot,\cdot]$ and $T_G[\cdot]$ can be computed in $O(n^3)$ time.* □

Proof First observe that given $\tilde{T}_v[\cdot,\cdot]$ for $v \in V$, table $T_{G(v)}[\cdot]$ can be computed in linear time according to Lemma 14. In particular, $T_G[\cdot]$ can be computed from $\tilde{T}_r[\cdot,\cdot]$ in linear time.



(a) Leaving cycle intact. **(b)** Splitting cycle.

Figure 5.30: Two ways of dealing with RTG containing a cycle. Either we put all edges leaving the cycle into the copy set and keep the cycle (left side), or we split the cycle, leaving us with a tree RTG (right side).

We now bound the computation time for $\tilde{T}_r[\cdot,\cdot]$. Let $v \in V$ with children $v_1, \ldots, v_k$. Given the tables $\tilde{T}_{v_i}[\cdot,\cdot]$, we can compute $\tilde{T}_v[\cdot,\cdot]$ by Lemma 15. More precisely, for each $j = 1, \ldots, k$, we first compute $T_{G_{\neg j}}[\cdot]$ in quadratic time by Lemma 12 followed by $O(n)$ table lookups, one for each value of d'. Hence, processing v takes time $O(\deg^+(v) \cdot n^2)$. Since $\sum_{v \in V} \deg^+(v) = n-1$, the total processing time to compute $\tilde{T}_r[\cdot,\cdot]$ in a bottom-up fashion is $O(n^3)$. ■

5.3.5.3. Connected RTGs Containing a Cycle

We now look at connected RTGs that contain a cycle. Such an RTG contains a single directed cycle. Figure 5.30 shows our idea: every copy set contains either an edge of that cycle or it contains all edges that have their source on the cycle but do not belong to the cycle. This leads to a linear number of tree instances, which we solve using Lemma 16.

We first introduce an additional decomposition for copy sets to simplify the following calculations.

Lemma 17 *Let $G = (V, E)$ be a connected RTG containing a directed cycle K and let $e_1, \ldots, e_k$ denote the edges of K whose source has out-degree at least 2. Let further $O = \{(u, v) \in E \mid u \in K, (u, v) \notin K\}$. Then*

$$C(G; d) = C(G - O; d) \otimes \{O\} \cup \bigcup_{i=1}^{k} C(G - e_i; d) \otimes \{\{e_i\}\}. \qquad \square$$

Proof Every copy set $C \in C(G; d)$ contains either some edge of K or it contains all edges in O. Note that edges of K that are not among $e_1, \ldots, e_k$ are not contained in any copy set. Thus, in the former case, $e_i \in C$ for some $i \in \{1, \ldots, k\}$ and hence $C \in C(G - e_i; d) \otimes \{\{e_i\}\}$. In the latter case $C \setminus O$ is a copy set of $G - O$, hence $C \in C(G - O; d) \otimes \{O\}$. Conversely, any copy set in $C(G - O; d) \otimes \{O\}$ forms a copy set of G and also every copy set in $C(G - e_i; d) \otimes \{\{e_i\}\}$ for any value of i forms a copy set of G. This finishes the proof. ■

As before, this decomposition can be used to efficiently compute $T_G[\cdot]$ from the tables of smaller subgraphs of a connected RTG G containing a cycle.

Lemma 18 *Let $G = (V, E)$ be a connected RTG containing a directed cycle K and let $e_1, \ldots, e_k$ denote the edges of K whose source has out-degree at least 2. Let further $O = \{(u, v) \in E \mid u \in K, (u, v) \notin K\}$. Then*

$$T_G[d] = \min \left\{ T_{G-O}[d], \min_{i=1}^{k} T_{G-e_i}[d] \right\}. \qquad \square$$

Proof Using the definition of $T_G[\cdot]$ and Lemma 17, we find that

$$T_G[d] = \min_{C \in C(G;d)} \operatorname{cost}(G - C) = \min_{C \in (C(G-O;d) \otimes \{O\}) \cup \bigcup_{i=1}^{k} (C(G-e_i;d) \otimes \{\{e_i\}\})} \operatorname{cost}(G - C).$$

As we minimize cost over a union of sets, we can minimize it over the sets individually and then take the minimum of the results. Hence, we find that

$$\min_{C \in C(G-O;d) \otimes \{O\}} \operatorname{cost}(G - C) = \min_{C \in C(G-O;d)} \operatorname{cost}(G - O - C) = T_{G-O}[d]$$

and

$$\min_{C \in \mathcal{C}(G-e_i;d)\otimes\{\{e_i\}\}} \operatorname{cost}(G-C) = \min_{C \in \mathcal{C}(G-e_i;d)} \operatorname{cost}(G-e_i-C) = T_{G-e_i}[d],$$

which together yield the claim. ■

Lemma 19 *Let $G = (V, E)$ be a connected RTG containing a directed cycle. The table $T_G[\cdot]$ can be computed in $O(n^4)$ time.* □

Proof Let $e_1, \ldots, e_k$ be the edges of the cycle K. First, observe that $G - e_i$ is a tree for $i = 1, \ldots, k$. Hence, we can compute each table $T_{G-e_i}[\cdot]$ in $O(n^3)$ time by Lemma 16. Thus, computing all these tables takes $O(n^4)$ time.

Second, let $O = \{(u,v) \in E \mid u \in K, (u,v) \notin K\}$. The graph $G - O$ is the disjoint union of the cycle K and several tree RTGs $G_1, \ldots, G_t$. The table $T_K[\cdot]$ has only one finite entry and can be computed in constant time. The tables $T_{G_i}[\cdot]$ can be computed in $O(n^3)$ time. Using Lemma 12, we then compute $T_{G-O}[\cdot]$ in quadratic time.

With these tables available, we can compute $T_G[\cdot]$ according to Lemma 18. This takes $O(n^2)$ time. The overall running time is thus $O(n^4)$. ■

5.3.5.4. Putting Things Together

To compute $T_G[\cdot]$ for an arbitrary RTG G, we first compute $T_K[\cdot]$ for each connected component K of G using Lemmas 16 and 19. Then, we compute $T_G[\cdot]$ using Lemma 12 and the length of an optimal shuffle code using Lemma 8. To actually compute the shuffle code, we augment the dynamic program computing $T_G[\cdot]$ such that an optimal copy set C can be found by backtracking in the tables. An optimal shuffle code is then found by applying Greedy to $G - C$ and adding one copy operation for each edge in C.

Theorem 3 *Given an RTG G, an optimal shuffle code can be computed in $O(n^4)$ time.* □

Proof We compute all tables $T_C[\cdot]$ where C is a connected component of G in $O(n^4)$ time using Lemmas 16 and 19. Using Lemma 12, we then compute $T_G[\cdot]$ in $O(n^2)$ time. From this, we can compute the length of an optimal shuffle code by Lemma 8.

In fact, it is not difficult to modify the dynamic program in a way that, given an entry $T_G[d]$, a corresponding copy set C of G with $\text{cost}(G - C) = T_G[d]$ can be computed by backtracking in the tables. Hence, to compute an optimal shuffle code for G, we first compute an optimal copy set C_{opt} of G in $O(n^4)$ time. Then, we compute an optimal shuffle code $\pi_1, \dots, \pi_k$ for $G - C_{\text{opt}}$ using Greedy, which takes linear time according to Theorem 2.

Let $\pi = \pi_k \circ \dots \circ \pi_1$. For each edge $(u, v) \in C_{\text{opt}}$, we define a corresponding copy operation $\pi(u) \rightarrow v$. Let $c_1, \dots, c_t$ be these copy operations in arbitrary order. Then the sequence $S = \pi_1, \dots, \pi_k, c_1, \dots, c_t$ is an optimal shuffle code.

This can be seen as follows. First, by Lemma 8, the length of S is minimal. It remains to show that S is indeed a shuffle code for G. This is clearly true, as it first shuffles the values in the registers so that a subset of the values is in the correct position and then uses copy operations to transfer the remaining values to their destinations. ■

5.3.6. Related Work

From a practical point of view, our work is related to work that studies parallel copies in the context of SSA-based register allocation. Instead of implementing the parallel copies at the place where the ϕ-function is (usually the end of the preceding basic blocks), Bouchez et al. [Bou+10] propose a technique to move the parallel copy so that its implementation involves fewer copies. Brandner et al. [BC13] further improve upon this technique using data dependence graphs. Rideau et al. [RSL08] give a formal proof for the implementation correctness of parallel copies.

From a theoretical point of view, the most closely related work studies the case where the input RTG consists of a union of disjoint directed cycles, which can be interpreted as a permutation π. Then, no copy operations are necessary for an optimal shuffle code and hence the problem of finding an optimal shuffle code using `permi23` and `permi5` is equivalent to writing π as a shortest product of permutations of maximum size 5, where a permutation of n elements has size k if it fixes $n - k$ elements.

There has been work on writing a permutation as a product of permutations that satisfy certain restrictions. The factorization problem on permutation groups from computational group theory [Ser03] is the task of writing an element g of a permutation group as a product of given generators S. Hence, an algorithm for solving the factorization problem could be applied in our context by using all possible permutations of size 5 or less as the set S. However, the algorithms do not guarantee minimality of the product. For the case that S consists of all permutations that reverse a contiguous subsequence of the elements, known as the pancake sorting problem, it has been shown that computing a factoring of minimum size is NP-complete [Cap97].

Farnoud and Milenkovic [FM12] consider a weighted version of factoring a permutation into transpositions. They present a polynomial constant-factor approximation algorithm for factoring a given permutation into transpositions where transpositions have arbitrary non-negative costs. In our problem, we cannot assign costs to an individual transposition as its cost is context-dependent, e.g., four transpositions whose product is a cycle require one operation, whereas four arbitrary transpositions may require two.

Stanley [Sta81] investigates the number of ways a permutation $\pi \in S_n$ can be expressed as a product of k n-cycles. Similarly, [Str96] presents an overview of work on the problem of determining the number of ways a given permutation can be written as the product of transpositions such that the transpositions generate the full symmetric group, and such that the number of factors is as small as possible. However, we are not interested in the number of ways a given permutation can be expressed as products of transpositions or cycles. Instead, we want to efficiently find a specific product minimizing a special cost measure.

5.4. Evaluation

Our experimental evaluation consists of four parts. First, we analyze the structure of the RTGs in our test inputs. Second, we compare the quality of our two proposed code-generation approaches relative to each other by comparing the number of instructions generated for the RTGs in our test inputs. Moreover, we investigate the running times of our code-generation approaches. Third, and most importantly, we determine the benefit of using permutation instructions. We generate code for our test inputs and then measure precise dynamic instruction counts of the produced executables. We then validate these numbers by measuring the actual running time of the same executables on our hardware prototype presented in Section 5.2. In both cases, we compare an executable using permutation instructions to an executable that uses the regular instruction set. Finally, we discuss the impact of permutation reversion and present an area and frequency overhead analysis for the hardware prototype implementation.

5.4.1. Setup

We have implemented the code generation strategies from Sections 5.3.4 and 5.3.5 in libFirm [BBZ11][44]. This compiler features a mature SPARC backend and multiple completely SSA-based register allocators and copy-coalescing schemes. As compiler input, we used the test programs contained in the integer part CINT2000 of the SPEC CPU2000 benchmark suite [Hen00]. We excluded the program `252.eon` from the measurements because the frontend[45] does not support C++ code.

We performed all compile-time measurements on an Intel Core i7-3770 workstation with 3.4 GHz and 16 GiB RAM using Linux kernel 3.5. To measure the quality of the generated code, we modified the CPU emulator

[44] For libFirm, we used Git revision `88c319e982d42c57b06ccecfee20b5286aafe3ec`. For the comparison with the optimal code-generation approach, we used Git revision `778065d2dde3b7c20d4f7a25485f2d63068f6f1b` of libFirm.

[45] We used Git revision `7c6cd91cc5a2bef0b9f4555250c1266cf07d0da5` for the C frontend cparser.

QEMU [Bel05][46] to support our ISA extension consisting of the `permi` instructions and to count the number of executed instructions. Using QEMU, we were able to obtain precise dynamic instruction counts for the generated executables. All programs were compiled in soft-float mode because our prototype did not have an FPU.

To validate the results acquired from QEMU, we conducted running-time measurements on an FPGA prototype implementation of a CPU supporting our proposed instruction set extension as described in Section 5.2, in the following called PERM. We used the same binaries that ran under QEMU.

The Gaisler LEON 3 CPU [Cob17b] served as a basis for this prototype. We synthesized a LEON 3 System-on-Chip design for the ML509 evaluation board based on Xilinx Virtex-5 FPGAs [Xil16]; Figure 5.31 shows our board setup. We configured the CPU with 32 KiB instruction cache, 32 KiB data cache, 8 register windows, no FPU, and a hardware multiplier. Our board had 256 MiB of 667 MHz DDR2 SO-DIMM DRAM (not visible on picture as slot is located on the underside of the board).

We booted a self-compiled Buildroot Linux (kernel version 2.6.36) distribution [Kor16] on the FPGA prototype. We did not compile Linux with our modified compiler, i.e., the kernel was regular SPARC code and did not use permutation instructions. We connected to the board via GRMON [Cob17a] using a USB-JTAG cable during initialization and using ssh via ethernet once Linux was running. We used a compact flash card to flash a hardware design onto the FPGA. However, the booted operating system did not have access to this flash memory. Hence, our system only had a RAM disk backed by our DRAM memory. To decrease RAM usage, for each experiment, we only copied the necessary input data and binaries via ethernet onto the RAM disk. After the experiment, we deleted the files to free up memory.

To test our architecture extension with a varying number of RTGs and RTGs of varying complexity, we used four different copy-coalescing strategies, ordered from best to worst coalescing quality.

[46] We used Git revision `09c6c738e23ea8737ea01ec5f54a84f6b83b6d75`.

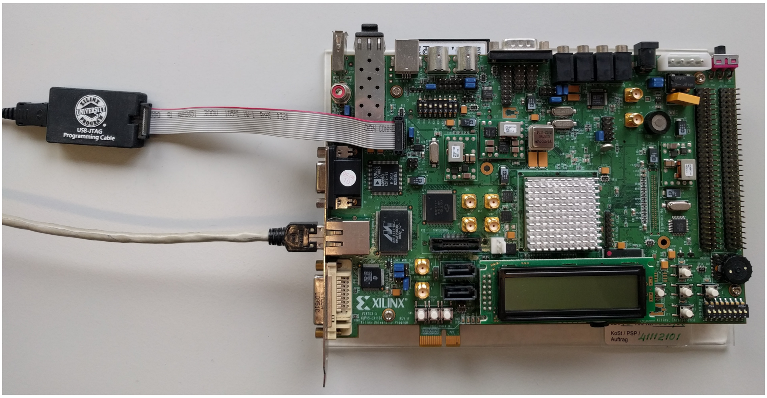


Figure 5.31: The Xilinx Virtex-5 ML509 evaluation board as used in our experiments. The board connects to a host PC via a USB-JTAG cable and to the LAN via the integrated ethernet port.

ILP An integer-linear-programming-based copy coalescer [GH07]. This produces RTGs with minimal cost according to the cost model. The cost incurred for a parallel copy is the number of unequally assigned registers multiplied by the estimated execution frequency of the parallel copy. Note that the number of unequally assigned registers is an estimate for the number of copy and swap operations that have to be generated for the parallel copy.
In our experiments, we set the ILP solver's timeout to 5 minutes per instance of the coalescing problem. If the time limit was exceeded, we used the best solution found so far. Note that exceeding the time limit does not imply the non-optimality of the found solution. In some cases, although the solution is optimal, the solver cannot prove this fact in time. The solver used in these experiments was Gurobi 5.10 [Gur12].

Recoloring A recoloring approach, which is currently one of the best conservative coalescing heuristics [HG08], resulting in RTGs with slightly higher costs.

Biased A biased coloring approach that yields good coalescing results while offering very fast allocation [BMH10]. For our benchmarks, we disabled the initial preference analysis. In this configuration, the approach is highly suitable for just-in-time compilation scenarios. The generated code contains RTGs of higher cost than with the recoloring approach.

Naive This approach does not perform any sophisticated copy coalescing at all. Except for trying to avoid copy instructions because of register constraints, no effort is made to coalesce copies. In general, this results in RTGs with high costs.

For each coalescing strategy, we inspected the properties of typical RTGs occurring in our test programs to estimate the potential benefit of using permutation instructions. Furthermore, for each of the four coalescing strategies, we tested three compiler configurations: one that generated permutation instructions using the heuristic code-generation strategy presented in Section 5.3.4, one that used the optimal strategy from Section 5.3.5, and one that emitted regular SPARC code. In the following, we focused on two configurations: the one using heuristic code generation and the one emitting SPARC code. For each of the resulting eight compiler configurations, we measured the compilation time and the quality of the generated code.

We mentioned parallel-copy-motion techniques in Section 5.3.6. In libFirm, we use a faster but less sophisticated technique, which leaves more parallel copies in the code as it only works on a single basic block. Essentially, it tries to hoist parallel copies inside a block to a location with less register pressure. However, this technique is not a contribution of this dissertation. It was enabled during all measurements.

5.4.2. Register-Transfer-Graph Properties

The number and properties of RTGs directly depend on the used coalescing strategy, which tries to minimize the cost of RTGs according to a cost model. For ease of presentation, we will use the number of RTGs and their average size as an approximation for the costs assigned to the RTGs. In general, the number and sizes of RTGs and their costs are highly correlated.

For each coalescing strategy, we analyzed the number and average size of RTGs over all programs of the CINT2000 benchmark suite. Moreover, we checked what percentage of RTGs do not duplicate any values, i.e., can

Coalescer	Number of RTGs	Average size	No value duplication
ILP (best)	77 783	2.9	74%
Recoloring	78 194	2.9	74%
Biased	178 812	4.6	54%
Naive (worst)	185 035	6.6	89%

Table 5.2: Register-transfer-graph properties. Numbers accumulated over all input programs.

be implemented only with our `permi` instructions and without additional copy instructions.

Table 5.2 shows that the number of RTGs as well as the average complexity of an RTG, represented by its number of nodes, increase with decreasing coalescing quality. Furthermore, depending on the coalescing scheme, between about half and almost 90% of the RTGs did not duplicate any values, i.e., did not need additional copy instructions. For the RTGs that did need additional copies, on average 1.26 copies per RTG were needed for the ILP, the recoloring and the naive coalescing, and 1.99 copies per RTG were needed for the biased coalescing approach. This means that the vast majority of RTGs already are in permutation form or very close to it. Thus, few additional copy instructions must be inserted during the conversion step presented in Sections 5.3.4 and 5.3.5, and most of the work can be done using only permutation instructions.

5.4.3. Heuristic and Optimal Code Generation

In the following, we compare the quality of the heuristic from Section 5.3.4 and the optimal approach from Section 5.3.5. We first analyze the heuristic and then compare it to the optimal approach.

Table 5.3 shows the total number of instructions generated by the heuristic solution for implementing the RTGs of all programs of the CINT2000 benchmark suite. The numbers confirm the expressivity of the presented

	SPARC	Per RTG	PERM	Per RTG	Change
ILP (best)	144 356	1.86	88 670	1.14	−38.6%
Recolor	159 511	2.04	89 274	1.14	−44.0%
Biased	534 378	2.99	275 079	1.54	−48.5%
Naive (worst)	947 439	5.12	343 582	1.85	−63.7%

Table 5.3: Number of instructions generated by the heuristic approach for implementing RTGs.

permutation instructions as we can implement RTGs more concisely, reducing the number of needed instructions by up to 63.7%. As every SPARC instruction, including our `permi` instructions, is encoded with 4 bytes, this also means that the code size induced by implementing RTGs is reduced by the same percentage. Additionally, regardless of the coalescing scheme, the average RTG can be implemented using fewer than two instructions when `permi` instructions are available, whereas up to 5.12 instructions are needed using the regular instruction set.

Table 5.4 compares the number of instructions generated by the heuristic with the number of instructions generated by the optimal approach. We omitted the ILP approach due to its high compilation time. As ILP solutions are at least as good as the solutions found by the recoloring approach, the change for the ILP approach is (according to the absolute amount) at most as high as for the recoloring approach and probably even lower.

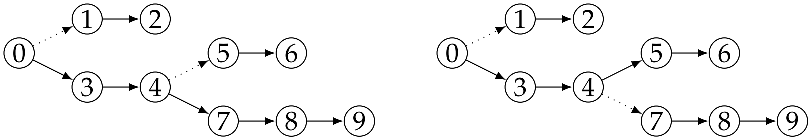
We see that the quality difference between heuristic and optimal solution is negligible in practice. The heuristic finds the optimal solution for the overwhelming majority of RTGs. Additionally, the maximum difference in RTG implementation length is 1, hence using the optimal approach saves at most one instruction for all observed RTGs.

We found that the RTGs where it does make a difference are mostly variants of the RTG shown in Figure 5.32, which we already mentioned in Section 5.3.4. In practice, it seems most important to efficiently combine small paths or cycles to exploit `permi23`. Both approaches use the same efficient and optimal greedy algorithm from Section 5.3.3 for this step.

	Heuristic	Optimal	Change
ILP (best, omitted)	—	—	—
Recolor	89 274	89 194	−0.08%
Biased	275 079	274 431	−0.24%
Naive (worst)	343 582	341 141	−0.71%

Table 5.4: Number of instructions generated for implementing RTGs of the heuristic solution compared to the optimal approach.

Finding the optimal copy set seems far less important. Hence, the additional effort for implementing the optimal approach is not worthwhile. In the following, we therefore focus exclusively on the heuristic and leave the optimal approach aside.



(a) Heuristic solution, requires 5 instructions.

(b) Optimal solution, requires 4 instructions.

Figure 5.32: Comparison of copy set chosen by heuristic with optimal copy set. Variants of this RTG show up in the set of input programs. We show copy sets with dotted edges.

5.4.4. Compilation Time

We measured the running time of our heuristic code-generation approach described in Section 5.3.4 compiling the entire CINT2000 benchmark set and compared it to the default version described in Section 5.3.1, which was already implemented in libFirm.

Table 5.5 shows the compilation times for the biased coalescing strategy. This compiler configuration has the fastest register allocation and copy

	Default	Our code gen.
RTG impl. (total)	629.1	917.3
decomposition	394.3	413.7
conversion	234.8	503.6
Backend (total)	63 598.0	63 927.0

Table 5.5: Time spent (in milliseconds) for RTG implementation during the compilation process.

coalescing while producing a high number of non-trivial RTGs. Hence, in this configuration, the relative compile-time impact of our code generation scheme is larger than in all other configurations.

We divide the total time needed for implementing RTGs into the time needed for the conversion step (from RTG to PRTG) and the time needed for the decomposition step (from PRTG to trivial RTG). Without permutation instructions, i.e., in the default implementation of libFirm, the conversion and decomposition steps correspond to the two steps in the approach presented in Section 5.3.1. With permutation instructions, the two steps correspond to the heuristic for finding copy sets explained in Section 5.3.4 and to the Greedy decomposition algorithm presented in Section 5.3.3. We repeated each experiment ten times. The standard deviation was below 1% in all cases, so we just report the minimum running time.

We found that the running time of the initial conversion into a PRTG is nearly identical for both systems. This is not surprising, considering that, as presented in Section 5.4.2, at least half of the RTGs do not require additional copies and thus can be left untouched by the conversion step. Moreover, if an RTG does require copies, on average it only requires between one and two copies, depending on the coalescing scheme. Hence, the conversion step has a low influence, both on the compile time and on the code quality.

The time needed for the decomposition step increases by a factor of 2.1. This was to be expected considering the more complex nature of our permutation instructions. To put these numbers into perspective, we included the total time spent in the backend, i.e., the total time for code selection, instruction scheduling, register allocation and emitting of assembly code. The total time spent in the backend increases by 0.5%, so the presented code generation approach does not cause significant overhead.

5.4.5. Code Quality

We evaluated the quality of the generated code using two experiments:

1. We performed a full run of the CINT2000 benchmark suite, collecting precise dynamic instruction counts using our modified QEMU version.
2. We validated these results by measuring the running times of the same executables on our FPGA prototype.

Table 5.6 shows the absolute number of executed instructions during a full CINT2000 run. We used the full input datasets provided by SPEC. The table lists the instruction count of the version using permutation instructions and the regular SPARC version, as well as the matching instruction-count change. The results are shown separately for each of the four coalescing schemes.

As expected from the numbers presented in Section 5.4.2, the benefit of using permutation instructions directly depends on the quality of coalescing: the worse the coalescing, the higher the benefit of using permutation instructions. However, regardless of the coalescing scheme used, every program profited from the use of permutation instructions. For the biased coalescing scheme, suitable for just-in-time compilation scenarios, the number of executed instructions is reduced by up to 5.1%. Even using the optimal coalescing solution, permutation instructions can reduce the instruction count by up to 1.9%.

Interestingly, the use of permutation instructions can often more than compensate for a copy coalescing of lower quality: For 8 of the 11 tested programs, the executable with permutation instructions and shuffle code produced by the worst copy-coalescing scheme (naive) executes *fewer* instructions than the regular SPARC version with shuffle code produced by the next best coalescing scheme (biased).

Benchmark	ILP			Recoloring		
	SPARC	PERM	Change	SPARC	PERM	Change
164.gzip	427.3	424.5	−0.7%	428.7	424.4	−1.0%
175.vpr	2 204.9	2 199.2	−0.3%	2 209.5	2 201.8	−0.3%
176.gcc	184.5	183.7	−0.4%	184.8	183.8	−0.5%
181.mcf	64.6	63.4	−1.9%	64.7	63.4	−1.9%
186.crafty	251.4	248.9	−1.0%	251.0	249.0	−0.8%
197.parser	515.0	510.5	−0.9%	515.7	510.4	−1.0%
253.perlbmk	558.3	555.2	−0.6%	531.8	531.0	−0.1%
254.gap	243.7	243.1	−0.3%	243.6	241.3	−0.9%
255.vortex	358.9	357.0	−0.5%	361.0	358.1	−0.8%
256.bzip2	331.0	330.0	−0.3%	333.1	331.1	−0.6%
300.twolf	1 261.2	1 256.9	−0.3%	1 261.5	1 257.1	−0.3%
Geom. mean			−0.5%			−0.5%

Benchmark	Biased			Naive		
	SPARC	PERM	Change	SPARC	PERM	Change
164.gzip	450.5	441.8	−1.9%	542.9	454.1	−16.4%
175.vpr	2 252.9	2 229.8	−1.0%	2 309.3	2 230.7	−3.4%
176.gcc	197.1	191.8	−2.7%	215.9	191.2	−11.4%
181.mcf	68.0	66.0	−2.8%	71.5	65.9	−7.8%
186.crafty	276.1	265.3	−3.9%	315.2	267.4	−15.2%
197.parser	539.1	524.4	−2.7%	617.4	539.7	−12.6%
253.perlbmk	550.9	541.0	−1.8%	611.6	551.1	−9.9%
254.gap	257.6	252.4	−2.0%	275.4	255.9	−7.1%
255.vortex	402.1	381.6	−5.1%	467.3	396.9	−15.1%
256.bzip2	360.2	349.2	−3.1%	393.1	348.5	−11.3%
300.twolf	1 275.0	1 264.7	−0.8%	1 288.9	1 264.7	−1.9%
Geom. mean			−2.2%			−8.7%

Table 5.6: Number of executed instructions (in billions) during a full run of the CINT2000 benchmark suite. Results are shown separately for each of the four coalescing schemes (ILP, Recoloring, Biased, and Naive). The third column for each scheme shows the relative change of the number of executed instructions when using permutation instructions.

In some cases, the solution found by the ILP coalescing approach executes more instructions than the executable produced by the recoloring scheme, which can happen due to two reasons. First, if the ILP solver exceeds its timeout, the best solution found up to this point might be worse than the solution found by the recoloring scheme. Second, the cost model, which is based on statically-computed execution frequencies, might not reflect the actual running time profile of the program. Hence, the optimal solution according to the cost model can be worse in practice.

To validate the results presented in Table 5.6, we measured the running times of the same executables on our FPGA prototype. As our test platform ran at a clock speed of only 80 MHz, we used the reduced input dataset distribution provided by SPEC [KL02]. The reduced inputs try to preserve the profile of the original programs while significantly reducing the running time compared to using the full input datasets.

The next issue we have to address are interferences with other system activities. The benchmark programs we used require certain system features, such as a file system for basic file I/O. Hence, it is not directly possible to execute them on the bare hardware. Instead, we used a Buildroot Linux (kernel version 2.6.36) distribution [Kor16] and run our executables in a Linux environment. As a side effect, this enables us to use the exact same binaries that we used with QEMU.

However, running under a multi-tasking OS means that inevitably some other background processes run on the system. Periodically, the Linux scheduler may perform context switches, which disturb our measurements. To alleviate this effect, we reduced background activity to a minimum by disabling all unnecessary services. Additionally, we ran our executables with the highest scheduling priority. We then ran each executable ten times, found that the standard deviation is all cases was below 5% and thus report the lowest running time.

Table 5.7 shows the running times of the executables. In general, the measurements on our FPGA prototype support our observations from the QEMU runs: the worse the coalescing, the higher the speedup gained using permutation instructions. Also, the magnitude of the speedups matches the magnitude of the instruction count reductions for each of the four coalescing configurations. Again, every program ran faster with permutation instructions.

Benchmark	ILP			Recoloring		
	SPARC	PERM	Change	SPARC	PERM	Change
164.gzip	256.8	255.6	−0.5%	257.4	255.8	−0.6%
175.vpr	446.6	443.6	−0.7%	448.3	445.7	−0.6%
176.gcc	175.9	175.0	−0.5%	175.8	175.4	−0.2%
181.mcf	45.5	45.5	−0.2%	45.6	45.5	−0.2%
186.crafty	59.3	59.3	−0.1%	59.7	58.4	−2.2%
197.parser	123.7	123.6	−0.1%	126.2	123.2	−2.4%
253.perlbmk	127.9	125.2	−2.1%	124.8	123.7	−0.9%
254.gap	31.1	30.9	−0.7%	31.2	31.0	−0.4%
255.vortex	51.4	51.0	−0.7%	51.5	51.1	−0.8%
256.bzip2	171.4	170.8	−0.3%	172.2	171.2	−0.6%
300.twolf	90.9	88.7	−2.4%	91.5	89.5	−2.2%
Geom. mean			−0.5%			−0.7%

Benchmark	Biased			Naive		
	SPARC	PERM	Change	SPARC	PERM	Change
164.gzip	263.1	258.3	−1.8%	278.2	261.3	−6.1%
175.vpr	456.4	452.6	−0.8%	466.9	464.5	−0.5%
176.gcc	190.9	185.2	−3.0%	210.8	186.5	−11.5%
181.mcf	45.7	45.2	−1.0%	45.9	45.5	−0.7%
186.crafty	64.3	62.6	−2.8%	71.7	63.2	−11.8%
197.parser	128.5	124.7	−3.0%	139.3	125.9	−9.7%
253.perlbmk	131.6	125.0	−5.0%	141.2	126.7	−10.2%
254.gap	33.3	32.1	−3.6%	34.8	32.6	−6.3%
255.vortex	56.8	52.9	−7.0%	65.2	56.1	−14.1%
256.bzip2	177.9	175.4	−1.4%	187.3	176.5	−5.7%
300.twolf	92.6	90.6	−2.2%	95.8	91.7	−4.3%
Geom. mean			−2.4%			−5.1%

Table 5.7: Running times (in seconds) of the executables on the FPGA prototype with enabled caches. We used reduced input datasets. Results are shown separately for each of the four coalescing schemes (ILP, Recoloring, Biased, and Naive).

Benchmark	ILP			Recoloring		
	SPARC	PERM	Change	SPARC	PERM	Change
164.gzip	1118.51	1096.06	−2.0%	1116.67	1102.69	−1.3%
175.vpr	2407.49	2406.62	−0.0%	2412.93	2406.44	−0.3%
176.gcc	726.55	721.08	−0.8%	728.25	717.38	−1.5%
181.mcf	131.57	131.30	−0.2%	131.02	129.16	−1.4%
186.crafty	201.52	200.58	−0.5%	201.84	200.48	−0.7%
197.parser	714.81	706.20	−1.2%	717.13	703.99	−1.8%
253.perlbmk	630.58	598.83	−5.0%	612.13	610.01	−0.3%
254.gap	140.28	138.63	−1.2%	139.83	139.50	−0.2%
255.vortex	225.10	221.69	−1.5%	222.54	222.41	−0.1%
256.bzip2	1054.36	1054.36	−0.0%	1068.79	1063.88	−0.5%
300.twolf	372.65	372.18	−0.1%	372.47	372.12	−0.1%
Geom. mean			−0.8%			−0.5%

Benchmark	Biased			Naive		
	SPARC	PERM	Change	SPARC	PERM	Change
164.gzip	1129.68	1111.77	−1.6%	1169.46	1142.56	−2.3%
175.vpr	2430.95	2421.36	−0.4%	2431.55	2415.02	−0.7%
176.gcc	738.68	738.12	−0.1%	762.74	723.85	−5.1%
181.mcf	135.17	131.19	−2.9%	131.99	131.23	−0.6%
186.crafty	207.12	205.10	−0.1%	214.84	203.75	−5.2%
197.parser	711.73	709.41	−0.3%	758.16	717.07	−5.4%
253.perlbmk	632.39	615.77	−2.6%	644.34	639.22	−0.8%
254.gap	144.30	141.39	−2.0%	148.47	143.04	−3.7%
255.vortex	228.89	225.26	−1.6%	238.20	232.58	−2.4%
256.bzip2	1082.61	1072.50	−0.9%	1131.31	1094.96	−3.2%
300.twolf	373.30	366.47	−1.8%	375.85	372.58	−0.9%
Geom. mean			−0.8%			−2.1%

Table 5.8: Running times (in seconds) of the executables on the FPGA prototype with disabled caches. We used reduced input datasets. Results are shown separately for each of the four coalescing schemes (ILP, Recoloring, Biased, and Naive).

To further improve the reliability of our results, we repeated the experiments on the FGPA prototype with disabled caches. In the following we will give details on our test setup and explain our rationale for choosing this configuration.

To avoid the influence of caches on our measurements, we disabled both the L1 code cache and the L1 data cache of our system. It is well documented [CB13; Myt+09] that the memory layout of code and data can have a significant impact on the running time. The root cause of this are architectural features of the underlying hardware whose behavior depends on memory addresses, mainly caches and branch predictors.

For example, multiple load instructions in a program may compete for the same cache line, resulting in frequent cache misses. Whether this actually happens depends on the addresses that are accessed by the load instructions as well as on the cache structure. Likewise, some branch predictors take into account (parts of) the addresses of conditional branch instructions. Hence, the predictor's behavior depends on the memory location where the branch instruction resides.

Obviously, different executable versions, e.g., with or without permutation instructions, may result in a different code layout. Unfortunately, many other factors influence code and data layout as well. Examples include the ordering of the object files during linking [CB13], the size of the environment variables [Myt+09], which influences the stack's starting address, and security features such as address space layout randomization [FSA97], which may randomize, amongst other parameters, the starting addresses of stack and heap.

By disabling the caches, we eliminate the biggest cause of address sensitivity from our setup. Moreover, our LEON 3 processor uses the simple "always taken" strategy for its branch prediction [Cob15b, §2.3.1]. The idea behind this strategy is that for loops, the conditional branch usually jumps back to the loop header as most loops run many times and therefore it makes sense to predict such conditional branches as "always taken". Hence, the LEON's branch prediction is independent of the addresses of branch instructions. Combined with the deactivation of the caches, we have thus reduced the influence of memory addresses on the behavior of our hardware to a minimum.

Disabling the caches naturally increases the running times of our executables, making them possibly less meaningful. After all, are we not actually interested in how permutation instructions behave under realistic conditions? However, we will argue in the following that, for our particular architecture, measuring speedups with deactivated caches and combining the results with the speedups from Table 5.6 should allow deriving lower and upper speedup bounds.

First, consider the dynamic instruction count obtained using QEMU. Here, we treat all instructions equally; in particular, every load and store instruction has a cost of 1. In other words, we treat each load and store instruction as if it triggered a cache hit. Furthermore, our LEON 3 processor is a non-superscalar in-order processor, i.e., it executes at most one instruction per clock cycle. Viewed this way, we can interpret the dynamic instruction-count reduction as the speedup in a best-case scenario in which the execution of every instruction takes exactly one clock cycle and every memory access is a cache hit. Hence, the dynamic instruction-count reduction should serve as an upper bound for the speedup that is possible due to permutation instructions.

Now consider running time measurements on the FPGA prototype with disabled caches. Here, every memory access is a cache miss and will therefore take multiple clock cycles. Hence, viewed this way, this experiment is a worst-case scenario and the measured speedup should form a lower bound for the possible speedup on this platform.

Therefore, when running the benchmarks with enabled caches and eliminating the influence of caches on the measurements, e.g., using the randomization techniques from [CB13], the observed speedup should fall into the range established by our lower and upper bounds. More precisely, $S_{disabled} \leq S_{enabled} \leq S_{qemu}$ should hold, where $S_{disabled}$ and $S_{enabled}$ are the speedups with disabled and enabled caches, and S_{qemu} is the speedup computed via dynamic instruction counts gathered by QEMU.

Table 5.8 shows the running times of the executables on the system with disabled caches. Again, we ran each experiment ten times and checked that the standard deviation was below 5%. To indicate geometric means despite some (rounded) speedups of 0%, we only consider non-zero speedups when computing the geometric means.

Surprisingly, the inequalities do not hold. In fact, it is hard to find a benchmark where they do hold, as most numbers violate our considerations. We cannot fully explain these effects, but discuss possible ideas in the following.

First, there is one exception, which violates our inequality also in theory: as a side effect of the shorter encoding of RTGs with permutation instructions, it could happen that an important part of the code, say a loop body, now fits into the code cache independently of other factors, such as link order, mentioned above. Then it would, in theory, be possible to observe a speedup that is higher than our established upper bound, i.e., $S_{enabled} > S_{qemu}$. Of course, this is highly specific to the platform parameters, such as cache size and structure. While we think it quite unlikely to happen, it could contribute to the observed effect.

To our knowledge, there are three possible reasons left that could further distort our results. First, there could be random effects, such as small fluctuations in DRAM latency and hardware interrupts due to I/O or network devices. We account for these by repeating the measurements ten times and checking the standard deviation, so that it is unlikely that we only measured outliers.

Second, we used different input datasets for the two runs. While the small input datasets are specifically engineered to preserve the program profiles, they may not do so perfectly. Thus, program sections that contain a lot of permutation instructions might be underrepresented or overrepresented in the profile of the run with the reduced input, leading to a lower or to a higher speedup, respectively.

And third, permutation reverts can reduce the speedup. Due to the early committing nature of the permutation instructions, up to four permutation instructions must be reverted in case of traps. Hence, permutation reversion is potentially a multi-cycle operation. Reversion is only performed when using permutation instructions, so this could penalize the executables using permutation instructions. However, we study the performance impact of reversion in Section 5.4.6 in more detail and find that its effect is negligible.

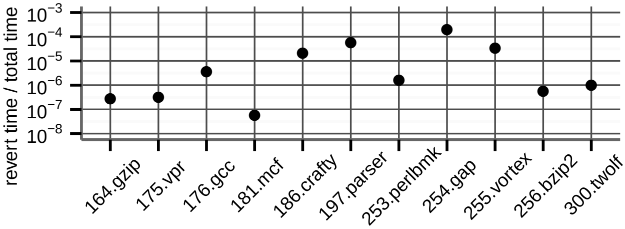


Figure 5.33: Ratio of time spent for permutation reversion to total running time of each SPEC benchmark. Data gathered from FPGA prototype using the reduced input dataset.

5.4.6. Hardware Overhead

As the hardware implementation was done by Bauer et al. (see Section 5.2), they also performed the following evaluation. Hence, this section is not a contribution of this dissertation, but the work of Bauer et al. and based on [Moh+13]. We still include this section for the sake of completeness.

Performance impact of permutation reverts. Bauer et al. measured the impact of permutation reversion, which is required to handle traps (see Section 5.2.2). They performed the measurements using a performance counter in the FPGA implementation, which counts the cycles spent for reversion. Figure 5.33 shows the ratio of time spent for reversion compared to total application running time. If traps were occurring with the same frequency for all applications, the ratio would be the same. However, the large spread of nearly 10^4 shows that for some applications window overflow/underflow traps (e.g., due to recursion) or traps due to I/O or syscalls occur more frequently. Still, the performance loss due to permutation reversion is always below 0.1% (i.e., 10^{-3}).

Area overhead of FPGA implementation. Table 5.9 shows the resource usage for the base system compared to the PERM. The PERM implementation uses multiple large multiplexers for extracting the current window and applying the new permutation to the existing one. When

	base system	PERM	Overhead
LUTs	15 024 (21%)	21 630 (31%)	44%
Slices	7 249 (41%)	9 507 (55%)	31%
Flip-flops	7 607 (11%)	8 851 (12%)	16%
BlockRAMs	28 (19%)	28 (19%)	0%
Frequency	80 MHz	80 MHz	0%

Table 5.9: Hardware implementation comparison between base system and PERM with 8 register windows. FPGA resource utilization percentage in parentheses.

using an FPGA as target technology, multiplexers are realized by look-up tables (LUTs), which explains the increased number of required LUTs. As, to the best of Bauer et al.'s knowledge, there are no publicly available memory-compilers for multi-port memories targeting ASICs, Bauer et al. focused their evaluation on FPGAs. However, multiplexer synthesis is discussed in [EL09], stating that *"Multiplexers are expensive in FPGAs and cheap in ASICs"*. Therefore, it can be assumed that the area overhead of an ASIC implementation would be considerably smaller.

Additional flip-flops are required for storing the logical-physical register-address mapping (highlighted table component in Figures 5.9 and 5.10). Compared to the base system, there is no frequency loss, as the Decode (where the extensions for register-file permutation are added) and Exception (where permutation reverts are performed if necessary) stages are not the critical path in the system. The implementation does not need additional on-chip block memory (BlockRAM).

Figure 5.34 shows the floorplan of the placed and routed PERM design on the Virtex-5 LX110T FPGA. The main logic of the permutator (multiplexers and permutation table) is in the purple area Ⓟ. The LEON 3 CPU is located in the yellow area Ⓛ, while the remaining components in the system (e.g., DDR controller, debug unit, bus arbiter, etc.) are in the green area Ⓢ.

Bauer et al. synthesized the design with different numbers of register windows to analyze the impact on area. Figure 5.35 shows the number of LUTs and Flip-flops. Decreasing the number of register windows from 8

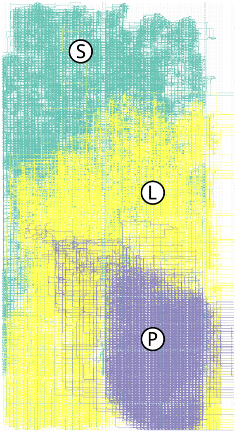


Figure 5.34: Floorplan of our FPGA implementation.

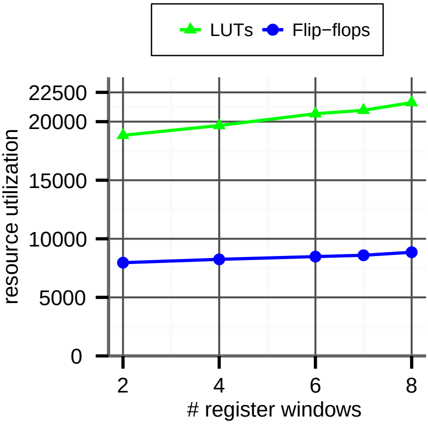


Figure 5.35: Design space exploration for different number of register windows.

to 2 significantly reduces the number of required LUTs (approximately by half)—which contribute the largest part to the area overhead. The reason is the reduction of the size of the multiplexers used for extracting the current window from the permutation table. However, programs that make use of nested function calls generally profit from a large number of register windows, thus the number of register windows is a performance-area trade-off determined at design time.

5.4.7. Threats to Validity

In this section, we try to list all limitations of our experiments as well as decisions that may have influenced our results.

The weakest point in our evaluation is that we extended an in-order architecture with a permutation table instead of reusing existing components of an out-of-order architecture (we share our thoughts on this topic in Section 5.5). We made this decision for practical reasons. At that time, no open-source out-of-order processor was available to us. Hence, Bauer et al. chose the LEON3, as they already had prior experience with that platform.

However, choosing an in-order architecture means that our overhead numbers are significantly higher than necessary and not representative for our originally targeted architectures.

Additionally, we performed our running-time measurements on a scalar in-order architecture. A real out-of-order superscalar architecture can process multiple instructions in one clock cycle. Hence, we assume that running times on such an architecture are distinctly different than on our test platform.

Our architecture is compute-bound. As our CPU runs at 80 MHz but we use regular DDR2 SO-DIMM clocked at 667 MHz, our memory is disproportionally faster compared to a real chip. Hence, loads and stores to main memory are significantly cheaper, which means the total program running time is less than what it would be if the memory speed matched the processor speed. This, in turn, means that shuffle code, which does not contain any memory accesses, takes up a higher relative portion of the total program running time. Therefore, our speedup numbers may be too high.

On the other hand, our system only has an L1 cache (no L2 cache) and the cache is quite small (only 32 KiB). This is significantly less than real machines. Therefore, the working sets of our test programs may not fit into the data cache, increasing the total program running time and reducing the relative portion spent on shuffle code. Hence, our speedup numbers may be too low.

If we compare the running-time numbers with enabled caches to those gathered with disabled caches, we find that the running time only increases by roughly a factor in the order of 5. On a real chip, where memory is much slower than the CPU, the slowdown would be multiple orders of magnitude higher. This could support our argument that our memory is unrealistically fast. On the other hand, it could also mean that the working sets of our benchmark programs are too large and we often hit main memory even with enabled caches. While our platform even had performance counters for querying the number of cache misses, unfortunately, we did not gather this data during our benchmark runs. We did not repeat the benchmark runs due to time constraints.

Additionally, multiple factors increase the cost of shuffle code on our platform. First, the SPARC instruction set has no swap instruction on registers. Hence, if there is no free register at the program point of the RTG, its implementation must use three `xor` instructions to implement each transposition. Here, it would have been interesting to compare code quality of full permutation support to a baseline where we restricted the use of `permi` to swaps, i.e., use `permi` to emulate the missing swap instruction. However, we did not perform this experiment due to time constraints.

Second, our prototype did not have an FPU. Hence, we compiled all programs in soft-float mode. In this mode, all floating-point computations are performed on integer registers. Hence, for programs that used float-point computations this overstates the amount of shuffle code for integer registers. However, only two programs of our integer benchmark set make noteworthy use of floating-point arithmetics: 175.vpr and 300.twolf[47]. Hence, for all other programs, the amount of shuffle code is representative.

Third, the SPARC calling convention passes the first six function arguments in registers (the rest via the stack). Hence, in general, there is an RTG before every call, except if coalescing can make the RTG trivial. This increases the amount of shuffle code in comparison to, e.g., 32-bit x86 code, where all arguments are passed via the stack. However, passing registers is more common, and, e.g., the 64-bit x86 ABI and ARM ABI also use registers to pass arguments. Hence, we do not consider this an unfair advantage for our technique.

Lastly, we suspect register permutations to be interesting in just-in-time compilation scenarios. However, we did not test them in that context. There could be other factors, e.g., more inefficient code in general due to compilation-time constraints, that could reduce the relative amount of running time spent on shuffle code. Hence, the benefit of using permutation instructions in that context could be lower than one would expect from our results.

[47] This also explains the high number of executed instructions of these two benchmarks in Table 5.6.

5.5. Generalization

We saw in Section 5.4.6 that implementing permutation instructions in an existing in-order architecture has a high area overhead. While this implementation allowed us to evaluate our concept more thoroughly, the real aim of permutation instructions are out-of-order architectures as mentioned in the initial motivation of this chapter. In this section, we first give a more detailed overview of an implementation technique for out-of-order execution. Based on this presentation, we then argue that permutation instructions could be added cheaply in this context, and discuss their advantages.

5.5.1. Out-of-Order Execution

A simple pipelined processor (e.g., see Section 5.2.1) executing instructions in-order can experience *pipeline stalls* due to data dependencies. For example, suppose that we have a processor where division takes more than one cycle. Furthermore, suppose that this processor has two functional units that can both handle division and multiplication. Now, consider the following sequence of instructions:

i_1: $r_1 \leftarrow$ **div** r_2, r_3
i_2: $r_4 \leftarrow$ **mul** r_1, r_1
i_3: $r_4 \leftarrow$ **mul** r_6, r_7

Here, the multiplication i_2 is data-dependent on the division i_1 as it requires the computed quotient in register r_1. However, division is a multi-cycle operation. Thus, i_2 has to wait for the result of i_1. In fact, as instruction i_2 cannot progress through the pipeline stages, no other instruction can progress through earlier stages, such as instruction fetch or decode. Hence, the pipeline is halted or *stalled* until the long-running division has finished.

In this example, i_3 is not data-dependent on the previous two instructions and could be executed on the second functional unit, which is idle during execution of the division. However, this requires executing the instructions of the program in an order that is different from the program order.

Out-of-order execution allows to dynamically rearrange instruction streams in hardware to the extent permitted by the data dependencies between the instructions. The main idea behind this approach is to track data dependencies between instructions and begin their execution as soon as all data operands are available. The technique was first proposed and implemented in the context of the IBM 360 architecture by Tomasulo [Tom67].

However, such architectures also need to take care of false data dependencies, i.e., anti-dependencies or output-dependencies, between instructions. In our example, both i_3 and i_2 write their respective results to the same register r_4. This output dependence causes problems if i_3 finishes first and writes its result to r_4 before i_2 does. Subsequent instructions would then read the wrong value from r_4.

To prevent this problem, out-of-order architectures employ *register renaming*. Register renaming removes false dependencies by using a different register in one of the conflicting instructions. In our example, we could change the destination register of i_3 from r_4 to r_t, assuming that r_t is a temporary register. We must also rename possible subsequent uses of r_4 to r_t. Then, we can execute i_2 and i_3 in any order, as renaming removed the output dependency.

Modern processors implement register renaming by providing more *physical* registers than *logical* registers, i.e., the registers visible to compiler and programmer through the instruction set architecture. Figure 5.36 shows a possible structure of such a renaming unit, see [Sim00] for alternative implementations. Here, the processor contains a *register alias table* (*RAT*) that maps logical register indices to physical register indices. In addition to a RAT, such a processor also contains a *free list* (FL) of currently unused physical registers and an *active list* (AL) with information about physical registers that are currently in use.

With this setup, register renaming usually proceeds as follows [Jou+98]. A group of instructions enter the register-renaming unit. For each instruction, the renamer[48] removes a physical register from the FL, which will be used as the new (physical) destination register for the instruction. Then, the

[48] Also called "register allocator" in the literature, not to be confused with the compiler task during code generation.

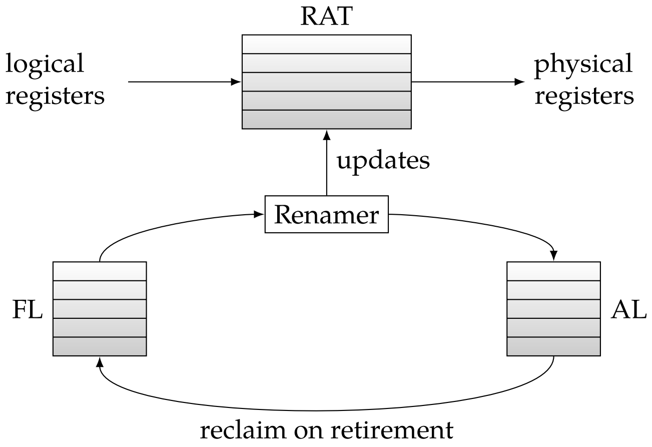


Figure 5.36: Register-renaming unit using a register alias table (RAT). The RAT maps logical to physical register indices. The free list FL contains currently unused physical registers. The active list AL contains information about physical registers that are in use.

RAT and the AL are updated according to the new mapping from logical to physical registers.

Accesses to the destination register by subsequent instructions will use the register's logical register name and will thus be rerouted to the correct physical register by the RAT. Hence, the destination register has effectively been renamed. Once the instruction has finished executing and its results are visible in the architectural state (when the instruction has *retired*), the physical register is reclaimed and transferred from the AL to the FL.

5.5.2. Implementing Permutation Instructions

In this section, we argue that permutation instructions should be cheap to implement on modern high-performance processors that already support out-of-order execution and register renaming to exploit instruction level parallelism.

Once a RAT as presented in the previous section is available, we can implement certain operations by just modifying this table. Hence, we handle these instructions during the renaming phase and do not need to actually execute them on a functional unit of the processor. For example, current Intel microprocessors implement [Int16, §2.2.2] the *move elimination* technique [Tom67; Jou+98]. These processors implement copy (or move) instructions such as `copy r1, r2` by changing the RAT mapping of `r2` to point to the physical register that `r1` currently points to. Registers `r1` and `r2` now effectively share a value saved in a single physical register.

To ensure correctness, these processors save a reference counter per physical register [Jou+98, section 2.2]. Registers in the free list FL have a reference count of 0. Whenever a physical register is allocated, its reference count is incremented. When a physical register is reclaimed, its counter value is decremented. During retirement, we only transfer a register to the free list if its reference count is 0. Hence, compared to our permutation table, a reference counter enables mapping multiple logical registers to the same physical register. Such a RAT is thus strictly more powerful than our permutation table.

Other applications of the RAT are exploiting so-called zero idioms and implementing certain exchange instructions. Zero idioms [Int16, §3.5.1.8] denote instructions that set a register to zero, e.g., `xor reg, reg`. The renamer recognizes these idioms and modifies the RAT so that `reg` points to a special physical register that is permanently set to zero. Hence, these instructions are also executed without being passed to a functional unit. The renamer also handles the floating-point instruction `fxch`, which exchanges two floating-point registers[49] [Int16, §2.3.3.1].

We argue that with a RAT already in place, adding permutation instructions should be cheap. As we see from the support for `fxch` instructions, swapping two registers is possible and already implemented in common processors. The only difference for permutation instructions is that they change more than two (in our case up to five) entries in the RAT at once. According to a detailed study of Intel's microarchitecture [Fog16, sections 8.7 and 9.8], Intel processors can rename up to four registers in one clock cycle.

Assume that we add a permutation instruction to the instruction set of such an out-of-order processor. Once a permutation instruction arrives at the renaming unit, the unit is completely occupied for one clock cycle as the instruction exhausts the available renaming capabilities. Hence, no other instructions can be subject to renaming in the same cycle. However, the permutation instructions still offer benefits: they are a more compact encoding of the wanted operation, and they may reduce latency.

First, compared to expressing the permutation as a series of exchange operations, the permutation instruction is more compact as it avoids repeating register names. For example, to do a cyclic shift of registers r_1, r_2, r_3, the encoding of `permi r1, r2, r3` is more compact than doing `swap r2, r3` followed by `swap r1, r2`.

And second, the permutation instruction may have a lower latency. In the example, the second `swap` instruction has a true data dependency on the first `swap` instruction, as it reads register r_2, which is written by the first `swap` instruction. Hence, depending on the capabilities of the renaming hardware, this dependency may prevent renaming these instructions in the same cycle. However, according to [Fog16, section 9.8], some modern microarchitectures also support eliminating "chained movs",

[49] To be precise, as x87 floating-point registers are organized as a stack, `fxch` allows exchanging an arbitrary floating-point register with the register on top of the stack.

i.e., sequences such as `copy r1, r2; copy r2, r3`. Here, the second instruction has a true data dependency on the first. If a microarchitecture is able to rename such dependent instructions in the same clock cycle, it could also rename multiple dependent `swap` instructions as those shown before. Then permutation instructions would offer no latency benefits. Only if the hardware does not support renaming dependent instructions in the same clock cycle do permutation instructions offer an advantage regarding latency.

In summary, we are confident that adding support for permutation instructions to a modern out-of-order processor is possible with low implementation overhead. However, the benefit may be limited: if the processor supports renaming dependent instructions in a single clock cycle, permutation instructions only provide a more compact encoding of RTGs. A more compact encoding is, in general, beneficial; for example, it helps for keeping tight loops in the code cache. However, the performance impact of a more compact encoding is difficult to quantify due to sensitivity concerning addresses or cache structure.

Regarding code generation, a RAT with a reference counter per physical register is strictly more powerful than our permutation table from Section 5.2 as it also allows mapping multiple logical registers to the same physical register. This would enable more powerful instructions to implement RTGs that also allow value duplication. However, it would also require different instruction formats and different code generation approaches to exploit the more powerful hardware.

Summary

- The compiler benefits from the ability to permute small sets of registers during the implementation of shuffle code.
- We can add such functionality to an existing architecture in the form of novel permutation instructions that arbitrarily permute up to five registers.
- Near-optimal implementation of shuffle code using these permutation instructions is possible in linear time.
- Generating optimal code using the permutation instructions is not worth the additional effort.
- Our permutation instructions offer a performance advantage in practice and allow interesting trade-offs. Specifically, they can sometimes compensate for a register allocation of inferior quality.
- Permutation instructions should be cheap to realize in the context of an out-of-order architecture; however, the benefits might be limited.

Dissertations are not finished; they are abandoned.

Frederick P. Brooks, Jr.

6 Conclusion and Future Work

We have made contributions regarding compilation and code generation along both dimensions of modern parallel architectures: memory and core design. In the first part of this dissertation, we presented an in-depth overview of non-cache-coherent architectures and explained the cost and trade-offs of implementing common programming models. We then took a concrete example as a case study, namely compiling the PGAS language X10 to invasive many-core architectures. Based on this platform, we identified data transfers between coherence domains as a crucial building block for efficient program execution. We exhaustively studied possible implementation techniques and trade-offs. Moreover, we developed a novel approach to avoid serialization of complex data structures through automatic compiler-directed software-managed coherence. We performed an extensive evaluation of data-transfer techniques on a prototype of an invasive many-core architecture. We could show using programs from an existing benchmark suite that our novel approach provides a speedup. Moreover, we investigated hardware acceleration for range-based cache operations and evaluated benefits and overheads.

Regarding code generation, we investigated the use of permutation instructions to allow implementing shuffle code more efficiently. Starting from a design driven by hardware constraints, we built a solid theoretical foundation for our problem setting. We developed two code generation approaches and proved multiple optimality guarantees about them. We then evaluated both approaches on an actual hardware prototype and could show that our extension provides a speedup and enables interesting trade-offs. We also discussed the implementation of such instructions on modern out-of-order architectures.

In the following, we take a step back and share our ideas for possible research directions in the future.

Compilation to invasive architectures. In our opinion, our most important realization concerning invasive architectures is that support for fine-grained cache control is useful and enables efficient program execution. We think that right now, this topic does not get the attention it deserves. The lack of such fine-grained control is one of the most prominent points of criticism of the Intel SCC, which has a structure similar to invasive architectures. The Invasive Computing project is in the unique position that it can actually adapt and improve its hardware platform. Our clear recommendation is to add support for fine-grained cache control. We advise against putting too much logic into the hardware; functionality to write back or invalidate the relevant cache line for a given address is sufficient.

We also strongly recommend to add support for off-chip memory to the existing DMA units. Currently, copying data forth and back between off-chip memory and TLM using regular loads and stores often negates the performance advantage of using hardware-accelerated asynchronous transfers. There is no conceptual reason why the current DMA units are limited to TLMs. In conjunction with fine-grained cache control, this would enable very efficient implementation [CS16; CS17] of one-sided block-wise communication means, such as `Array.asyncCopy()` in X10. As our memory tile contains some cores close to off-chip memory, it would even be possible to realize the idea of van Tol et al. [Tol+11] and let dedicated copy cores located on the memory tile handle off-chip memory transfers asynchronously without causing any NoC traffic.

It could be interesting to implement the idea of remote invalidation (and its dual operation remote writeback) for true one-sided communication as proposed by Christgau et al. [CS16; CS17]. These operations would likely be implemented either in the network adapter of a tile, in the cache controllers, or in a combination of both. We suspect that it is possible to reuse some of our work on range operations that we presented in Section 4.5.

We give more technical details on our ideas for improvements of invasive architectures in Appendix A.1. There, we provide lists of concrete implementation steps that we suspect would improve the system as a whole and make it more efficient.

Our main focus in Chapter 4 was the acceleration of copying data structures between memory partitions on non-cache-coherent architectures. However, avoiding copies altogether would be even more worthwhile. Friedley et al. [Fri+13] propose ownership passing for MPI programs on clusters. As mentioned before, clusters usually provide shared memory inside a node and use message passing between nodes. However, many application developers use message passing also inside a node to avoid having to combine multiple programming models.

Friedley et al. propose to avoid copying buffers if shared memory is available and avoiding the copy does not change program semantics. In this case, they transfer the ownership of the buffer, i.e., hand over a single pointer, instead of copying the complete buffer. They devise a data-flow analysis [ASU86, section 9.2] to identify where ownership passing is applicable, and a matching compiler transformation to automatically apply ownership passing. They mention [Fri+13, section 3] that their approach is also usable on non-cache-coherent architectures, but did not evaluate it there.

In the context of Invasive Computing, we can envision a similar compiler-based technique for X10. Especially when distributing input data, which is only read but not modified, it could be worthwhile to avoid copies. It should be possible to apply approaches based on escape and shape analysis used for detecting read-only methods in Java [Bog00] to X10's `at` blocks. We expect the object-oriented nature of X10 to be a challenge due to frequent aliasing.

Regarding correctness, it may be interesting to look at formal verification of (partially) software-based coherence protocols. There is a large body of work on verifying hardware-based coherence protocols; e.g., recently, Li et al. [Li+16] presented an approach that generates Isabelle [NPW02] proofs. However, to the best of our knowledge, no machine-checked formalizations exist of software-based coherence protocols or hybrid protocols with software and hardware components.

As a long-term research direction, we can envision the comparison of multiple programming models on invasive architectures. So far, we have focused on the PGAS programming model using X10. As we have seen in Chapter 2, other programming models are feasible as well. Additionally, as our hardware is not fixed, we can add the required features to lower the costs of implementing, e.g., the shared-memory programming model.

Moreover, at the time of writing, there is an ongoing effort [Sri+17] to enable invasive hardware to dynamically and selectively combine multiple tiles into a single coherence domain. In our opinion, this allows to explore an interesting design space. We have multiple programming models with different requirements and can either avoid coherence-related problems (e.g., with message passing), provide it (at least partially) in software (e.g., via the compiler or operating system), or provide it in hardware (using the aforementioned extension).

To the best of our knowledge, no comprehensive study exists on which programming model is the most suitable for non-cache-coherent architectures. Many papers propose a multitude of programming models and evaluate them on, e.g., the Intel SCC. In our experience, the PGAS model is a good match. Just like non-cache-coherent shared-memory architectures are situated somewhere between shared-memory and message-passing architectures, the PGAS model positions itself between the shared-memory and message-passing programming models. However, we do not know of a solid comparison that takes into account a wider range of programming models and supports the comparison with empirical data gathered on real non-cache-coherent hardware. Future work in the scope of Invasive Computing could close this gap.

Code generation using permutation instructions. From the theoretical side, it would be interesting to generalize our greedy algorithm to larger permutations. We proved in Chapter 5 that our greedy algorithm finds optimal solutions for expressing a permutation as a product of permutations of maximum size 5 (for our definition of permutation size). What if we want to find optimal solutions using permutations of maximum size k? Rutter proved that the shuffle-code-generation problem is NP-complete if k is part of the input (see Appendix A.3). However, if k is fixed, we suspect that for every maximum size k, we can find an optimal greedy algorithm. That algorithm may increase exponentially in size due to combinatorial explosion of the necessary case distinction. Still, we suspect that such a greedy algorithm exists for every k.

As we have seen in Section 5.5, on real out-of-order microarchitectures, we could even express more powerful operations that copy values and not just permute them. It would be interesting to explore what an instruction set extension would look like if it enabled permuting as well as copying values between multiple registers. Also, this would require new code-generation algorithms. We suspect that these algorithms would differ substantially from our current ones.

From the practical side, it would be interesting to look at parallel-copy-motion techniques [Bou+10; BC13]. By default, these techniques use cost models that target traditional architectures with copy and swap instructions. However, it could be interesting to tailor a cost model to permutation instructions and see if that produces more efficient code.

Moreover, we would like to explore permutation instructions (or even more powerful ones as mentioned above) on a real out-of-order microarchitecture. One possibility is to integrate the extension into an open-source out-of-order core, such as BOOM v2 [Cel+17]. Here, our goal would be to get a better idea of the costs and benefits of additional instructions in the context of an architecture that already has some necessary components. After our discussion in Section 5.5, we suspect that the benefits are not worth the effort. However, in our opinion, this suspicion is best proved by an actual implementation. Another possibility is to use a tool for architectural simulation, such as Gem5 [Bin+11]. Here, we could gather no hardware overhead numbers, but at least perform representative running time measurements on an out-of-order architecture.

Just one more thing. . .

Columbo

A Appendix

A.1. Recommendations for Invasive Architectures

In this section, we make concrete recommendations for improvements of invasive architectures in general, and the prototype platform in specific. We describe our conclusions from Chapter 6 with more technical depth and list concrete implementation steps. We cover both hardware and software components.

We agree with Christgau et al. [CS16; CS17] that software-managed coherence for efficient one-sided communication is important on non-cache-coherent architectures. We showed in Section 4.6.3 that this also applies to invasive architectures and in Section 4.7 that it is especially important for invasive programs.

Concerning flat data structures, this requires an efficient implementation of asynchronous copy operations, e.g., `Rail.asyncCopy()` in X10. To reach this goal, the following steps are necessary (preferably to be implemented in that order):

1. Add support for fine-grained cache control to the L1 caches. We showed in Section 4.6.5 that functionality to write-back and invalidate the relevant cache line for a given address is sufficient. The functionality in the L2 cache can serve as a model. As our L1 caches are configured in write-through mode, invalidation is sufficient.
2. Add support for off-chip memory, i.e., DRAM, to the DMA units. Ideally, DMA units would not only issue loads and stores but use larger burst transfers via the NoC. Alternatively, we could offload memory copy operations to the LEON cores present on our memory tile to avoid NoC traffic completely. We suspect that the prototype platform would benefit greatly from this approach. However, we think it is of limited use regarding realistic architectures, as they are unlikely to have any cores positioned near memory.
3. Ideally, add support for remote invalidations (as proposed by Christgau et al. [CS16]) and remote write-backs. Remote invalidations enable true one-sided copying from local memory to remote memory. Remote write-backs enable true one-sided copying from remote memory to local memory. We suspect that it is possible to implement such remote cache operations on our platform by a collaboration of network adapter and cache controllers. For example, if tile *A* issues a remote invalidation to tile *B*, the network adapter of tile *B* would receive this request and broadcast it on *B*'s local bus. Subsequently, all of *B*'s cache controllers would snoop this request and then execute the invalidation on *B*'s caches. As such remote operations should proceed without software involvement, we suspect that our insights from Section 4.5 can be useful. In particular, a design where the network adapter delegates invalidations or write-backs to the cache controllers could completely reuse our work on range operations.

Concerning pointered data structures, we recommend the following steps (preferably to be implemented in that order):

1. Use a static (or even better, a hybrid) allocation approach for TLMs. Our results from Section 4.6.4.2 show that most object graphs are quite small. We recommend to extend the X10 runtime system to reserve a fixed portion of each TLM for transferring small object graphs without requiring a forth-and-back communication to allocate memory in the remote TLM. It may be worthwhile to investigate directly serializing object graphs into the TLM of the receiving tile, and then using software-

managed coherence (see above). This avoids the overhead for copying data between TLMs.

2. Use cloning only as an opt-in mechanism as it is, in general, incompatible with custom serialization formats. We think adding an annotation so that programmers can switch to cloning for performance-critical transfers would be a viable way.

A.2. The Intermediate Representation Firm

Firm is graph-based intermediate representation (*IR*) for compilers [BBZ11]. The project was started in 1996 as the IR for the Sather-K compiler Fiasco [AR96], giving Firm its name: Fiasco's Intermediate Representation Mesh. Today, libFirm [Fir17] provides an open-source implementation of Firm as well as multiple frontends and code-generation backends.

Firm represents programs based on the "sea of nodes" idea by Click [CP95; Cli95]. Each function of the program is represented by a graph, where nodes represent operations and directed edges represent both data flow and control flow. In contrast to representations relying on instruction lists, the graph representation only defines a partial order on the operations. Hence, it makes the compiler's degrees of freedom explicit concerning, e.g., evaluation order.

As an example, we look at the following function that computes the maximum of its two signed integer parameters:

```
int max(int x, int y)
{
  int res;
  if (x > y)
    res = x;
  else
    res = y;
  return res;
}
```

In the following, we will give a brief explanation of how Firm models this function. We refer to [BBZ11] for details.

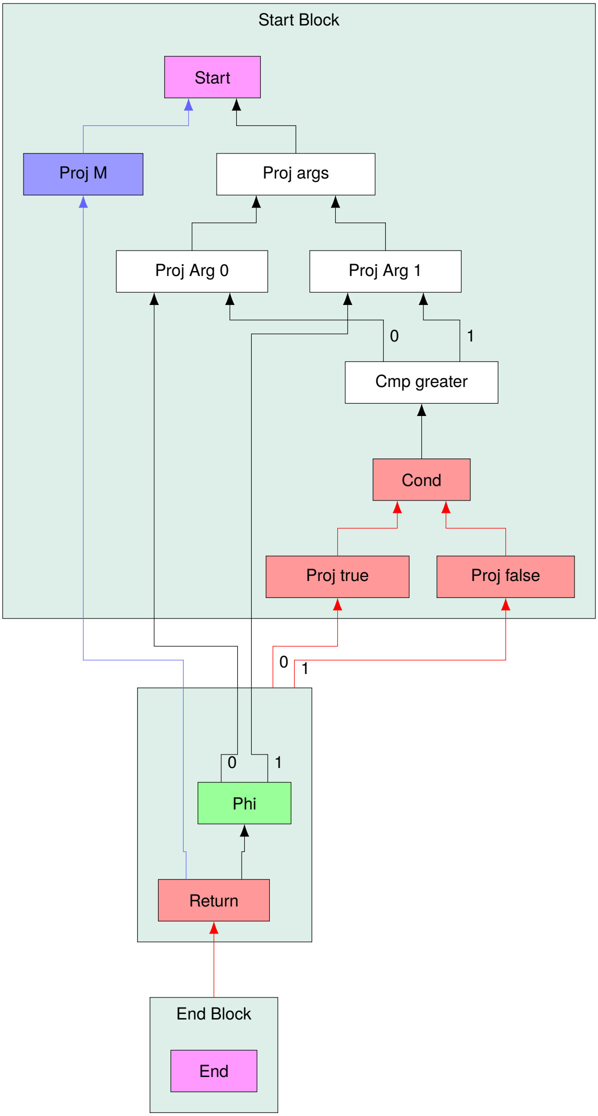


Figure A.1: The FIRM graph of a function returning the maximum of its two integer parameters. The graph has been simplified for presentation reasons.

Figure A.1 shows a Firm graph of the function max. The graph has been simplified for presentation reasons. Note that the direction of edges in Firm is backwards, i.e., edges go from the dependent operation to the operation it depends on. This applies to both reversed data-flow edges (shown in black) and reversed control-flow edges (shown in red).

We see that there are three basic blocks: the start block, a middle block, and the end block. The start block contains most of the program logic. First, it compares the two function arguments using a Cmp node. The arguments are retrieved using Proj nodes. As proposed by Click [CP95] for efficiency reasons, Firm models operations that produce multiple results as nodes that produce a result tuple. The desired component of a tuple is then extracted (or "projected out") using a Proj node. Hence, we have one Proj node for each function argument value. Following the comparison, the Cond node uses the truth value carrying the result of the comparison and produces a tuple containing control-flow information. In our example, we branch to the same basic block regardless of the comparison result.

In the second basic block, we see a Phi node. The Phi node always has exactly as many operands as its basic block has predecessors in the control-flow graph. Following the semantics of ϕ-functions, the Phi node selects operand 0 (corresponding to parameter x) if its basic block is entered via edge 0 (if x > y), or it selects operand 1 (corresponding to parameter y) if its basic block is entered via edge 1 (if x <= y). The resulting value is returned by the Return node.

Actually, Firm nodes are weakly typed. The types are called "modes"[50]. We omitted modes in our example graph for presentation reasons.

Other notable properties of Firm and libFirm are:

- libFirm directly constructs SSA form without taking a detour via a non-SSA IR [Bra+13]. Moreover, optimizations, such as constant folding, are performed during construction of the IR.
- libFirm performs SSA-based register allocation [Hac07]. This enables libFirm to retain SSA form of programs even in its backend.
- Firm models memory (and side effects in general) as a special Memory value. If Firm can prove operation independence, multiple Memory values can coexist and are only joined when needed.

[50] Which probably comes from Algol 68.

A.3. k-Shuffle Code Generation is NP-complete

Throughout this dissertation we have assumed that our permutation instructions may arbitrarily permute up to five registers. Rutter considered the case where a permutation instruction may permute up to k registers arbitrarily. We call such a shuffle code a *k-shuffle code*. The problem *k-shuffle code generation* asks for a shortest k-shuffle code that implements a given RTG. Rutter shows that k-shuffle code generation is NP-complete if k is part of the input. Moreover, Rutter presents an approximation algorithm for k-shuffle code generation. These results are not a contribution of this dissertation, but are original (and at the time of writing unpublished) work by Rutter.

A.3.1. Complexity

Here we refer to the decision version of the k-shuffle code generation problem where the task is to decide the existence of a k-shuffle code with length at most b.

Theorem 4 *k-shuffle code generation is NP-complete for PRTGs.* □

Proof The problem is clearly in NP, since we can guess a shuffle code and verify that it implements the given PRTG and has size at most b.

To show NP-hardness, Rutter gives a reduction from the strongly NP-complete problem 3-Partition [GJ90]. An instance (A, B) of 3-Partition consists of an integer bucket size B and a multiset $A = \{a_1, \ldots, a_{3m}\}$ of $3m$ integers such that $B/4 < a_i < B/2$. The task is to decide whether A can be partitioned into sets $S_1, \ldots, S_m$ such that $\sum_{a \in S_j} a = B$ for each S_j. Note that due to the restrictions on the a_i, each set of the partition necessarily contains precisely three elements.

Given an instance (A, B) of 3-Partition, we define a PRTG G that contains for each element a_i a directed cycle C_i of length a_i. Since 3-Partition is strongly NP-complete, we can assume that the a_i and B are polynomially bounded in m. The reduction can then be carried out in polynomial time. We claim that G admits a B-shuffle code of length m if and only if (A, B) is a yes-instance of 3-Partition.

Given a partition $S_1, \dots, S_m$, we create for each $S_j = \{a_x, a_y, a_z\}$ a corresponding instruction that resolves exactly the cycles C_x, C_y, and C_z, which by assumption consist of exactly B vertices.

Conversely, assume we have a shuffle code consisting of m operations. By construction, the PRTG G has mB vertices and none of them has a self-loop, i.e., each vertex has to be touched by at least one operation. However, each operation can touch at most B vertices, and hence every vertex is touched exactly once. This in turn implies that every vertex that is touched by an operation must be mapped to the target of its single outgoing edge by this operation. It follows that if an operation touches any vertex of a cycle, then it must resolve this cycle completely. It thus follows that every operation resolves exactly three cycles whose total size is B. Thus, the operations define a solution of 3-Partition. ■

A.3.2. Approximation Algorithm

Rutter presents a simple linear-time approximation algorithm for k-shuffle code generation.

Lemma 20 *A $(1 + 1/k)$-approximation of an optimal k-shuffle code can be computed in linear time.* □

Proof By Lemma 1 an optimal shuffle code exists with $N = \sum_{v \in V} \max\{\deg(v) - 1, 0\}$ copy operations, and in fact every shuffle code uses at least N copy operations. Let C_{opt} denote an optimal copy set and let $G_{\text{opt}} = G - C_{\text{opt}}$. Let K_{opt} denote the number of vertices that are incident to a non-loop edge in G_{opt}. Clearly, each of these vertices has to be touched by at least one permutation instruction, each of which can touch up to k registers. Thus, $N + K_{\text{opt}}/k$ is a lower bound on the number of instructions.

Now let C_{apx} be a copy set of G of size C that contains no loops. Let K_{alg} denote the number of vertices of $G_{\text{alg}} = G - K_{\text{alg}}$ that are incident to a non-loop. Since C_{apx} contains no loops we have $K_{\text{alg}} \leq K_{\text{opt}}$. It is not hard to see that a k-shuffle code exists for G_{alg} using at most $K_{\text{alg}}/(k-1)$ operations. Thus, our algorithm requires $N + K_{\text{alg}}/(k-1)$ operations in total. It follows that our algorithm is a $(1 + 1/k)$-approximation. ■

For the case of $k = 5$, which we considered in this dissertation, this yields a 1.2-approximation.

Weeks of coding can save you hours of planning.

Unknown author

B Software Artifacts

In printed copies of this dissertation, we enclosed all relevant software artifacts on a slightly anachronistic DVD. Additionally, we provide all software artifacts as a download at

`https://manuelmohr.de/dissertation/artifacts.tar.bz2`

In the following, we give an overview of the directory structure. We try to reflect the dissertation structure in our directory structure to ease navigation. Hence, we provide artifacts for Chapters 4 and 5 separately. Every software or hardware revision mentioned in the evaluation sections in this dissertation is relative to the projects we provide here. We focus on the software and provide necessary hardware artifacts as synthesized bitfiles and not as VHDL sources.

For convenience, we also provide a PDF version of this dissertation with clickable references at

`https://manuelmohr.de/dissertation/diss-genehmigt.pdf`

- /
  - Promotion
    - dissertation .... LaTeX sources of this dissertation
    - Formalities .... Various forms needed for this dissertation
  - Chapter4
    - Braun2012TR .... LaTeX sources of [Bra+12]
    - Braun2014X10 .... LaTeX sources of [Bra+14]
    - Mohr2015X10 .... LaTeX sources of [Moh+15]
    - Mohr2017DATE .... LaTeX sources of [MT17]
    - benchmarks .... Binaries and raw benchmark data
    - bitfiles .... Bitfiles used for CHIPit platform
    - chipit-runner .... Testrunner scripts for CHIPit platform
    - grmon_tools .... Scripts for CHIPit platform
    - imsuite .... Sources of adapted benchmark programs
    - irtss .... Sources of *i*RTSS
    - multigrid .... The multigrid application
    - x10i .... Sources of adapted X10 compiler
      - bdwgc .... Adapted garbage collector
      - cparser .... C frontend (see Section 3.5)
      - jFirm .... Java bindings for Firm
      - libfirm .... Unmodified Firm library
      - liboo .... Object-orientation support library
      - tests .... Synthetic benchmark programs
      - x10.firm .... Firm backend for X10 compiler
      - x10.firm_runtime .... Adapted X10 runtime system
        - src-c/octopos .... Mapping to OctoPOS interfaces
        - src-c/posix .... Mapping to POSIX interfaces
  - Chapter5
    - Mohr2013CASES .... LaTeX sources of [Moh+13]
    - Buchwald2015WADS .... LaTeX sources of [BMR15b; BMR15a]
      - prog/paper_impl .... C++ implementation of Section 5.3.5
    - bitfiles .... Bitfiles used for FPGA platform
    - benchmarks .... Binaries and raw benchmark data
    - cparser .... Sources of adapted libFirm C frontend
      - libfirm .... Sources of adapted libFirm supporting permi
    - linux .... Buildroot Linux image
    - qemu .... Sources of adapted QEMU version

List of Figures

List of Tables

Bibliography

[Adv+91] Sarita V. Adve, Vikram S. Adve, Mark D. Hill, and Mary K. Vernon. “Comparison of Hardware and Software Cache Coherence Schemes”. In: *Proceedings of the 18th Annual International Symposium on Computer Architecture*. ISCA ’91. Toronto, Ontario, Canada: ACM, 1991, pp. 298–308. ISBN: 0-89791-394-9. DOI: `10.1145/115952.115982`.

[Adv10] Advanced Micro Devices. *AMD64 Architecture Programmer’s Manual Volume 2: System Programming*. `http://developer.amd.com/wordpress/media/2012/10/24593_APM_v21.pdf`. 2010.

[Ald+11] Jonathan Aldrich, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, and Roger Wolff. “Permission-Based Programming Languages (NIER track)”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. New York, NY, USA: ACM, 2011, pp. 828–831. DOI: `10.1145/1985793.1985915`.

[All+05] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. *The Fortress Language Specification*. Tech. rep. 2005.

[Alm11] George Almasi. "PGAS (Partitioned Global Address Space) Languages". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Springer Publishing Company, Incorporated, 2011, pp. 1539–1545. ISBN: 9780387097657.

[AP03] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. 2nd. New York, NY, USA: Cambridge University Press, 2003. ISBN: 052182060X.

[App97] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. New York, NY, USA: Cambridge University Press, 1997. ISBN: 0-521-58775-1.

[AR96] Markus Armbruster and Christian von Roques. "Entwurf und Realisierung eines Sather-K-Übersetzers". In German. MA thesis. Dec. 1996. URL: `http://www.info.uni-karlsruhe.de/papers/ArRo_96-fiasco_diplomarbeit.ps.gz`.

[ARM09] ARM. *ARM1136J-S technical reference manual*. r1p5. ARM, 2009.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science and information processing. Addison-Wesley Publishing Company, 1986. ISBN: 9780201100884.

[BA08] Hans-J. Boehm and Sarita V. Adve. "Foundations of the C++ Concurrency Memory Model". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 68–78. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375591`.

[Bal+16] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. "OpenPiton: An Open Source Manycore Research Framework". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA:

ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5. DOI: `10.1145/2872362.2872414`.

[Bar+15] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 29:1–29:16. ISBN: 978-1-4503-3238-5. DOI: `10.1145/2741948.2741962`.

[Bar16] Blaise Barney. *Introduction to Parallel Computing*. `https://computing.llnl.gov/tutorials/parallel_comp/`. 2016.

[Bas+16] Arkaprava Basu, Sooraj Puthoor, Shuai Che, and Bradford M. Beckmann. "Software Assisted Hardware Cache Coherence for Heterogeneous Processors". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Alexandria, VA, USA: ACM, 2016, pp. 279–288. ISBN: 978-1-4503-4305-3. DOI: `10.1145/2989081.2989092`.

[Bas04] Cedric Bastoul. "Code Generation in the Polyhedral Model Is Easier Than You Think". In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 7–16. ISBN: 0-7695-2229-7. DOI: `10.1109/PACT.2004.11`.

[Bau+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: `10.1145/1629575.1629579`.

[Bau09] Lars Bauer. "RISPP: A Run-time Adaptive Reconfigurable Embedded Processor". Karlsruhe, KIT, Dissertation 2009. PhD thesis. 2009.

[BBZ11] Matthias Braun, Sebastian Buchwald, and Andreas Zwinkau. *Firm—A Graph-Based Intermediate Representation*. Tech. rep. 35. Karlsruhe Institute of Technology, 2011. URL: `http://pp.info.uni-karlsruhe.de/uploads/publikationen/braun11wir.pdf`.

[BC13] Florian Brandner and Quentin Colombet. "Elimination of Parallel Copies using Code Motion on Data Dependence Graphs". In: *Computer Languages, Systems & Structures* 39.1 (2013), pp. 25–47.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. "Improvements to Graph Coloring Register Allocation". In: *ACM Transactions on Programming Languages and Systems* 16.3 (May 1994), pp. 428–455. ISSN: 0164-0925. DOI: `10.1145/177492.177575`.

[BDR07] Florent Bouchez, Alain Darte, and Fabrice Rastello. "On the Complexity of Register Coalescing". In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 102–114. ISBN: 0-7695-2764-7. DOI: `10.1109/CGO.2007.26`.

[BDR08] Florent Bouchez, Alain Darte, and Fabrice Rastello. "Advanced Conservative and Optimistic Register Coalescing". In: *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '08. Atlanta, GA, USA: ACM, 2008, pp. 147–156. ISBN: 978-1-60558-469-0. DOI: `10.1145/1450095.1450119`.

[Bec+] Jürgen Becker, Stephanie Friederich, Jan Heißwolf, Ralf Koenig, and David May. "Hardware Prototyping of Novel Invasive Multicore Architectures". In: *Proceedings of the 17th Asia and South Pacific Design Automation Conference*. ASP-DAC '12. Sydney, Australia, pp. 201–206. DOI: `10.1109/ASPDAC.2012.6164945`.

[Bec16] Johannes Bechberger. *Besser Benchmarken*. Bachelor's thesis. In German. Apr. 2016.

[Bel05] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–41.

[Bin+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718.

[Blu+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*. Vol. 30. 8. ACM, 1995.

[BM02] L. Benini and G. De Micheli. "Networks on Chips: a New SoC Paradigm". In: *Computer* 35.1 (Jan. 2002), pp. 70–78. ISSN: 0018-9162. DOI: 10.1109/2.976921.

[BMH10] Matthias Braun, Christoph Mallon, and Sebastian Hack. "Preference-Guided Register Assignment". In: *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*. CC'10/ETAPS'10. Paphos, Cyprus: Springer-Verlag, 2010, pp. 205–223. ISBN: 978-3-642-11969-9. DOI: 10.1007/978-3-642-11970-5_12.

[BMR15a] Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter. "Optimal Shuffle Code with Permutation Instructions". In: *CoRR* abs/1504.07073 (2015). URL: http://arxiv.org/abs/1504.07073.

[BMR15b] Sebastian Buchwald, Manuel Mohr, and Ignaz Rutter. "Optimal Shuffle Code with Permutation Instructions". In: *Algorithms and Data Structures*. Ed. by Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege. Vol. 9214. WADS'15. Lecture Notes in Computer Science. Springer International Publishing, 2015, pp. 528–541. DOI: 10.1007/978-3-319-21840-3_44.

[BMZ15] Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. "Malleable Invasive Applications". In: *Proceedings of the 8th Working Conference on Programming Languages*. ATPS'15. Springer Berlin Heidelberg, 2015, pp. 123–126.

[Bog00] Jeff Bogda. "Detecting Read-Only Methods in Java". In: *Languages, Compilers, and Run-Time Systems for Scalable Computers: 5th International Workshop*. Ed. by Sandhya Dwarkadas. Berlin, Heidelberg: Springer Berlin Heidelberg, May 2000, pp. 143–154. ISBN: 978-3-540-40889-5. DOI: 10.1007/3-540-40889-4_11.

[BOS14] James Beyer, David Oehmke, and Jeff Sandoval. *Transferring user-defined types in OpenACC*. 2014.

[Bou+07] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. "Register Allocation: What Does the NP-completeness Proof of Chaitin Et Al. Really Prove? Or Revisiting Register Allocation: Why and How". In: *Proceedings of the 19th International Conference on Languages and Compilers for Parallel Computing*. LCPC'06. New Orleans, LA, USA: Springer-Verlag, 2007, pp. 283–298. ISBN: 978-3-540-72520-6.

[Bou+10] Florent Bouchez, Quentin Colombet, Alain Darte, Fabrice Rastello, and Christophe Guillon. "Parallel Copy Motion". In: *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '10. St. Goar, Germany: ACM, 2010, 1:1–1:10. ISBN: 978-1-4503-0084-1. DOI: 10.1145/1811212.1811214.

[Bra+12] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. *An X10 Compiler for Invasive Architectures*. Tech. rep. 9. Karlsruhe Institute of Technology, 2012. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028112.

[Bra+13] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. "Simple and Efficient Construction of Static Single Assignment Form". In: *Compiler Construction*. Ed. by Ranjit Jhala and Koen Bosschere. Vol. 7791. Lecture Notes in Computer Science. Springer Berlin

Heidelberg, 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9_6.

[Bra+14] Matthias Braun, Sebastian Buchwald, Manuel Mohr, and Andreas Zwinkau. *Dynamic X10: Resource-Aware Programming for Higher Efficiency*. Tech. rep. 8. X10 '14. Karlsruhe Institute of Technology, 2014. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000041061.

[Bri+06] P. Brisk, F. Dabiri, R. Jafari, and M. Sarrafzadeh. "Optimal Register Sharing for High-Level Synthesis of SSA Form Programs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25.5 (May 2006), pp. 772–779. ISSN: 0278-0070. DOI: 10.1109/TCAD.2006.870409.

[BS93] William J. Bolosky and Michael L. Scott. "False Sharing and Its Effect on Shared Memory Performance". In: *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*. Sedms'93. San Diego, California: USENIX Association, 1993, p. 3.

[Bun+13] Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber, Gregor Snelting, and Andreas Zwinkau. "Invasive Computing in HPC with X10". In: *Proceedings of the third ACM SIGPLAN X10 Workshop*. X10 '13. New York, NY, USA: ACM, 2013, pp. 12–19. DOI: 10.1145/2481268.2481274.

[BW88] Hans-Juergen Boehm and Mark Weiser. "Garbage Collection in an Uncooperative Environment". In: *Software: Practice and Experience* 18.9 (1988), pp. 807–820.

[BZB11] Sebastian Buchwald, Andreas Zwinkau, and Thomas Bersch. "SSA-Based Register Allocation with PBQP". In: *Compiler Construction*. Ed. by Jens Knoop. Vol. 6601. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 42–61. DOI: 10.1007/978-3-642-19861-8_4.

[Cap97] Alberto Caprara. "Sorting by Reversals is Difficult". In: *Proceedings of the First Annual International Conference on Computational Molecular Biology*. RECOMB '97. Santa Fe, New Mexico, USA: ACM, 1997, pp. 75–83. ISBN: 0-89791-882-7. DOI: 10.1145/267521.267531.

[Car+13] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu. "Runnemede: An Architecture for Ubiquitous High-Performance Computing". In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*. HPCA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 198–209. DOI: `10.1109/HPCA.2013.6522319`.

[Cav+11] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. "Habanero-Java: The New Adventures of Old X10". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Kongens Lyngby, Denmark: ACM, 2011, pp. 51–61. ISBN: 978-1-4503-0935-6. DOI: `10.1145/2093157.2093165`.

[CB13] Charlie Curtsinger and Emery D. Berger. "STABILIZER: Statistically Sound Performance Evaluation". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 219–228. ISBN: 978-1-4503-1870-9. DOI: `10.1145/2451116.2451141`.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". In: *Int. J. High Perform. Comput. Appl.* 21.3 (Aug. 2007), pp. 291–312. ISSN: 1094-3420. DOI: `10.1177/1094342007078442`.

[Cel+17] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolić, David A Patterson, and Krste Asanović. "BOOMv2: an Open-Source Out-Of-Order RISC-V Core". In: *First Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2017.

[Cha+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. "X10: An Object-oriented Approach to Non-uniform Cluster Computing". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Program-*

ming, Systems, Languages, and Applications. OOPSLA '05. San Diego, CA, USA: ACM, 2005, pp. 519–538. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094852.

[Cha82] G. J. Chaitin. "Register Allocation & Spilling via Graph Coloring". In: *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. SIGPLAN '82. Boston, Massachusetts, USA: ACM, 1982, pp. 98–105. ISBN: 0-89791-074-5. DOI: 10.1145/800230.806984.

[Che+07] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. "Cell Broadband Engine Architecture and Its First Implementation: A Performance View". In: *IBM J. Res. Dev.* 51.5 (Sept. 2007), pp. 559–572. ISSN: 0018-8646. DOI: 10.1147/rd.515.0559.

[CHH11] Keith Chapman, Ahmed Hussein, and Antony L. Hosking. "X10 on the Single-chip Cloud Computer: Porting and Preliminary Performance". In: *Proceedings of the 2011 ACM SIGPLAN X10 Workshop*. X10 '11. San Jose, California: ACM, 2011, 7:1–7:8. ISBN: 978-1-4503-0770-3. DOI: 10.1145/2212736.2212743.

[Cho+11] Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Smolinski, Nima Honarmand, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. "DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism". In: *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 155–166. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.21.

[Chr14] George Chrysos. "Intel® Xeon Phi™ Coprocessor — the Architecture". In: *Intel Whitepaper* (2014).

[Cla+11] Carsten Clauss, Stefan Lankes, Pablo Reble, and Thomas Bemmerl. "Evaluation and Improvements of Programming Models for the Intel SCC Many-core Processor". In: *International Conference on High Performance Computing Simulation*. July 2011, pp. 525–532. DOI: 10.1109/HPCSim.2011.5999870.

[Cli95] Cliff Click. "Combining Analyses, Combining Optimizations". PhD thesis. Rice University, Feb. 1995.

[Cob15a] Cobham Gaisler. *LEON Bare-C Cross Compilation System*. `http://www.gaisler.com/index.php/products/operating-systems/bcc`. Retrieved on 2015-11-13. 2015.

[Cob15b] Cobham Gaisler. *LEON SRMMU Behaviour*. Technical note 2015-10-27, Doc. No GRLIB-TN-0002, Issue 1.0. 2015. URL: `http://www.gaisler.com/doc/antn/GRLIB-TN-0002.pdf`.

[Cob16] Cobham Gaisler. *GRLIB IP Core User's Manual*. `http://gaisler.com/doc/grusbdc.pdf`. Version 1.5.0, retrieved on 2017-04-21. Jan. 2016.

[Cob17a] Cobham Gaisler. *GRMON*. Debug monitor for LEON processors. 2017. URL: `http://www.gaisler.com/index.php/products/debug-tools/grmon`.

[Cob17b] Cobham Gaisler. *LEON 3*. 2017. URL: `http://www.gaisler.com/leonmain.html`.

[Col+11] Quentin Colombet, Benoit Boissinot, Philip Brisk, Sebastian Hack, and Fabrice Rastello. "Graph-Coloring and Treescan Register Allocation Using Repairing". In: *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '11. Taipei, Taiwan: ACM, 2011, pp. 45–54. ISBN: 978-1-4503-0713-0. DOI: `10.1145/2038698.2038708`.

[CP95] Cliff Click and Michael Paleczny. "A Simple Graph-Based Intermediate Representation". In: *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*. IR '95. San Francisco, California, USA: ACM, 1995, pp. 35–49. ISBN: 0-89791-754-5. DOI: `10.1145/202529.202534`.

[CS16] Steffen Christgau and Bettina Schnor. "Software-Managed Cache Coherence for Fast One-Sided Communication". In: *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. PMAM'16. Barcelona, Spain: ACM, 2016, pp. 69–77. ISBN: 978-1-4503-4196-7. DOI: `10.1145/2883404.2883409`.

[CS17] Steffen Christgau and Bettina Schnor. "Exploring One-Sided Communication and Synchronization on a Non-Cache-Coherent Many-Core Architecture". In: *Concurrency and Computation: Practice and Experience* 29.15 (2017). ISSN: 1532-0634. DOI: `10.1002/cpe.4113`.

[Cun+14] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. "Resilient X10: Efficient Failure-aware Programming". In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014, pp. 67–80. ISBN: 978-1-4503-2656-8. DOI: `10.1145/2555243.2555248`.

[De +15] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. "Partitioned Global Address Space Languages". In: *ACM Comput. Surv.* 47.4 (May 2015), 62:1–62:27. ISSN: 0360-0300. DOI: `10.1145/2716320`.

[Den+74] Robert H. Dennard, Fritz H. Gaensslen, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268.

[Dij02] Edsger W. Dijkstra. "Cooperating Sequential Processes". In: *The Origin of Concurrent Programming*. Ed. by Per Brinch Hansen. New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 65–138. ISBN: 0-387-95401-5.

[Dur+14] Y. Durand, P. M. Carpenter, S. Adami, A. Bilas, D. Dutoit, A. Farcy, G. Gaydadjiev, J. Goodacre, M. Katevenis, M. Marazakis, E. Matus, I. Mavroidis, and J. Thomson. "EUROSERVER: Energy Efficient Node for European Micro-Servers". In: *17th Euromicro Conference on Digital System Design (DSD)*. Aug. 2014, pp. 206–213. DOI: `10.1109/DSD.2014.15`.

[Eic+92] TV Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. "Active Messages: a Mechanism for Integrated Communication and Computation". In: *Proceedings of the 19th Annual International Symposium on Computer Architecture*. IEEE. 1992, pp. 256–266.

[EL09] Andreas Ehliar and Dake Liu. “An ASIC Perspective on FPGA Optimizations”. In: *Proceedings of the 19th International Conference on Field Programmable Logic and Applications*. FPL’09. 2009, pp. 218–223.

[Fat+16] P. Fatourou, N. D. Kallimanis, E. Kanellou, O. Makridakis, and C. Symeonidou. “Efficient Distributed Data Structures for Future Many-Core Architectures”. In: *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2016, pp. 835–842. DOI: `10.1109/ICPADS.2016.0113`.

[FHB14] S. Friederich, J. Heisswolf, and J. Becker. “Hardware/software Debugging of Large Scale Many-Core Architectures”. In: *27th Symposium on Integrated Circuits and Systems Design (SBCCI)*. Sept. 2014, pp. 1–7. DOI: `10.1145/2660540.2661013`.

[Fir17] Firm Developers. *libFirm: The Graph-Based Intermediate Representation*. `http://libfirm.org`. Retrieved on 2016-10-18. 2017.

[FM12] F. Farnoud and O. Milenkovic. “Sorting of Permutations by Cost-Constrained Transpositions”. In: *IEEE Transactions on Information Theory* 58.1 (2012), pp. 3–23. DOI: `10.1109/TIT.2011.2171532`.

[FNW15] Yaosheng Fu, Tri M. Nguyen, and David Wentzlaff. “Coherence Domain Restriction on Large Scale Systems”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 686–698. ISBN: 978-1-4503-4034-2. DOI: `10.1145/2830772.2830832`.

[Fog16] Agner Fog. *The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers*. Jan. 2016. URL: `http://agner.org/optimize/microarchitecture.pdf`.

[Fre16] Freescale Semiconductor. *AltiVec™ Technology Programming Interface Manual*. Revision 0. June 2016.

[Fri+13] Andrew Friedley, Torsten Hoefler, Greg Bronevetsky, Andrew Lumsdaine, and Ching-Chen Ma. “Ownership Passing: Efficient Distributed Memory Programming on Multi-core Systems”. In: *Proceedings of the 18th ACM SIGPLAN Symposium* *on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: ACM, 2013, pp. 177–186. ISBN: 978-1-4503-1922-5. DOI: 10.1145/2442516.2442534.

[Fri16] Stephanie Friederich. "Automated Hardware Prototyping for 3D Network on Chips". PhD thesis. Karlsruher Institut für Technologie, 2016.

[FSA97] S. Forrest, A. Somayaji, and D. H. Ackley. "Building Diverse Computer Systems". In: *The Sixth Workshop on Hot Topics in Operating Systems*. May 1997, pp. 67–72. DOI: 10.1109/HOTOS.1997.595185.

[GH07] Daniel Grund and Sebastian Hack. "A Fast Cutting-plane Algorithm for Optimal Coalescing". In: *Proceedings of the 16th International Conference on Compiler Construction*. CC'07. Braga, Portugal: Springer-Verlag, 2007, pp. 111–125. ISBN: 978-3-540-71228-2.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1990.

[GN15] Suyash Gupta and V. Krishna Nandivada. "IMSuite: A Benchmark Suite for Simulating Distributed Algorithms". In: *Journal of Parallel and Distributed Computing* 75 (2015), pp. 1–19. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.10.010.

[Gos+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. First Edition. Addison-Wesley Professional, 2014.

[Gro+11] David Grove, Olivier Tardieu, David Cunningham, Ben Herta, Igor Peshansky, and Vijay Saraswat. "A Performance Model for X10 Applications". In: *ACM SIGPLAN 2011 X10 Workshop*. San Jose, California, June 2011.

[GRR00] T. Grundmann, M. Ritt, and W. Rosenstiel. "TPO++: an Object-Oriented Message-Passing Library in C++". In: *Proceedings of the International Conference on Parallel Processing*. 2000, pp. 43–50. DOI: 10.1109/ICPP.2000.876070.

[Gru+15] Charles Gruenwald III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. "Hare: A File System for Non-cache-coherent Multicores". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 30:1–30:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741959.

[Gur12] Gurobi Optimization Inc. *Gurobi Optimizer Reference Manual*. 2012. URL: http://www.gurobi.com.

[Hac07] Sebastian Hack. "Register Allocation for Programs in SSA Form". PhD thesis. Universität Karlsruhe, 2007. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/6532.

[Han+11] Frank Hannig, Sascha Roloff, Gregor Snelting, Jürgen Teich, and Andreas Zwinkau. "Resource-Aware Programming and Simulation of MPSoC Architectures through Extension of X10". In: *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '11. New York, NY, USA: ACM, June 2011, pp. 48–55. DOI: 10.1145/1988932.1988941.

[Han+14] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. "Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach". In: *ACM Trans. Embed. Comput. Syst.* 13.4s (Apr. 2014), 133:1–133:29. ISSN: 1539-9087. DOI: 10.1145/2584660.

[Hei+14] J. Heisswolf et al. "The Invasive Network on Chip - A Multi-Objective Many-Core Communication Infrastructure". In: *Proceedings of the 27th International Conference on Architecture of Computing Systems*. ARCS. Feb. 2014, pp. 1–8.

[Hei14] Jan Heißwolf. "A Scalable and Adaptive Network on Chip for Many-Core Architectures". Karlsruhe, KIT, Diss., 2014. PhD thesis. Karlsruher Institut für Technologie (KIT), Nov. 2014.

[Hen+12] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hübner, R. K. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. "Invasive Manycore Architectures". In: *17th*

Asia and South Pacific Design Automation Conference. Jan. 2012, pp. 193–200. DOI: 10.1109/ASPDAC.2012.6164944.

[Hen00] John L. Henning. "SPEC CPU2000: Measuring CPU Performance in the New Millennium". In: *Computer* 33.7 (July 2000), pp. 28–35. ISSN: 0018-9162. DOI: 10.1109/2.869367.

[HG08] Sebastian Hack and Gerhard Goos. "Copy Coalescing by Graph Recoloring". In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 227–237. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375610.

[HGG06] Sebastian Hack, Daniel Grund, and Gerhard Goos. "Register Allocation for Programs in SSA-Form". In: *Proceedings of the 15th International Conference on Compiler Construction*. CC'06. Vienna, Austria: Springer-Verlag, 2006, pp. 247–262. ISBN: 978-3-540-33050-9. DOI: 10.1007/11688839_20.

[Hof05] H. Peter Hofstee. "Power Efficient Processor Architecture and The Cell Processor". In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. HPCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 258–262. ISBN: 0-7695-2275-0. DOI: 10.1109/HPCA.2005.26.

[How+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. "A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS". In: *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*. Feb. 2010, pp. 108–109. DOI: 10.1109/ISSCC.2010.5434077.

[How+16] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. *Linux Kernel Memory Barriers*. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/memory-barriers.txt?id=HEAD. 2016.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach*. Elsevier, 2011.

[IBM14] IBM. *Elastic X10*. http://x10-lang.org/documentation/practical-x10-programming/elastic-x10.html. 2014.

[Int12] Intel Corporation. *The SCC Platform Overview*. https://communities.intel.com/docs/DOC-5512. Revision 0.80, retrieved on 2016-10-17. 2012.

[Int16] Intel Corporation. *Intel® 64 and IA-32 Architecture Optimization Reference Manual*. Jan. 2016.

[Int17] Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference*. 319433-030. Oct. 2017.

[Jou+98] Stephen Jourdan, Ronny Ronen, Michael Bekerman, Bishara Shomar, and Adi Yoaz. "A Novel Renaming Scheme to Exploit Value Temporal Locality Through Physical Register Reuse and Unification". In: *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*. MICRO 31. Dallas, Texas, USA: IEEE Computer Society Press, 1998, pp. 216–225. ISBN: 1-58113-016-3.

[KAC14] Rakesh Komuravelli, Sarita V. Adve, and Ching-Tsun Chou. "Revisiting the Complexity of Hardware Cache Coherence and Some Implications". In: *ACM Trans. Archit. Code Optim.* 11.4 (Dec. 2014), 37:1–37:22. ISSN: 1544-3566. DOI: 10.1145/2663345.

[Kah+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. "Introduction to the Cell Multiprocessor". In: *IBM J. Res. Dev.* 49.4/5 (July 2005), pp. 589–604. ISSN: 0018-8646.

[KCT12] Rob Knauerhase, Romain Cledat, and Justin Teller. "For Extreme Parallelism, Your OS Is Sooooo Last-Millennium". In: *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism*. Berkeley, CA: USENIX, 2012. URL: https://www.usenix.org/conference/hotpar12/extreme-parallelism-your-os-sooooo-last-millennium.

[Kel+10] John H. Kelm, Daniel R. Johnson, William Tuohy, Steven S. Lumetta, and Sanjay J. Patel. "Cohesion: A Hybrid Memory Model for Accelerators". In: *Proceedings of the 37th Annual International Symposium on Computer Architecture*. ISCA '10.

Saint-Malo, France: ACM, 2010, pp. 429–440. ISBN: 978-1-4503-0053-7. DOI: 10.1145/1815961.1816019.

[KHS12] Fredrik Kjolstad, Torsten Hoefler, and Marc Snir. "Automatic Datatype Generation and Optimization". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 327–328. ISBN: 978-1-4503-1160-1. DOI: 10.1145/2145816.2145878.

[KK10] S. Kaxiras and G. Keramidas. "SARC Coherence: Scaling Directory Cache Coherence in Performance and Power". In: *IEEE Micro* 30.5 (Sept. 2010), pp. 54–65. ISSN: 0272-1732. DOI: 10.1109/MM.2010.82.

[KL02] A. J. KleinOsowski and David J. Lilja. "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research". In: *IEEE Computer Architecture Letters* 1.1 (Jan. 2002), p. 7. ISSN: 1556-6056. DOI: 10.1109/L-CA.2002.8.

[Kob+11] Sebastian Kobbe, Lars Bauer, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Jörg Henkel. "DistRM: Distributed Resource Management for On-chip Many-core Systems". In: *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '11. Taipei, Taiwan: ACM, 2011, pp. 119–128. ISBN: 978-1-4503-0715-4. DOI: 10.1145/2039370.2039392.

[Kor16] Peter Korsgaard. *Buildroot Linux*. 2005–2016. URL: https://buildroot.org/.

[Kum+11] Rakesh Kumar, Timothy G. Mattson, Gilles Pokam, and Rob Van Der Wijngaart. "The Case for Message Passing on Many-Core Chips". In: *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Ed. by Michael Hübner and Jürgen Becker. New York, NY: Springer New York, 2011, pp. 115–123. ISBN: 978-1-4419-6460-1. DOI: 10.1007/978-1-4419-6460-1_5.

[Kum+14] Vivek Kumar, Yili Zheng, Vincent Cavé, Zoran Budimlić, and Vivek Sarkar. "HabaneroUPC++: A Compiler-free PGAS Library". In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. PGAS '14. Eugene, OR, USA: ACM, 2014, 5:1–5:10. ISBN: 978-1-4503-3247-7. DOI: 10.1145/2676870.2676879.

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO '04. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–86. ISBN: 0-7695-2102-9.

[Li+16] Y. Li, K. Duan, Y. Lv, J. Pang, and S. Cai. "A Novel Approach to Parameterized Verification of Cache Coherence Protocols". In: *34th International Conference on Computer Design*. ICCD '16. Oct. 2016, pp. 560–567. DOI: 10.1109/ICCD.2016.7753341.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999.

[Loc12] Andreas Lochbihler. "A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012. DOI: 10.5445/KSP/1000028867. URL: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028867.

[Lot+12] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, and Babak Falsafi. "Scale-out Processors". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 500–511. ISBN: 978-1-4503-1642-2.

[LWZ14] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. "K2: A Mobile Operating System for Heterogeneous Coherence Domains". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*.

ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 285–300. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541975.

[Lyb+12a] S. Lyberis, G. Kalokerinos, M. Lygerakis, V. Papaefstathiou, D. Tsaliagkos, M. Katevenis, D. Pnevmatikatos, and D. Nikolopoulos. "Formic: Cost-efficient and Scalable Prototyping of Manycore Architectures". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. Apr. 2012, pp. 61–64. DOI: 10.1109/FCCM.2012.20.

[Lyb+12b] Spyros Lyberis, Polyvios Pratikakis, Dimitrios S. Nikolopoulos, Martin Schulz, Todd Gamblin, and Bronis R. de Supinski. "The Myrmics Memory Allocator: Hierarchical,Message-passing Allocation for Global Address Spaces". In: *Proceedings of the 2012 International Symposium on Memory Management*. ISMM '12. Beijing, China: ACM, 2012, pp. 15–24. ISBN: 978-1-4503-1350-6. DOI: 10.1145/2258996.2259001.

[Lyb+16] Spyros Lyberis, Polyvios Pratikakis, Iakovos Mavroidis, and Dimitrios S. Nikolopoulos. "Myrmics: Scalable, Dependency-aware Task Scheduling on Heterogeneous Manycores". In: *CoRR* abs/1606.04282 (2016). URL: http://arxiv.org/abs/1606.04282.

[Mac11] C. A. Mack. "Fifty Years of Moore's Law". In: *IEEE Transactions on Semiconductor Manufacturing* 24.2 (May 2011), pp. 202–207. ISSN: 0894-6507. DOI: 10.1109/TSM.2010.2096437.

[Mat+10] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. "The 48-core SCC Processor: The Programmer's View". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: 10.1109/SC.2010.53.

[Mec16] Michael Mechler. "Flexibles, partielles Parametrisieren von softwaredefinierten Bereichen des adaptiven Caches zur Laufzeit in einem Shared-Memory Multi-/Many-Core-System". In German. MA thesis. Karlsruhe Institute of Technology, June 2016.

[Mes15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. Website. Version 3.1. June 2015. URL: `http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf`.

[MHS12] Milo M. K. Martin, Mark D. Hill, and Daniel J. Sorin. "Why On-chip Cache Coherence is Here to Stay". In: *Communications of the ACM* 55.7 (July 2012), pp. 78–89. ISSN: 0001-0782. DOI: `10.1145/2209249.2209269`.

[MMT16] Louis Mandel, Josh Milthorpe, and Olivier Tardieu. "Control Structure Overloading in X10". In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. X10'16. Santa Barbara, CA, USA: ACM, 2016, pp. 1–6. ISBN: 978-1-4503-4386-2. DOI: `10.1145/2931028.2931032`.

[Mod13] Tobias Modschiedler. "Erweiterung der LEON3-CPU um einen Permutationsregistersatz zum beschleunigten Abbau der SSA-Zwischendarstellung". In German. MA thesis. Karlsruhe Institute of Technology, Aug. 2013.

[Moh+13] Manuel Mohr, Artjom Grudnitsky, Tobias Modschiedler, Lars Bauer, Sebastian Hack, and Jörg Henkel. "Hardware Acceleration for Programs in SSA Form". In: *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. CASES'13. Piscataway, NJ, USA: IEEE Press, 2013, 14:1–14:10. DOI: `10.1109/CASES.2013.6662518`.

[Moh+15] Manuel Mohr, Sebastian Buchwald, Andreas Zwinkau, Christoph Erhardt, Benjamin Oechslein, Jens Schedel, and Daniel Lohmann. "Cutting out the Middleman: OS-Level Support for X10 Activities". In: *Proceedings of the ACM SIGPLAN Workshop on X10*. X10'15. Portland, OR, USA: ACM, 2015, pp. 13–18. ISBN: 978-1-4503-3586-7. DOI: `10.1145/2771774.2771775`.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. "The Java Memory Model". In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. Long Beach, California, USA: ACM, 2005, pp. 378–391. ISBN: 1-58113-830-X. DOI: `10.1145/1040305.1040336`.

[MS10] Ross McIlroy and Joe Sventek. “Hera-JVM: A Runtime System for Heterogeneous Multi-core Architectures”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 205–222. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869478.

[MSM04] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. First edition. Addison-Wesley Professional, 2004. ISBN: 0321228111.

[MT17] Manuel Mohr and Carsten Tradowsky. “Pegasus: Efficient Data Transfers for PGAS Languages on Non-Cache-Coherent Many-Cores”. In: *Proceedings of Design, Automation and Test in Europe Conference Exhibition*. DATE’17. IEEE, Mar. 2017, pp. 1781–1786. DOI: 10.23919/DATE.2017.7927281.

[Myt+09] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. “Producing Wrong Data Without Doing Anything Obviously Wrong!” In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 265–276. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508275.

[NL91] Bill Nitzberg and Virginia Lo. “Distributed Shared Memory: A Survey of Issues and Algorithms”. In: *Computer* 24.8 (Aug. 1991), pp. 52–60. ISSN: 0018-9162. DOI: 10.1109/2.84877.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a Proof Assistant for Higher-Order Logic*. Vol. 2283. Springer Science & Business Media, 2002.

[Nür+14] Stefan Nürnberger, Gabor Drescher, Randolf Rotta, Jörg Nolte, and Wolfgang Schröder-Preikschat. “Shared Memory in the Many-Core Age”. In: *European Conference on Parallel Processing*. Springer. 2014, pp. 351–362.

[Nys+08] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. “Constrained Types for Object-oriented Languages”. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applica-*

tions. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 457–474. ISBN: 978-1-60558-215-3. DOI: `10.1145/1449764.1449800`.

[Ode14] Martin Odersky. *The Scala Language Specification v 2.9*. 2014.

[Oec+11] Benjamin Oechslein, Jens Schedel, Jürgen Kleinöder, Lars Bauer, Jörg Henkel, Daniel Lohmann, and Wolfgang Schröder-Preikschat. "OctoPOS: A Parallel Operating System for Invasive Computing". In: *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures*. Ed. by Ross McIlroy, Joe Sventek, Tim Harris, and Timothy Roscoe. Vol. USB Proceedings. SFMA '11. Salzburg, 2011, pp. 9–14.

[Ope17] OpenACC Group. *The OpenACC Application Program Interface*. 2017. URL: `http://www.openacc.org/`.

[Ora16] Oracle. *The Java Remote Method Invocation API (Java RMI)*. 2016. URL: `http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/`.

[Pet+11a] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. "Early Experience with the Barrelfish OS and the Single-Chip Cloud Computer". In: *MARC Symposium*. 2011, pp. 35–39.

[Pet+11b] Simon Peter, Adrian Schüpbach, Dominik Menzi, and Timothy Roscoe. "Early experience with the Barrelfish OS and the Single-Chip Cloud Computer". In: *Proceedings of the 3rd Intel Multicore Applications Research Community Symposium (MARC)*. Ettlingen, Germany, July 2011.

[Pha+05] D Pham, S Asano, M Bolliger, MN Day, HP Hofstee, C Johns, J Kahle, A Kameyama, J Keaty, Y Masubuchi, et al. "The Design and Implementation of a First-generation CELL Processor – a Multi-core SoC". In: *2005 International Conference on Integrated Circuit Design and Technology*. ICICDT '05. IEEE. 2005, pp. 49–52.

[Phi11] Michael Philippsen. "JavaParty". In: *Encyclopedia of Parallel Computing*. Springer, 2011, pp. 992–997.

[PM04] Jinpyo Park and Soo-Mook Moon. "Optimistic Register Coalescing". In: *ACM Transactions on Programming Languages and Systems* 26.4 (July 2004), pp. 735–765. ISSN: 0164-0925. DOI: 10.1145/1011508.1011512.

[PN14] Anastasios Papagiannis and Dimitrios S. Nikolopoulos. "Hybrid Address Spaces: A Methodology for Implementing Scalable High-Level Programming Models on Non-Coherent Many-Core Architectures". In: *Journal of Systems and Software* 97.Supplement C (2014), pp. 47–64. ISSN: 0164-1212. DOI: 10.1016/j.jss.2014.06.058.

[Pöp+17] Alexander Pöppl, Marvin Damschen, Florian Schmaus, Andreas Fried, Manuel Mohr, Matthias Blankertz, Lars Bauer, Jörg Henkel, Wolfgang Schröder-Preikschat, and Michael Bader. "Shallow Water Waves on a Deep Technology Stack: Accelerating a Finite Volume Tsunami Model using Reconfigurable Hardware in Invasive Computing". In: *Euro-Par 2017: Parallel Processing Workshops*. Lecture Notes in Computer Science. Heidelberg, Berlin: Springer-Verlag, Aug. 2017.

[Pow+13] Jason Power, Arkaprava Basu, Junli Gu, Sooraj Puthoor, Bradford M. Beckmann, Mark D. Hill, Steven K. Reinhardt, and David A. Wood. "Heterogeneous System Coherence for Integrated CPU-GPU Systems". In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 457–467. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540747.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. "Register Allocation via Coloring of Chordal Graphs". In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 315–329. ISBN: 978-3-540-29735-2. DOI: 10.1007/11575467_21.

[PP84] Mark S. Papamarcos and Janak H. Patel. "A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories". In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. ISCA '84. New York, NY, USA: ACM, 1984, pp. 348–354. ISBN: 0-8186-0538-3. DOI: 10.1145/800015.808204.

[PRN11] Thomas Prescher, Randolf Rotta, and Jörg Nolte. "Flexible Sharing and Replication Mechanisms for Hybrid Memory Architectures". In: *Proceedings of the 4th Many-Core Applications Research Community Symposium*. Vol. 55. MARC. 2011, pp. 67–72.

[Rav15] Andreas Herkersdorf Ravi Kumar Pujari Thomas Wild. "A Hardware-based Multi-objective Thread Mapper for Tiled Manycore Architectures". In: *33rd IEEE International Conference on Computer Design*. ICCD '15. New York, Oct. 2015, pp. 459–462. DOI: 10.1109/ICCD.2015.7357148.

[RCL13] Pablo Reble, Carsten Clauss, and Stefan Lankes. "One-sided Communication and Synchronization for Non-coherent Memory-coupled Cores". In: *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE. 2013, pp. 390–397.

[Rot+12] Randolf Rotta, Thomas Prescher, Jana Traue, and Jörg Nolte. "Data Sharing Mechanisms for Parallel Graph Algorithms on the Intel SCC". In: *Proceedings of the 6th Many-core Applications Research Community Symposium*. MARC. ONERA, The French Aerospace Lab. 2012, pp. 13–18.

[Rot11] Randolf Rotta. "On Efficient Message Passing on the Intel SCC". In: *3rd Many-core Applications Research Community (MARC) Symposium*. Vol. 7598. KIT Scientific Publishing. 2011.

[RSL08] Laurence Rideau, Bernard Paul Serpette, and Xavier Leroy. "Tilting at Windmills with Coq: Formal Verification of a Compilation Algorithm for Parallel Moves". In: *Journal of Automated Reasoning* 40.4 (May 2008), pp. 307–326. ISSN: 0168-7433. DOI: 10.1007/s10817-007-9096-8.

[Sar+10] Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. "The Asynchronous Partitioned Global Address Space Model". In: *The First Workshop on Advances in Message Passing*. 2010, pp. 1–8.

[Sar+16] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. *X10 Language Specification*. Tech. rep. IBM, June 2016. URL: http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf.

[Ser03] Ákos Seress. *Permutation Group Algorithms*. Vol. 152. Cambridge University Press, 2003.

[SHW11] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. 1st. Morgan & Claypool Publishers, 2011. ISBN: 9781608455645.

[Sim00] Dezsö Sima. "The Design Space of Register Renaming Techniques". In: *IEEE Micro* 20.5 (Sept. 2000), pp. 70–83. ISSN: 0272-1732. DOI: 10.1109/40.877952.

[Sin+13] Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O'Connor, and Tor M. Aamodt. "Cache Coherence for GPU Architectures". In: *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*. HPCA '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 578–590. ISBN: 978-1-4673-5585-8. DOI: 10.1109/HPCA.2013.6522351.

[SPA92] SPARC International Inc. *The SPARC Architecture Manual, Version 8*. Revision SAV080SI9308. 1992.

[Sri+17] Akshay Srivatsa, Sven Rheindt, Thomas Wild, and Andreas Herkersdorf. "Region Based Cache Coherence for Tiled MPSoCs". In: *30th IEEE International System-on-Chip Conference*. SOCC '17. Munich, Germany, 2017.

[Sta81] Richard P. Stanley. "Factorization of Permutations into n-Cycles". In: *Discrete Mathematics* 37.2–3 (1981), pp. 255–262. DOI: 10.1016/0012-365X(81)90224-7.

[Ste90] Per Stenström. "A Survey of Cache Coherence Schemes for Multiprocessors". In: *Computer* 23.6 (June 1990), pp. 12–24. ISSN: 0018-9162. DOI: 10.1109/2.55497.

[Sto+14] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. "AEminium: A Permission Based Concurrent-by-Default Programming Language Approach". In: *ACM Transactions on Programming Languages and Systems*. TOPLAS 36.1 (Mar. 2014), 2:1–2:42. DOI: 10.1145/2543920.

[Str96] Volker Strehl. "Minimal Transitive Products of Transpositions: the Reconstruction of a Proof of A. Hurwitz". In: *Séminaire Lotharingien De Combinatoire* 37 (1996).

[Sut12] Herb Sutter. *Welcome to the Jungle*. https://herbsutter.com/welcome-to-the-jungle/. 2012.

[Syn15] Synopsis Inc. *CHIPit Platinum Edition and HAPS-600 Series ASIC Emulation and Rapid Prototyping System – Hardware Reference Manual*. 2015.

[SZG13] Joshua Suettlerlein, Stéphane Zuckerman, and Guang R. Gao. "An Implementation of the Codelet Model". In: *Proceedings of the 19th International Conference on Parallel Processing*. Euro-Par'13. Aachen, Germany: Springer-Verlag, 2013, pp. 633–644. DOI: 10.1007/978-3-642-40047-6_63.

[Tak+11] Mikio Takeuchi, Yuki Makino, Kiyokuni Kawachiya, Hiroshi Horii, Toyotaro Suzumura, Toshio Suganuma, and Tamiya Onodera. "Compiling X10 to Java". In: *ACM SIGPLAN 2011 X10 Workshop*. San Jose, California, June 2011.

[Tar72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160.

[Tav+16] Sanket Tavarageri, Wooil Kim, Josep Torrellas, and P Sadayappan. "Compiler Support for Software Cache Coherence". In: *23rd IEEE International Conference on High Performance Computing*. HiPC '16. IEEE, Dec. 2016, pp. 341–350. DOI: 10.1109/HiPC.2016.047.

[Tei+11] Jürgen Teich, Jörg Henkel, Andreas Herkersdorf, Doris Schmitt-Landsiedel, Wolfgang Schröder-Preikschat, and Gregor Snelting. "Invasive Computing: An Overview". In: *Multiprocessor System-on-Chip – Hardware Design and Tool Integration*. Ed. by

M. Hübner and J. Becker. Springer, Berlin, Heidelberg, 2011, pp. 241–268.

[Tei+16] Jürgen Teich et al. *Transregional Collaborative Research Center 89: Invasive Computing (InvasIC)*. `http://www.invasic.de`. 2016.

[Tex14] Texas Instruments. *OMAP4470 Multimedia Device Technical Reference Manual*. `http://www.ti.com/product/OMAP4470/technicaldocuments`. Silicon Revision 1.0, Texas Instruments OMAP Family of Products, Version T, retrieved on 2017-01-12. 2014.

[TM97] Igor Tartalja and Veljko Milutinović. "Classifying Software-Based Cache Coherence Solutions". In: *IEEE Softw.* 14.3 (May 1997), pp. 90–101. ISSN: 0740-7459. DOI: `10.1109/52.589244`.

[Tol+11] Michiel W. van Tol, Roy Bakker, Merjin Verstraaten, Clemens Grelck, and Chris R. Jesshope. "Efficient Memory Copy Operations on the 48-core Intel SCC Processor". In: *3rd Many-core Applications Research Community (MARC) Symposium*. KIT Scientific Publishing. 2011. ISBN: 9783866447172.

[Tom67] R. M. Tomasulo. "An Efficient Algorithm for Exploiting Multiple Arithmetic Units". In: *IBM J. Res. Dev.* 11.1 (Jan. 1967), pp. 25–33. ISSN: 0018-8646. DOI: `10.1147/rd.111.0025`.

[Tra16] Carsten Tradowsky. "Methoden zur applikationsspezifischen Effizienzsteigerung adaptiver Prozessorplattformen". In German. PhD thesis. Karlsruhe Institute of Technology, 2016. DOI: `10.5445/IR/1000067258`.

[TWL12] Olivier Tardieu, Haichuan Wang, and Haibo Lin. "A Work-stealing Scheduler for X10's Task Parallelism with Suspension". In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '12. New Orleans, Louisiana, USA: ACM, 2012, pp. 267–276. DOI: `10.1145/2145816.2145850`.

[URK] Isaías A. Comprés Ureña, Michael Riepen, and Michael Konow. "RCKMPI – Lightweight MPI Implementation for Intel's Single-chip Cloud Computer (SCC)". In: *Proceedings of* *the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*. EuroMPI '11. Santorini, Greece: Springer-Verlag, pp. 208–217. ISBN: 978-3-642-24448-3.

[VP03] R. Veldema and M. Philippsen. "Compiler Optimized Remote Method Invocation". In: *Proceedings of the International Conference on Cluster Computing*. Dec. 2003, pp. 127–136. DOI: 10.1109/CLUSTR.2003.1253308.

[War02] Henry S. Warren. *Hacker's Delight*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201914654.

[WBJ16] Joss Whittle, Rita Borgo, and Mark W. Jones. "Implementing Generalized Deep-Copy in MPI". In: *PeerJ Computer Science* 2 (2016), e95.

[WF10] Christian Wimmer and Michael Franz. "Linear Scan Register Allocation on SSA Form". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO '10. Toronto, Ontario, Canada: ACM, 2010, pp. 170–179. ISBN: 978-1-60558-635-9. DOI: 10.1145/1772954.1772979.

[Wil+16] Stefan Wildermann, Michael Bader, Lars Bauer, Marvin Damschen, Dirk Gabriel, Michael Gerndt, Michael Glaß, Jörg Henkel, Johny Paul, Alexander Pöppl, Sascha Roloff, Tobias Schwarzer, Gregor Snelting, Walter Stechele, Jüurgen Teich, Andreas Weichslgartner, and Andreas Zwinkau. "Invasive Computing for Timing-Predictable Stream Processing on MPSoCs". In: *it – Information Technology* 58.6 (2016), pp. 267–280. ISSN: 1611-2776. DOI: 10.1515/itit-2016-0021.

[Xil16] Xilinx. *Xilinx University Program XUPV5-LX110T Development System*. 2016. URL: http://www.xilinx.com/univ/xupv5-lx110t.htm.

[Zai+15] Aurang Zaib, Jan Heißwolf, Andreas Weichslgartner, Thomas Wild, Jürgen Teich, Jürgen Becker, and Andreas Herkersdorf. "Network Interface with Task Spawning Support for NoC-Based DSM Architectures". In: *Architecture of Computing Systems*. ARCS '15. Porto, Portugal: Springer, 2015, pp. 186–198.

[ZBS13] Andreas Zwinkau, Sebastian Buchwald, and Gregor Snelting. *InvadeX10 Documentation v0.5*. Tech. rep. 7. Karlsruhe Institute of Technology, 2013. URL: `http://pp.info.uni-karlsruhe.de/~zwinkau/invadeX10-0.5/manual.pdf`.

[ZP14] Foivos S. Zakkak and Polyvios Pratikakis. "JDMM: A Java Memory Model for Non-cache-coherent Memory Architectures". In: *Proceedings of the 2014 International Symposium on Memory Management*. ISMM '14. Edinburgh, United Kingdom: ACM, 2014, pp. 83–92. ISBN: 978-1-4503-2921-7. DOI: `10.1145/2602988.2602999`.

[ZP16a] Foivos S. Zakkak and Polyvios Pratikakis. "Building a Java™ Virtual Machine for Non-Cache-Coherent Many-core Architectures". In: *Proceedings of the 14th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '16. Lugano, Switzerland: ACM, 2016, 1:1–1:10. ISBN: 978-1-4503-4800-3. DOI: `10.1145/2990509.2990510`.

[ZP16b] Foivos S. Zakkak and Polyvios Pratikakis. "DiSquawk: 512 Cores, 512 Memories, 1 JVM". In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '16. Lugano, Switzerland: ACM, 2016, 2:1–2:12. ISBN: 978-1-4503-4135-6. DOI: `10.1145/2972206.2972212`.

[Zuc+11] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. "Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper". In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT '11. San Jose, California, USA: ACM, 2011, pp. 64–69. DOI: `10.1145/2000417.2000424`.

Index

*Alles hat ein Ende, nur der Compiler
hat drei.*