Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture

by

Erik Hansson

Linköping University
INSTITUTE OF TECHNOLOGY

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2014

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.

A doctor's degree comprises 240 ECTS credits (4 year of full-time studies).

A licentiate's degree comprises 120 ECTS credits.

Copyright © 2014 Erik Hansson

ISBN 978-91-7519-189-8

ISSN 0280-7971

Printed by LiU Tryck 2014

URL: <http://urn.kb.se/resolve?urn:nbn:se:liu:diva-111333>

Abstract

In this thesis we describe techniques for code generation and global optimization for a PRAM-NUMA multicore architecture. We specifically focus on the REPLICa architecture which is a family massively multithreaded very long instruction word (VLIW) chip multiprocessors with chained functional units that has a reconfigurable emulated shared on-chip memory. The on-chip memory system supports two execution modes, PRAM and NUMA, which can be switched between at run-time. PRAM mode is considered the standard execution mode and targets mainly applications with very high thread level parallelism (TLP). In contrast, NUMA mode is optimized for sequential legacy applications and applications with low amount of TLP. Different versions of the REPLICa architecture have different number of cores, hardware threads and functional units. In order to utilize the REPLICa architecture efficiently we have made several contributions to the development of a compiler for REPLICa target code generation. It supports both code generation for PRAM mode and NUMA mode and can generate code for different versions of the processor pipeline (i.e. for different numbers of functional units). It includes optimization phases to increase the utilization of the available functional units. We have also contributed to quantitative the evaluation of PRAM and NUMA mode. The results show that PRAM mode often suits programs with irregular memory access patterns and control flow best while NUMA mode suites regular programs better. However, for a particular program it is not always obvious which mode, PRAM or NUMA, will show best performance. To tackle this we contributed a case study for generic stencil computations, using machine learning derived cost models in order to automatically select at runtime which mode to execute in. We extended this to also include a sequence of kernels.

This work has been supported by VTT Oulu, Finland and SeRC.

Acknowledgement

First of all I want to thank my supervisor Christoph Kessler for his help, constructive feedback and patience. Secondly, I want to thank Martti Forsell, VTT Oulu, for letting me work in the REPLICA project and for sharing his knowledge about computer architecture. Then I want to thank Kristian Sandahl for his support. I also want to thank everyone at PELAB, my colleagues at the other labs, and the administrative staff. Finally I want to thank all my friends and family.

Erik Hansson, Linköping, October 2014

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Contributions	2
1.3	List of Publications	2
1.4	Outline	4
2	REPLICA architecture	5
2.1	Introduction and motivation	5
2.2	Related work	8
2.3	Architecture	10
2.3.1	Memory abstraction	10
2.3.2	Execution model	11
2.3.3	Specific special operations	11
2.3.4	Assembly programming model	12
2.4	REPLICA language	13
2.4.1	Compilation model	14
2.4.2	Language constructs	16
2.4.3	Kernel example	19
2.5	Summary	20
3	Instruction scheduling	25
3.1	Introduction	25
3.2	Dependency Graph	25
3.3	Register Compression	27
3.4	ILP scheduling	28
3.4.1	Algorithm	28
3.4.2	Parametrization	31
3.5	Evaluation	32
3.6	Comparison to Previous Work	34
3.6.1	Original virtual ILP algorithm	34
3.6.2	Earlier ILP scheduler for the REPLICA compiler	35
3.7	Conclusion and future work	35

4	Evaluation of REPLICA PRAM mode	37
4.1	Motivation to evaluate PRAM	37
4.2	Hardware Architectures	38
4.2.1	Intel Xeon CPU	38
4.2.2	NVidia Tesla GPGPU	38
4.3	Evaluation of PRAM mode	39
4.4	Conclusions	44
5	Exploiting NUMA mode	47
5.1	Introduction	47
5.2	Configurable ESM architectures	47
5.3	Adding NUMA support	48
5.4	NUMA realization alternatives	49
5.5	Programming considerations	51
5.6	Supporting the NUMA mode	54
5.6.1	Compiler support	56
5.6.2	NUMA optimizations	57
5.7	Evaluation	59
5.7.1	Code size and programmability	61
5.8	Conclusions	61
6	Automated mode selection	67
6.1	Introduction to execution mode selection	67
6.2	Parameterized Benchmark	70
6.3	Machine learning models	71
6.3.1	Eureqa Pro	72
6.3.2	C5.0 decision trees	73
6.4	Global optimization model	73
6.5	Eureqa Pro and C5.0 models evaluation	74
6.6	Evaluation and results for optimized global composition	78
6.7	Optimizing mode selection for loops	79
6.8	Possible extension towards structural composition	83
6.9	Related work	85
6.10	Conclusion	87
6.10.1	Mode selection	87
6.10.2	Global optimization	87
6.11	Future work	88
7	Concluding remarks	89
8	Glossary	91

Chapter 1

Introduction

1.1 Motivation

During the last decade multicore computers have become mainstream as an attempt to keep up with the never ending demand of more performance. The main reasons why industry now focuses on parallel architectures are that it is hard to increase clock speed without getting heat and energy problems and that instruction level parallelism usually is limited. There exist different types of parallel computer architectures; traditional examples are non-uniform memory access (NUMA) and single instruction multiple data (SIMD) and Later version of Intel Xeon CPUs [55] is are NUMA architectures and have SIMD instructions.

However, in order to achieve any performance increase from any kind of multicore architecture sequential, legacy programs have to be rewritten to utilize the parallel features. One drawback of traditional parallel architectures is that they do not provide desirable features such as strong and deterministic execution when executing parallel programs.

An exception to this are so-called emulated shared memory architectures (ESMs). They provide a synchronous programming model which follows the parallel random access machine (PRAM) model, which is a well known theoretical model from the literature [58]. Even though the PRAM model is mainly considered as a theoretical model, there are examples of realizing it in hardware [96, 58].

Here we specifically focus on another ESM architecture, namely the REPLICA architecture [42, 38]. It is a family of massively multithreaded very long instruction word (VLIW) chip multiprocessors with chained functional units that has a reconfigurable emulated shared on-chip memory subsystem.

Compared to other ESMs the on-chip memory system supports two execution modes, PRAM and NUMA, which can be switched between at runtime. PRAM mode is considered the standard execution mode and targets

mainly applications with very high thread level parallelism (TLP). In contrast, NUMA mode is optimized for sequential legacy applications and applications with a low amount of TLP. Different versions of the REPLICAs architecture have different number of cores, hardware threads and functional units.

1.2 Contributions

The scope of this work has been on optimized code generation, on different levels, for the REPLICAs architecture and its evaluation:

- A configurable REPLICAs baseline language compiler. We designed and implemented a LLVM-based compiler targeting REPLICAs PRAM and NUMA mode. It supports virtually any number of functional units in the configurable pipeline and also contains optimization phases to increase the utilization of the available functional units.
- Evaluation of REPLICAs architecture and REPLICAs baseline compiler and code generation. This includes contributions to the quantitative evaluation of PRAM and NUMA mode. The results show that PRAM mode often suits programs with irregular memory access patterns and control flow best while NUMA mode suits regular programs better.
- Contributions to a methodology how to select, in an optimized way, which execution mode to use (PRAM or NUMA) for best performance. We contributed a case study for generic stencil computations, using machine learning derived cost models in order to automatically select at runtime which mode (PRAM or NUMA) to execute in. We extended this to also include a sequence of kernels.

1.3 List of Publications

Scientific papers, fully or partly, covered in this work are the following:

- Jari-Matti Mäkelä, Erik Hansson, Daniel Åkesson, Martti Forsell, Christoph Kessler, Ville Leppänen: Design of the Language Replica for Hybrid PRAM-NUMA Many-Core Architectures. *Proc. ISPA 2012 4th IEEE International Workshop on Multicore and Multithreaded Architectures and Algorithms*, Washington DC, USA. 2012. [72]
- Christoph Kessler, Erik Hansson: Flexible Scheduling and Thread Allocation for Synchronous Parallel Tasks. *Proc. PASA-2012, München, Germany*, Feb. 2012. [60]
- Martin Kessler, Erik Hansson, Daniel Åkesson, Christoph Kessler: Exploiting Instruction Level Parallelism for REPLICAs - A Configurable

- VLIW Architecture With Chained Functional Units. *Proc. 18th Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA '12)*, Las Vegas, USA. July 2012. [64]
- Erik Hansson, Erik Alnervik, Christoph Kessler, Martti Forsell: A Quantitative Comparison of Emulated Shared Memory Architectures to Current Multicore CPUs and GPUs. *Proc. MCC'13, Sixth Swedish Workshop on Multicore Computing*, November 2013, Halmstad, Sweden. [50]
 - Martti Forsell, Erik Hansson, Christoph Kessler, Jari-Matti Mäkelä, Ville Leppänen: Hardware and Software Support for NUMA Computing on Configurable Emulated Shared Memory Architectures *Proc. 15th Workshop on Advances on Parallel and Distributed Processing Symposium (APDCM 2013), IPDPS-2013 Workshop proceedings* [44]
 - Erik Hansson, Erik Alnervik, Christoph Kessler, and Martti Forsell: A Quantitative Comparison of PRAM based Emulated Shared Memory Architectures to Current Multicore CPUs and GPUs. *Proc. PASA-2014, 11th Workshop on Parallel Systems and Algorithms*, Lübeck, Germany, Feb. 2014. [51]
 - Martti Forsell, Erik Hansson, Christoph Kessler, Jari-Matti Mäkelä and Ville Leppänen: NUMA Computing with Hardware and Software Co-Support on Configurable Emulated Shared Memory Architectures. *International Journal of Networking and Computing (IJNC)* ISSN: 2185-2847 (print), Special Issue on APDCM 2013. [43]
 - Erik Hansson, Christoph Kessler: Optimized selection of runtime mode for the reconfigurable PRAM-NUMA architecture REPLICA using machine learning *Proc. of 7th Int. Workshop on Multi-Core Computing Systems (MuCoCoS-2014) in conjunction with Euro-Par 2014* Porto, Portugal, August 2014. To appear. [53]
 - Erik Hansson, Christoph Kessler: Global optimization of execution mode selection for the reconfigurable PRAM-NUMA multicore architecture REPLICA Accepted to *6th International Workshop on Parallel and Distributed Algorithms and Applications (PDAA '14) in conjunction with CANDAR 2014*, Shizuoka City, Japan 2014. [52]

The following papers on related topics (not specific to REPLICA) are not covered in this thesis:

- Erik Hansson, Joar Sohl, Christoph Kessler, Dake Liu: Case Study of Efficient Parallel Memory Access Programming for an Embedded Heterogeneous Multicore DSP Architecture. *Proc. MCC-2010 Third Swedish Workshop on Multicore Computing*, Gothenburg, Sweden, Nov. 2010. [54]

- Erik Hansson, Joar Sohl, Christoph Kessler, Dake Liu: Case Study of Efficient Parallel Memory Access Programming for the Embedded Heterogeneous Multicore DSP Architecture ePUMA. *Proc. Int. Workshop on Multi-Core Computing Systems (MuCoCoS-2011)*, June 2011, Seoul, Korea. IEEE CS Press. [54]
- Amin Shafiee Sarvestani, Erik Hansson, and Christoph Kessler Extensible Pattern Recognition in DSP Programs using Cetus *Cetus Users and Compiler Infrastructure Workshop, in conjunction with 20. International Conference on Parallel Architectures and Compilation Techniques (PACT11)*, October 2011, Galveston, TX, USA. [86]
- Amin Shafiee Sarvestani, Erik Hansson, Christoph Kessler: Towards Domain Specific Automatic Parallelization. *Proc. MCC'12 Fifth Swedish Workshop on Multicore Computing*, Nov. 2012, Stockholm. [5]
- Amin Shafiee Sarvestani, Erik Hansson, Christoph Kessler: Extensible Recognition of Algorithmic Patterns in DSP Programs for Automatic Parallelization. *International Journal of Parallel Programming*, Nov. 2012. [85]

1.4 Outline

This rest of thesis is organized as follows:

In Chapter 2 we introduce the REPLICA architecture and some related architectures. We also introduce the REPLICA languages and compiler tool-chain. In Chapter 3 we show how the REPLICA compiler can generate target code for different versions of the REPLICA processor and how it utilizes the chained functional units. In Chapter 4 REPLICAs PRAM mode is quantitatively evaluated using benchmarks and compared to other related PRAM-based architectures as well as state-of-the-art commercial off-the-shelf available CPUs and GPUs. In Chapter 5 the NUMA mode is described together with an evaluation. Chapter 6 contains a case study for generic stencil operations. We use state-of-the-art machine learning methods to derive cost functions in order to automatically select at runtime which mode to execute in. It also contains an extension of how map a sequence of kernel invocations to PRAM and NUMA modes in an optimized way. In Chapter 7 we give some general conclusions and future work. Finally, in 8 we give a list of acronyms.

Chapter 2

REPLICA architecture

This chapter of the thesis is based on [72, 64, 60, 51, 44] and gives an introduction the REPLICA architecture.

2.1 Introduction and motivation

Since the race for faster processor clock speed and instruction level parallelism has slowed down, industry has again turned to parallel computing. In the past, the development of computational performance has resulted in the emergence of several styles of architectures [88] – *clusters*, *vector computing*, *message passing systems*, *symmetric multi-processing* (SMP), and *non-uniform memory access* (NUMA), to name a few. More recent attempts have focused on NUMA style computation with SIMD (*single instruction, multiple data*) vector instructions and general purpose GPU computing, all of which support synchronous execution at best at local level, and add to the complexity of parallel programming.

The reason for this change is that hardware manufacturers try to keep up with the demand of more computation power and at the same time consume less energy. As a consequence, speed-up of legacy, single-threaded computer programs does not come for free any more but requires rewriting to leverage many cores. Even worse is that, even where providing a shared memory abstraction, these new architectures mainly follow NUMA and SMP designs that lack features that could ease parallel programming, such as strong memory consistency or deterministic execution. Among the architectural approaches for parallel and multicore computing making use of memory – let it be distributed either on-chip or among a number of chips – there are very few that support simple programmability and performance scalability with respect to sequential computing [76]. This is because most approaches define asynchronous execution of computational threads and do not support efficient hiding of the new kind of memory reference/ inter-communication latency – delay caused by routing the references/messages

to their targets and if necessary replies back. This prevents programmers from using simple parallel algorithms with a clear notion of the state of computation and therefore makes programming complex and many parallel algorithms weakly scalable.

A notable exception is so called *emulated shared memory* (ESM) machine [82, 69, 29], which provides a synchronous programming model mimicking the *parallel random access machine* (PRAM) model of computation [45] and hides the latency of the distributed memory system by employing the high-throughput computing scheme, i.e. executing other threads while a thread is referring to the memory. The theoretical PRAM model also provides a synchronous and predictable model of programming on a homogenous hardware with an explicit form of parallelism.

To ease the burden for both application programmers and compiler engineers, some architecture projects [78, 40, 95] are working towards supporting more powerful, deterministic parallel programming models such as the PRAM model [45, 58]. The PRAM model is often considered as only a theoretical programming model, but already in the 1990s it has been realized in hardware, albeit not on a single chip, e.g. the SB-PRAM [78, 58].

PRAMs are instruction-level synchronous MIMD parallel architectures with shared memory and are traditionally programmed in the SPMD execution style using PRAM languages such as Fork [58, 63], e [33] etc. that map the naturally available tight synchronization of the underlying hardware to the expression and statement level, allowing to reduce explicit synchronization in the code while maintaining deterministic parallel execution.¹ While following the SPMD style across the whole machine gives full control over the assignment of computation to execution resources, it becomes cumbersome for more irregular application scenarios that require adaptive resource allocation strategies.

In the past, one of the main issues with PRAM was the lack of efficient implementations. This has now radically changed. In order to have full performance, an ESM machine needs to have applications with enough *thread-level parallelism* (TLP). This poses a problem related to functionalities with low TLP and a compatibility problem with existing sequential and *non-uniform memory access* (NUMA) programs. Earlier work have proposed a *configurable emulated shared memory* (CESM) machine to solve this problem by allowing a number of threads to be bunched together to mimic native NUMA operation so that the overhead introduced by plain ESM architectures can be eliminated [38, 41]. The original PRAM-NUMA model of computation [39, 41] defines separate networks and memory systems for the different modes of the machine, which is impractical from the point of view of writing unified programs making use of both modes. In order to simplify programming, we have proposed unifying the modes by embedding the NUMA system into the PRAM system so that there is no need for dedicated

¹The strict memory consistency model of PRAMs is the strongest possible shared memory consistency model, it is even stronger than sequential consistency.

NUMA network nor dedicated NUMA memories [40]. That work, however, left hardware details open and did not provide a clean way to integrate the usage of these two modes into a single program with intercommunication support.

A family of PRAM-NUMA style *configurable emulated shared memory* (CESM) architectures [38, 44] are being developed at VTT Oulu (Finland), and a platform which bears the same name was chosen as the hardware target for our parallel language Replica. The REPLICa architecture will be realized in hardware and is the successor of the *Total Eclipse* architecture [40]. The REPLICa architecture is a chip multiprocessor with configurable emulated shared memory (CESM) architecture [42]. As stated before, its computation model is based on the PRAM (Parallel Random Access Machine) model [58], but it also has support for execution in NUMA mode. The PRAM model gives a simple deterministic synchronous and predictable model of programming, where parallelism is homogeneous and explicit. The REPLICa core architecture is a massively hardware-multithreaded configurable very long instruction word (VLIW) processor with chained functional units so that the result of one VLIW sub-instruction can be used as input to another sub-instruction in the same (PRAM) execution step. It has powerful 2D mesh on-chip combining network providing uniform access to on-chip distributed shared memory.

The architecture has support for parallel multi-(prefix)operations on the hardware level [37]. It enables the user to access the same memory location from a multitude of parallel threads. There is no need for explicit locking as it would be in conventional parallel programming. By running a high number of threads in parallel, latency at memory accesses is effectively hidden by the architecture. Shared (physically distributed across on-chip memory modules) memory is in PRAM mode accessed in a UMA (Uniform Memory Access) fashion as if it was local memory.

At the moment a high-level programming language for REPLICa is under development and should be transformed to the REPLICa baseline language using a source-to-source transformer which is currently under development using ANTLRv3 and written in Scala [42].

The baseline language is based on C with some built-in variables to support parallelism, at the moment these are for example `_thread_id`, `_number_of_threads` and `_private_space_start`.

Programs written in the baseline language can be compiled using Clang to LLVM IR and then compiled to REPLICa target code and tested and evaluated on the REPLICa simulator. One key feature of the compiler is the parametrization of the scheduling algorithm. In an earlier version of the compiler [3] there was only support for one basic architecture configuration. Another key feature is that the REPLICa compiler supports both PRAM and NUMA mode. The main difference from a compiler perspective is that NUMA and PRAM have different pipelines. NUMA also gives different shared memory access paradigms, which are supported by the compiler.

2.2 Related work

The mainstream computing outside high performance server/mainstream domain is basically split into GPU computing and x86 style multi-core processors. Other types of processors such as the heterogenous Cell [57] platform have been proposed, but none of the approaches have yet gained widespread adoption. There is also the well known XMT [95] parallel architecture, which provides a different kind of SPMD (single program, multiple data) execution model for parallel programs. Another PRAM implementation was the SP-PRAM also based on emulated shared memory [58].

While the GPU architectures support running large number of simultaneous threads, the full utilization of the computational potential depends on algorithms that try to minimize the divergence of control flow. On x86 the situation is a bit different. The computational model and design tradeoffs focus on maximizing the performance of highly independent tasks, but make inter-thread communication and lightweight thread creation and synchronization expensive. The XMT architecture on the other hand abandons the PRAM style step-wise synchronization of REPLICA for a more throughput oriented SPMD model and on-demand thread creation. REPLICA's approach is to provide a more general purpose platform with MIMD (multiple instruction, multiple data) execution semantics.

SB-PRAM

SB-PRAM was a research project to implement the PRAM model in hardware during the nineties [58]. The last hardware prototype [78] had 64 processors with 2048 hardware threads in total and 4GB of shared memory, following the *multiple instruction multiple data* (MIMD) paradigm and had uniform memory access time for all processors. The processors are connected to the memory modules using a butterfly network. The processors have an extended Berkeley-RISC instruction set. The routers in the butterfly network support concurrent reads and writes and also parallel prefix operations.

The processor was clocked at 8MHz which is a modest frequency in today's perspective. Still it would be possible to have a higher clock frequency if a higher number of threads was used to hide memory latency, this compensates for the relative gap between memory and processor speed [28].

XMT

XMT (eXplicit Multi Threading) architecture has some similarities with both REPLICA and SB-PRAM since it is also inspired from the PRAM model and will be implemented in hardware, at the moment there exists a FPGA based prototype [92]. The XMT can be seen as master-slave architecture, with a larger core called *master thread control unit* (MTCU) and several smaller *thread control units* (TCUs). The MTCU runs in serial and

can spawn threads that will be executed on the TCUs [96]. The TCUs are divided into different physical clusters; from our perspective we see a cluster as processor or core that has several hardware threads. Each cluster has a shared set of functional units and a single port to shared memory. This implies that each thread can only run at speed c/t where c is the global clock frequency and t the number of threads per cluster.

The clusters are connected via a high-bandwidth low-latency network that is globally asynchronous locally synchronous [92]. Inter-thread communication between clusters is asynchronous and therefore does not realize the PRAM model.

The programmer can spawn more software threads than there are hardware threads, leading to both an elegant programming style and less overhead than if the application software had to explicitly make use of the fixed number of hardware threads. Unfortunately, this happens with the cost of synchronicity, making programs of using frequent synchronization expensive to execute.

Other parallel programming frameworks

In the field of parallel programming languages there are hundreds of research languages focusing on a single or multiple parallel programming topics and de facto standard languages used in the industry such as OpenMP [8]. There are also parallel programming libraries and frameworks such as the MPI [49] message passing interface. GPU architectures have yet another class of programming frameworks such as CUDA. While MPI is useful in clusters where the communication latencies grow higher, it can be seen as a totally different kind of emulation layer on top of the shared memory abstraction in REPLICA. OpenMP has traditionally focused on parallelization of loops, but now also offers a task abstraction.

While these frameworks all have their merits, the language design for a platform involves many kinds of tradeoffs which also involve the cost of implementation and ease of use. With REPLICA we wanted to combine the ease of implementation, simplicity, and suitability for PRAM like architectures of earlier approaches (e [34] and Fork [63]) with advanced, generic parallel constructs and patterns from research languages. In addition, the architecture has several unique features (e.g. all variations of multi-operations, thread bunching, NUMA mode) that do not have equivalent higher level abstractions in these frameworks.

A library oriented approach such as MPI allows larger portability and support due to minimal amount of changes to the compiler tools. However, the loose language integration in OpenMP and MPI are not optimal for further extending the parallel code analysis done in the compiler. In REPLICA we combine the library oriented design with tight language integration by splitting the language into a high level language with new constructs, which gets compiled to a lower level intermediate language. The intermediate language is as close to C as possible, with a minimum amount of extensions to

support the special features of the architecture. The last argument against existing frameworks is that the loosely integrated parallel features seem to encourage writing sequential programs as a default. There exists a sequential subset of the language that must be explicitly augmented with parallel constructs. The REPLICA language considers the parallel semantics as an integral part of the language.

2.3 Architecture

From the programming language’s point of view, the main concepts where the language semantics intertwine with the architectural model are the memory abstraction, execution model, and architecture specific special operations. Next, we discuss each of these aspects in more detail.

2.3.1 Memory abstraction

REPLICA’s CESM model extends the PRAM model and its unit cost memory access with a hybrid model comprising PRAM style synchronous distributed shared memory and interconnected NUMA style local memory hierarchies for each core. In the PRAM mode, each memory fetch costs exactly one logical time unit to execute, which is different from the actual cost to execute the cycle on hardware – amortized unit cost requires lots of parallel memory accesses interleaved with computation to hide the associated latencies. In the NUMA mode, the access cost of processor local memories resembles that of contemporary NUMA machines. In the PRAM mode, parallel memory accesses to a same memory location can take advantage of so called step caches by combining the requests originating from the same core. This capability is further extended with the so called multi-operations.

The distinction between the two memory types is done by dividing the address space into a part consisting of synchronous shared memory and several processor-local regions that provide fast local memory access to the threads residing on those processor cores. The synchronous shared memory is only available in the PRAM mode, while the processor local memory is accessible in both modes.

In CESM, each processor executes a fixed number of hardware threads in an interleaved fashion. The inherent parallel slackness of this approach is used to hide the communication latency arising from the routing in the processor interconnection network and the access latency of physical memory modules. While this approach helps with managing the cost of memory accesses when the processor utilization is high, the relative performance decreases as the utilization goes down and empty threads need to be executed nevertheless. Multiple techniques for mitigating low utilization are described in the next section.

2.3.2 Execution model

The default execution model in the REPLICAS CSM architecture is rather different from contemporary architectures; all threads execute instructions in globally synchronous steps using a synchronization wave technique [1, 82]. This means that conceptually any two synchronous execution paths consisting of the same amount of instructions will be executed at the same pace by any two threads. Another difference is the memory consistency model, which guarantees that all pending memory requests will be issued before the next execution step takes place. Together these properties eliminate a number of difficult memory coherence issues that need to be manually taken care of in user code in the current mainstream architectures.

The CSM architecture also has a second NUMA style execution mode. The execution mode can be dynamically switched for a group of thread at run time. While the PRAM mode is aimed for executing synchronous algorithms with a large number of threads, the NUMA mode favors locality aware parallel NUMA code. The REPLICAS execution pipeline is organized as a chain of functional units which allows various optimization in all modes. Even legacy sequential code can be further accelerated with a technique known as bunching, with which the chained pipeline is filled in VLIW (*very long instruction word*) fashion from a single thread to improve instruction level parallelism. The PRAM mode code can also be compiled to take advantage of the chaining. We call this low-level optimization as *virtual instruction level parallelism* (VILP).

2.3.3 Specific special operations

The final feature of the REPLICAS CSM architecture is the concept of multi-operations along with concurrent memory access optimizations – the traditional PRAM approach to data parallelism is to execute the same simple operation separately on each processor. In contemporary architectures SIMD instructions are used to perform multiple operations within a single thread and instruction. The purpose of multi-operations is different; multiple threads on a number of processors can perform an aggregate computational task, such as scan or sum, in constant time inside the processor, because the same scratchpad / step cache storage for the results can be reused and reads and writes combined when accessing the shared memory. The use of multi-operations is mainly as a performance optimization for aggregate operations, but they can also be used to implement synchronization mechanisms.

2.3.4 Assembly programming model

As mentioned before the REPLICA architecture is a VLIW processor, where each instruction word contains several sub-instructions where each sub-instruction maps to a physical functional unit. In traditional VLIW processors all sub-instructions in an instruction word have to be independent from each other. If the processor supports many sub-instructions in the same instruction word the program has to have enough *instruction level parallelism* (ILP) to be able to fully utilize the processor. Hence, having too long instruction words with too many functional units does not make any sense.

In contrast to traditional VLIW architectures, REPLICAS sub-instructions can (in PRAM mode) be dependent due to chained functional units. We refer to a single chained VLIW word as a *line*. Sub-instructions on the same line are to be issued at the same time. In Figure 2.1 we show a schematic overview of the REPLICA pipeline. Different types of sub-instructions are executed in different functional units; in REPLICA we distinguish between the following sub-instruction types [3]:

- Memory unit sub-instructions: Load, store and multi-prefix instructions.
- ALU sub-instructions: add, subtract etc.
- Compare unit: Compare sub-instructions which set status register flags.
- Sequencer: branch instructions, jump etc.
- Operand: sub-instruction for loading constants, labels etc. into an operand slot
- Writeback: sub-instruction for copying register contents.

In Table 2.1 different configurations are shown. The simplest configuration, T5, has one ALU before the memory unit and then a compare unit and a sequencer after each other.

When programming on assembly level for the architecture, one has to distinguish between two types of register storage for intermediate results:

- *general purpose registers* R1 to R30 can store values persistently;
- *output buffers* O0 to O_x², A0 to A_x, M0 to M_x are transient and only valid inside the line.

Both types, however, can be used in the same way:

²Here, x denotes the number of functional units of that type minus 1. For example the T7 configuration has 3 ALUs, and thus the ALU output buffers are named A0 to A2. The O_i buffers are for result of OPerand instr., the A_i of ALU instr. and the M_i of memory unit instr., respectively

Name	operands	pre ALU	memory	post ALU	compare unit	sequencer unit
T5	2	1	1	0	1	1
T7	3	2	1	1	1	1
T11	7	5	1	2	1	1
T14	7	5	4	2	1	1

Table 2.1: Available standard configurations. The ordering (left to right) of the functional unit types also resembles their ordering in the processor pipeline.

Listing 2.1: Using registers and output buffers as operands

```
ADD0 R1, R2
ADD1 O2, A0
```

As shown in Listing 2.1, every functional unit is bound to one output buffer, denoted by the number following the instruction mnemonic. Only one dedicated functional unit type, the write-back stage, can write values to the register file.

By using the output buffers inside the super-instruction different operations can be chained and the output of one functional unit is fed as an input to another one. This saves a lot of cycles compared to a way where every intermediate result first would have to be written back to the register file. A slight drawback is that results can only be used from the left to the right. Memory operations are an exception. They cannot be chained as latency hiding would not work any longer.

Listing 2.2: REPLICA assembly example [3]

```
OP0 16      ADD0 O0, R1  LD0 A0      WB1 A0      WB2 M0
OP0 2       MUL0 O0, R2  ST0 A0, R0
```

The example in Listing 2.2 computes at line 1 an address by adding 16 to R1 using the first ALU, ADD0, the result will be available in A0 and used to load a word. The result (available in M0) is copied to R2 using the writeback instruction WB2, at the same time A0 is copied to R1. In the second line (next execution step) R2 is multiplied with 2; the intermediate result is in A0 and will be stored at the address contained in R0. How optimizations can be done is discussed in Chapter 3.

2.4 REPLICA language

Our main emphasis in the design of the REPLICA language was on utilizing the underlying unique new hardware in an efficient way while also providing the user with an easy to use high level interface for expressing a wide multitude of parallel applications.

Although there are existing languages for the PRAM style platforms such as Fork [63] and e [34], there is room for improvement. The previous

Figure 2.1: Schematic overview of the processor pipeline. Adapted from [50, 51].

work fails to provide a streamlined interface for the hardware supported multi-operations. In addition, a general purpose parallel application not only operates on the level of individual threads, but has to support the concept of tasks and their scheduling. On top of that, a general purpose high performance language has to deal with algorithms and data structures in a generic way to improve reuse and to boost productivity. Typically such flexibility incurs a performance overhead e.g. in the form of virtual function in object oriented design.

In the REPLICA language the access to multi-operations is provided by standard library calls, but the operations are also planned to be used in generic collection operations, if possible. The support for e language style thread group concept is further extended to support larger scale tasks. The generic algorithms and data structures are enabled via a parametric compile-time type system. We use a parametric system inspired more by functional languages than C++ since the

2.4.1 Compilation model

REPLICA takes a pragmatic approach to language design. ANSI C [59] is a very general purpose, portable, "high level assembler" language with a wide tool support and familiarity. Unless a radically different execution model is needed, C with inline assembler extensions can be used as an intermediate language for targeting the architecture. We use this approach for our

baseline language. This language retains full compatibility with C, if such property is needed when interfacing with other code.

In addition, we define a new higher level language based on C that takes care of the thread and task level parallelism and architecture specific special operations. Not all architecture specific optimizations are applicable on this level, which forces us to use a custom backend for each target architecture configuration. However, previous C compiler frameworks can be reused if the high level language targets the intermediate baseline language. Our high level language also attempts to tackle the other requirements with respect to general purpose parallelism. As one of the language's goals was ease of programming, the high level REPLICA places some restrictions on the semantics of C in order to meet with the goals of simplicity and safety and for opening new potential for automatic optimizations. The tools and the language specification will be released as open source to further increase adoption and help people experiment with the language.

Figure 2.2: REPLICA tool chain. Picture adapted from [72].

REPLICA's approach is to build around an existing C compiler (i) a source-to-source compiler for high level language features, (ii) convey platform specific meta data through an optimizing C back-end, and to use (iii) a platform and configuration specific machine code translator for the REPLICA architecture. The compilation process is depicted in Figure 2.2. The source-to-source frontend is implemented with ANTLRv3 and Scala, LLVM is used as a C backend, and a configurable VILP optimizer is integrated in the backend. The REPLICA language is first compiled into a baseline language, which is ordinary C enriched with parallel programming primitives in the form of macros and annotations for storage specifiers.

2.4.2 Language constructs

The core set of features in REPLICA is based on ANSI C. Covering the whole set of language's features is here beyond the scope, but a short summary of the proposed semantic and syntactic changes is given. As the language is still few iterations away from a stable specification, the focus on this here is more on design principles and not on delivering a finalized precise specification of the features. The treatment of language features is expressed via syntactic transformations to the baseline language.

Storage type modifiers

REPLICA uses a set of conventions for specifying the location of data in the address space based on the context. By default, all variable declarations allocate thread local private storage, which is equal to explicitly tagging these variables with the `private` modifier. This is in stark contrast to ANSI C, where the language makes no assumptions about threads other than that all data is shared. When running plain C on CESM, the variables, function arguments and calls would all be replicated for all threads in a thread group.

In order to share data between threads, the language provides an explicit attribute `shared` for tagging this data. At the programming language level this makes the variable name refer to the same location on all threads of the group. The shared data adheres to the synchronous CRCW (concurrent read, concurrent write) PRAM semantics, which means that unlike contemporary architectures, REPLICA can guarantee a deterministic memory model even in this case. However, low level synchronization constructs, such as basic barriers for the threads in a group, are still provided by the language to prevent erroneous use of shared data in case of independent concurrent thread groups or processes.

The advantage of providing type modifiers for storage is that the language can impose rules for cases where a reference to private data is used in a context where a global reference is expected or when the programmer tries to pass private data between threads. These type rules mainly prohibit uses which are physically impossible due to the way private and shared memory areas are mapped to the address space. The modifiers are checked in the source to source frontend compiler, the necessary changes to storage allocation are postponed to the backend code generation.

Thread identity intrinsics

The execution of threads is organized around the concept of groups. In the beginning of the program execution, all threads form a single global group, and each thread gets assigned a unique global thread id between 0 and *number of threads* - 1. When the execution of threads diverges, subgroups can be formed. Among the threads in a subgroup with size $\# = n$, the threads are numbered from 0 to $n - 1$, respectively. On the other hand, the

same threads retain the assigned unique global thread id throughout the program execution.

Thread identity intrinsics form the very basic technique for implementing data parallel PRAM algorithms. The id values can also be used to diverge the control flow with normal control structures. The responsibility of updating the thread identity values is left both to the programmer and the compiler. The compiler handles subgroups only if the programmer uses the versions of control structures with automatic subgroup management (see the next Section 2.4.2).

The intrinsics for thread id and thread count can be accessed/inspected via special machine intrinsics symbols `$` and `#`, respectively. The same functions are available in textual form: `thread_id` and `number_of_threads`. The global versions of id and count are available via `$$` and `##` and functions with the `global_` prefix. As the functions are all intrinsics, they are rather easily converted to respective system calls.

Parallel control structures

The basic control flow inside a thread group is managed in REPLICA with the standard control constructs (for, if, while, do, switch). The language provides both synchronous and asynchronous versions of all constructs (postfix underscore `_` forces synchronization). To automatically manage the splitting/joining of thread groups in case the control flow diverges, versions of control structures annotated with a prefix underscore (e.g. `_if`, `_for`) are available. Both behaviours can be combined like in the e language. Changes in the control structures involves surrounding the inner scope with `split()` and `join()` macros and the macro `sync()` after synchronized blocks. These macros are provided by the baseline language.

In addition, REPLICA raises the level of abstraction by providing higher level parallel control structures that do not expose the arithmetics with thread ids to the programmer. One of the common constructs is `split`. It allows splitting the current group of threads into n subgroups, with $\left\lfloor \frac{\#}{n} \right\rfloor$ threads in each. The same construct can also create subgroups, each with a certain n_i amount of threads, and assign the remaining threads to a last subgroup. It is the programmer's responsibility to verify that each subgroup is assigned enough threads.

Specific special operations

The target architecture for REPLICA provides several kinds of special hardware modes and operations not typically available in languages. The NUMA mode and bunching of sequential code have been addressed with a `sequential` block which switches the threads related to the thread group into the sequential NUMA mode. By default, the operating mode for all threads is the PRAM mode.

The block mechanism allows the system to execute both NUMA and PRAM thread groups concurrently in an organized manner. The optimization related to thread bunching has not yet been addressed in the preliminary version of the language. Also, the NUMA style management of private memory would use a generic library function to perform the allocation instead of a built-in support.

The architecture also comes with a set of multi-operations. Since each of the multi-operations imposes some non-trivial restrictions on the scope it is applicable in, we found it problematic to provide automatic code transformations that can utilize these special instructions. The operations are exposed via type safe wrappers for the hardware intrinsics.

Generic programming support

The problem with lower level languages like C [59], e [35], and Fork [63] is the lack of support for generic programming. Non-parametric functions and data structures means that the level of expressivity cannot be improved by abstracting the reuse of code. However, a very basic problem with simple C++ style generics common in some parallel "skeleton" frameworks is the lack of disciplined control over the template parameters. The latest C++11 standard was expected to contain an extension for templates in the form of template concepts. Unfortunately the feature did not make it to the standard.

Another way to approach the issue of organizing the data and code is via paradigms such as object oriented programming (OOP) and functional programming. The problem with OOP is the inherent performance penalty related to possible virtual function calls and the focus on the mutable state, which can make parallelization more difficult.

Our decision with REPLICA was to extend the basic procedural execution model of C with a form of generics more common in the field of functional programming. We introduce generics that is syntactically similar to C++ and Java. The feature however lacks support for object orientation, but is extended with compile time type class [94] constraints that can be used to give a hierarchical structure to the related code and data.

Parallel runtime library

As the scope is to introduce only a preliminary language solution for our new architecture, the language design description will not go too much into details about the library design. Some of the runtime library functions such as the synchronization and thread group and identity handling code have already been introduced in the previous sections. Also the intrinsics code for the architecture specific special modes and operations requires runtime support due to two staged compilation process using a standard C compiler.

This is however not the final scope of the runtime library. We also have plans to extend the library with a multitude of standard parallel algorithms,

generic parallel data structures, and with a runtime system handling the execution and scheduling of task level parallelism [60]. It is also natural that the library provides support for functions similar to those in the standard C library. For instance (parallel) dynamic memory allocation is a requirement for all but the most trivial example algorithms.

2.4.3 Kernel example

As a proof of concept we have implemented some test examples in the high level REPLICA and the baseline language, compiled them using our LLVM based back-end and run them on our cycle accurate architecture simulator. The expressivity of the REPLICA high level language and the baseline language is equivalent since the high level language is translated to the lower level. However, the high level compiler performs static verifications and boilerplate transformations that are left as the programmer’s responsibility when working with the baseline language.

Figure 2.3 shows a parallel image threshold filter code written in the high level REPLICA language. The code makes use of thread identity functions, multioperations via a library function and storage attributes (e.g. `shared sum`). The code computes the average value for the sum of RGB components for each pixel in the image and uses this threshold value to recolor the pixels either as black or white.

The high level code is then compiled to the baseline language (Figure 2.4). The resulting code has been hand edited – without introducing any semantic changes – to shorten the unnecessarily long module identifier prefix (`$root$threshold$`) from symbols in this trivial example, to convert the multi-operation call into an inlined inline assembler code and to restructure the whitespace to better fit for reading.

The baseline language example makes use of the built in variables `_number_of_threads` and `_thread_id`. Notice that `sum_` and image variables are shared since their variable names end with `_`. The inline assembly makes use of the multi-operation for parallel add, where all threads that execute it contribute to calculating the sum; the result is stored in `sum_`. Of course we can wrap the code inside a function, but we show it here for clarity. The `sync()` function makes all the threads wait on each other; it is needed since in some cases the statements in the loop body are not executed by all threads.

# Proc.	# Threads/Proc.	Cycles Unoptimized	Cycles Optimized	VILP Speedup	Par. Speedup
4	512	6340558	5416960	1.1705	1.00
16	512	2007552	1689088	1.1854	3.21
64	512	926720	757248	1.2238	7.51

Table 2.2: Executed cycles on different processor configurations running threshold filter. The rightmost columns show (VILP) speedup for unoptimized vs. optimized code and parallel speedup, respectively

# Proc.	# Threads/Procc	Cycles Unoptimized	Cycles Optimized	VILP Speedup	Par. Speedup
4	512	61443584	53352445	1.1517	1.00
16	512	15972348	13837308	1.1543	3.86
64	512	4589564	3945468	1.1632	13.52

Table 2.3: Executed cycles on different processor configurations running blur filter. The rightmost columns show (VILP) speedup for unoptimized vs. optimized code and parallel speedup, respectively.

Our back-end compiler compiles baseline programs and optimizes the target code, which can run on the REPLICA simulator. The blur filter evaluated in Table 2.3 is implemented in a similar way as the threshold filter. Both blur and threshold filter use a 640×480 pixels image, and the blur filter is implemented as a cross stencil with 3 pixels on each side of the center pixel. The tests are run on different configurations, with and without preliminary optimizations turned on. The code optimization (instruction selection and scheduling), which is work under progress, and based on VILP gives currently up to 1.22x speedup compared to unoptimized instruction streams, as seen in Table 2.2. When comparing the scalability of the code, increasing the amount of processors 16 times from 4 to 64 gives a 13.52x speedup.

2.5 Summary

In this chapter we presented the main design principles of REPLICA, a new parallel systems programming language for a synchronous shared memory multi-core architecture bearing the same name. REPLICA aims to provide scalable parallel abstractions both on the low level via parallel control constructs and compiler intrinsics mapped to raw machine instructions (e.g. multi-operations), and on the high level with generic data parallel collections and algorithms that make use of the lower overhead PRAM algorithmic theory and a high level asynchronous task parallel concurrency framework.

The work on the high level language features and library support is still in its infancy. It is our hope that features such as the "synchronicity inference" can automatically detect cases where the machine generated boilerplate can be further reduced to decrease unnecessary overhead, when using higher level abstractions. The utilization of multi-operations in the context of higher level aggregate operations is also challenging due to unique limitations of each multi-operation. We also have a preliminary idea for a runtime task parallelism library which can be used to introduce multitasking and possibly better dynamic allocation of computational resources on PRAM. The system will give flexibility by using dynamic scheduling and late resource binding while preserving the PRAM execution properties and efficient data parallel algorithms within each task.

The example kernel function and its translated baseline version show the

programming style of the REPLICA language and its multistage compilation. Even though it is work under progress, our back-end can optimize the generated code by statically scheduling the instructions to utilize the chained pipeline and we thereby can gain up to 1.22x constant speedup for the code. We also show in our simulation that increasing the core count from 4 to 64 (a 16x increase) improves performance by a factor of 13.52, which equals to near linear scalability with respect to the amount of threads and cores available in this test. As these tests are only run on an architecture simulator, it is too early to draw conclusions of performance in real world tasks. However, the cost and scalability properties closely mimic those of the actual planned hardware.

Our preliminary implementation's main goal is to provide a tool chain for a whole program compilation to statically linked native binary code, which will work on top of a very minimal batch queue like operating system and on our simulator. An improved language version could later introduce a dynamically linkable object file format, with support for higher level features (e.g. generics).

Code generation for the REPLICA architecture can be optimized to take even more advantage of the hardware-provided virtual instruction level parallelism, see Chapter 3.

```

module threshold;
import multiops;
int SIZE = 640 * 480;

struct rgb { int r,g,b; }

struct image {
    int[54] meta;
    rgb[SIZE] data;
}

shared image inImage, outImage;
shared uint sum;
uint sum2;

int main() {
    uint i;
    if_ ($ == 0)
        for (i = 0; i < 54; i += 1)
            outImage.meta[i] = inImage.meta[i];

    for_ (i = $; i < SIZE; i += #) {
        sum = inImage.data[i].r +
              inImage.data[i].g +
              inImage.data[i].b;
        m_add(&sum, sum2);
    }
    sum2 = sum / SIZE;

    for (i = $; i < SIZE; i += #) {
        uint psum = inImage.data[i].r +
                    inImage.data[i].g +
                    inImage.data[i].b;
        outImage.data[i].r =
        outImage.data[i].g =
        outImage.data[i].b = sum2 > psum ? 0 : 255;
    }
}

```

Figure 2.3: High level REPLICA code example of threshold filter. Adapted from [72].

```

#include "replica.h"
#define SIZE 307200

// omitted struct definitions

struct image inImage_;
struct image outImage_;
unsigned int sum_ = 0;
unsigned int sum2 = 0;

int main() {
    unsigned int i;
    if (_thread_id == 0)
        for (i = 0; i < 54; i++)
            outImage_.meta[i] = inImage_.meta[i];
    sync();
    for (i = _thread_id; i < SIZE;
         i += _number_of_threads) {
        sum2 = inImage_.data[i].r +
               inImage_.data[i].g +
               inImage_.data[i].b;
        asm("MADDO %0,%1:: \"r\"(sum2),\"r\"(&sum_));
    }
    sync();
    sum2 = sum_ / SIZE;
    for (i = _thread_id; i < SIZE;
         i += _number_of_threads){
        unsigned int psum =
            inImage_.data[i].r +
            inImage_.data[i].g +
            inImage_.data[i].b;
        outImage_.data[i].r =
        outImage_.data[i].g =
        outImage_.data[i].b = sum2 > psum ? 0 : 255;
    }
}

```

Figure 2.4: REPLICA baseline code example of threshold filter. Adapted from [72].

Chapter 3

Instruction scheduling

This chapter is based on [64].

3.1 Introduction

The focus in this chapter is to show how the compiler actually utilizes instruction level parallelism for different configurations of the REPLICA architecture in which we have different combinations of chained functional units.

As a proof of concept we have written, in the baseline language, some test programs such as thread parallel blur and threshold filter as well as sequential programs such as multiply & move, discrete wavelet and inverse discrete cosine transformation. We compiled them for different configurations using our parametrized back-end. The compiled programs are run on the simulator. They all show speed-ups from instruction level parallelism.

3.2 Dependency Graph

In both this version and the earlier version [3] of the compiler, the support of VLIW is implemented by matching a set of pre-defined combinations of sub-instructions which we call super-instructions. Of course the instruction scheduling and register allocation will not be optimal. We try to solve this problem by splitting each super-instruction into its respective sub-instructions. User-written inline assembly code is also split up. After splitting, any dependency graph between the instructions previously built is no longer valid. We therefore need to build a new one, suited to the requirements given by both the register compression (see section 3.3) and the ILP scheduling pass (see section 3.4).

A new graph is constructed by adding one instruction after another in the existing order and determining dependencies to all previous instructions.

Two classes are used to store and structure the acquired information, as shown in Listing 3.1.

Listing 3.1: Definition of the MIEdge and MINode class

```

class MINode {
public:
    MachineBasicBlock::iterator I;
    list<MIEdge> edges;
    unsigned predecessors;
    bool scheduled;
    // [...]
};

class MIEdge {
public:
    int latency;
    MINode* to;
    // [...]
};

```

The first one, **class** MINode, is used to encapsulate the MachineBasicBlock::iterator to one instruction. Furthermore it stores, among other attributes used later on, a list of edges to other nodes which depend on this instruction. These edges are formed by instances of the **class** MIEdge.

The attribute latency¹ of an edge represents whether the two instructions:

- must stay within the same super-instruction, e.g. OP0 42 $\xrightarrow{=0}$ LD0 O0: The operand O0 is only valid within a line.
- the depending one *has* to be scheduled in a later super-instruction, e.g. TRAP R0 $\xrightarrow{>0}$ ADD0 R1,R2: We cannot mess with the order before and after a trap.
- or it does not matter ($\xrightarrow{\geq 0}$).

The information what was previously part of one super-instruction is only important to such a degree as to determine how long transient registers are alive. Between the instruction defining the output buffer and the one using it will be an edge with latency constraint = 0.

In order to simplify the scheduling pass later on, additional edges are inserted between instructions that are not directly depending on each other.

¹Latency k of a data dependence edge (i,j) is usually defined as if $i \xrightarrow{\geq k} j$ then $t_j \geq t_i + k$ must hold for a correct schedule.

Figure 3.1: Additional edges

This is only necessary for instructions that have to go into the same super-instruction. As shown in Figure 3.1(b), an additional edge is inserted between nodes A and B . This is required because node A would otherwise only force C into the same super-instruction and leave out B as the edges only provide a forward reference.

Compared to the dependency graph construction in the previous version of the compiler, the methods determining the dependencies have been optimized to report fewer unnecessary conflicts which results directly in fewer edges between the nodes. This in turn enables the scheduling algorithm later on to change the order of the instructions more freely and give a better result.

3.3 Register Compression

The number of slots provided for each functional unit type is limited per super-instruction. Therefore it is quite self explanatory that replacing instructions with shorter versions that do the same job. renders the resulting code more compact. As a result scheduling can pack the same code in fewer super-instructions and thereby speed up its execution. Up to now, the following substitutions are implemented:

- If the constant number 0 is required, use the register R0 (which is there for that very purpose) instead of defining an operand with OP 0 and then using the operand. Hence one operand slot is saved and can be used for something else. This is applicable whenever a variable is initialized or reset to zero.
- If an addition at which one operand is a constant zero (R0) is performed, the operation can be omitted and instead of the result the non-zero operand can be used. The same holds true for subtractions where the subtrahend is constant zero. This saves one slot in the ALU.
- With the last substitution we might end up with a situation where a register is written back to itself. Such a constellation can of course be removed entirely.
- If a constant zero (R0) is written back to a general purpose register, all upcoming usages of that register can be replaced with R0 up to the next redefinition of that register.

To enforce the compiler to use register compression a flag, `-enable-replica-register-compression`, can be used. While at this point the compiler and simulator still operate under the assumption that we have unlimited write-back slots, this will not hold true in future versions. At that point saving WB operations will pay off.

3.4 ILP scheduling

As REPLICa is a VLIW architecture, the compiler should identify operations that can be executed in parallel when there are no dependencies and resource conflicts between them, and put them in one super-instruction. Additionally the REPLICa architecture offers the possibility to chain instructions. That means that we can use the output of one functional unit inside a super-instruction as an input to the next one. Thereby we don't have to wait until the result is written back to the register file.

When the LLVM intermediate representation is lowered to the REPLICa instruction set, the created VLIW super-instructions do represent the correct semantics, but they utilize the available functionality quite poorly. An addition operation, for example, will only make use of one ALU and one write-back slot in a super-instruction. All other slots are idling at that time. To make better usage of both the available ILP and the possibility to chain instructions, we have provided an optimization pass that reschedules the instructions, at basic block level, into new super-instructions.

Another important advantage is that this also helps us to generate code for different REPLICa configurations, i.e. providing different amounts of available slots per functional unit type.

Before we start with a description of the scheduling algorithm, some naming conventions that we will use should be introduced. The scheduler is written in a way that we support an arbitrary number of slots per functional unit type. The available slots per functional unit type as a whole will be referred to as a *bucket* (operand bucket, pre-memory-ALU bucket, memory bucket, ...). A *chain* of instructions is a list of instructions that must be scheduled in the same super-instruction. A chain is characterized by edges with latency constraint = 0. We call an instruction *ready* if all its predecessors in the dependency graph are scheduled. We call an instruction *schedulable* when it is ready and all the instructions in the chain are ready as well as there is enough space in the respective buckets to schedule the whole chain. An instruction is *emitted* when it is finally moved from its respective bucket to the output. Instructions that are scheduled but not emitted yet have a special influence during the scheduling.

3.4.1 Algorithm

As Figure 3.2 shows we start the scheduling by creating a dependency graph as described in Section 3.2. By implementation, this directly provides us

Figure 3.2: Flowchart of the scheduling algorithm. Adapted from [64].

with a list of instructions that are ready. These instructions are put into the ready set.

The main part of the work is split between two major steps "find next schedulable instruction" and "schedule instruction", and a third step "emit instructions" which are explained in the following.

Find next schedulable instruction

In this first step, we try to determine the next instruction that is schedulable. Naturally, this instruction has to be picked from the ready set. Hence, `isSchedulable` is called on one element (instruction) after another until we find a result. The challenge here is, that we not only have to consider the instruction itself, but the entire chain; so there has to be enough space in

the respective buckets for all the instructions involved.

A goal was to implement this in an efficient way such that we can start with an arbitrary instruction but if find at some point halfway through the chain that there is not enough space, we don't have to do a complicated roll-back of the involved data structures.

Our greedy approach does this by starting with a resource vector of type **struct** RemainingSlots that holds the amount of slots left in each bucket. This resource vector is passed by reference to recursive isSchedulable calls.

Every instance of isSchedulable now does the following steps:

- It is checked if the instruction has to be stalled because of an instruction that is already scheduled but not yet emitted. This can happen because of sequencing or memory instructions.
- The resource vector is checked if there is enough space in the respective bucket. If so, the corresponding counter is decremented by one.
- isSchedulable is called recursively on all depending instructions with latency = 0.

If one of these tests fails the entire attempt on the chain fails and the next ready instruction is tested.

One of the important features of the REPLICA architecture is the chaining of instructions. Therefore isSchedulable checks if the current instruction is depending on a write-back (WB) that is scheduled but not emitted yet. If this is the case and all other prerequisites are met, we save a note that we bypass the write-back and use the output of the functional unit directly that the write-back would otherwise have to save first. Thereby we save one step. If this instruction is the only one that uses the register until it is redefined, the write-back is marked for removal altogether. At the same time we have to be aware of the order inside the super-instruction: E.g. a post-memory-ALU output cannot be used as an operand to a store instruction.

As an example, see the code in Listing 3.2:

```
OP0 1337          WB2 O0
OP0 42           LD0 R2,O0          WB2 M0
```

This will be rearranged to:

Listing 3.2: Skipping a writeback instruction

```
OP0 1337          OP1 42           LD0 O0,O1          WB2 M0
```

Schedule instruction

Now that we determined that an instruction plus all recursively dependent ones are schedulable, the instructions are all put in their respective buckets.

Two types of register modifications now have to be applied. Both can be observed in Listing 3.2.

1. The replacements that were earlier noted because we skip a writeback. LD0 now uses O0 instead of R2.
2. If we have more than one slot in a bucket, we replace the output buffer that is used. OP0 42 was moved from slot 0 to slot 1.

All instructions that are pulled in by a latency = 0 constraint are scheduled as well. This goes fine as it was part of the `isSchedulable` check earlier.

All instructions depending on this one have now one predecessor less. Those that have reached zero predecessors are ready. Hence they are put in the ready set.

Emit instructions

If "get next schedulable instruction" can't find a suitable instruction, there are apparently not enough slots left. In this case the instructions are emitted as a VLIW and the buckets are emptied. This way of picking the next instruction will eventually cover all instructions because the initial packing into superinstructions derived by the LLVM IR sub-tree matching only imposes minimal requirements (i.e. only one slot per functional unit) and the algorithm will terminate.

3.4.2 Parametrization

In order to enforce the compiler to generate target code for a specific configuration we have introduced a set of compiler flags that we will explain briefly. Since the flags are quite many, the compiler is actually run from a script. The first parameter is `-enable-replica-ilp` which tells that the rescheduling should be applied in order to increase instruction level parallelism (ILP); for debugging purposes it can be switched off. The concept of having buckets is mentioned in the previous section, where the idea is to collect instructions that go into the same functional unit. This is intentionally built in a way that there is no restriction as to how many slots there are per bucket. The limitation of the number of slots is only imposed by passing a resource vector from an instance of `isSchedulable` to another. The initial amount (how many slots are there per bucket in a new superinstruction) is not defined in advance but rather given as a command line parameter to the compiler; `-num-ops` tells the number of integer or floating point constants available per super-instruction, `-num-alus` tells how many ALUs are available before the memory unit, `-num-mus` tells number of memory units, finally `-num-alus-post` tells the number of ALUs available after the memory access is done. All these parameters can be shown by calling `llc -help-hidden`.

As explained extensively, rescheduling tries to use the output of one functional unit as an input to another one and thereby bypass write-back operations. Furthermore, the PRAM model implementation by REPLICA requires to hide the latency to memory with different access times. Due to

internal matters, this only succeeds when memory accesses are executed in parallel and not sequentially. Therefore chaining data going to and coming from memory is not possible: `-chained-mus` is therefore set to `false` by default. `-enable-replica-inline-integration` tells if an inline assembly string be split up into separate instructions and be subject to following optimization passes. The option to not split and reschedule inline assembly can be useful for debugging purposes. It is important to notice that the dependencies of inline assembly are actually taken care of. For evaluation purposes we have the parameter `-num-reg-read` and `-num-reg-write` to limit the number of general purpose registers that can be read and written respectively in each super instruction, the default is 32 (number of general purpose registers per thread).

3.5 Evaluation

In order to measure the improvement achieved through the different optimization passes, several test programs were written in REPLICA baseline language. Even though the focus was on instruction level parallelism we have also used some thread parallel programs as benchmark, to see their characteristics for instruction level parallelism. We have benchmark programs *threshold image filtering* and *blur filtering* which are both thread parallel, moreover a single threaded *discrete wavelet transform* (DWT), where our implementation is based on [77], and one simple test *multiply & move* where elements in an array are multiplied and then moved to another array. We also tested an *inverse discrete cosine transform* (IDCT), this computation kernel was extracted from the Mediabench [66] mpeg2 package.

The baseline case with unoptimized code means that no rescheduling is done, which means that unsplit super-instructions are used directly from the mapping of the LLVM IR. For each configuration (T5, T7, T11 and T14) we did two test series for each benchmark program. One puts no constraints on the number of register read and writes, which can be seen in Figure 3.3. In the second test series we limited the number of both read and write ports to the register file to a maximum of four. Figure 3.4 displays a relative comparison with respect to performance between the two cases. It shows the expected behavior: When we increase the number of functional units without also increasing the number of available registers, speed-up will suffer. For the benchmarks multiply & move and image blur with limited register read and write ports, we will still get almost the same speed-up as in the unlimited case, this is because the functional units can be utilized well since the data dependencies allow us to make use of the buffers in the chained functional units.

Figure 3.3: Instruction level speed-up comparison with different configurations, no constraints on number of read and write ports to reg. file. Adapted from [64].

Figure 3.4: Relative slowdown when limiting the number of read and write ports to four. Adapted from [64].

3.6 Comparison to Previous Work

There has been a lot of work done in instruction scheduling, see e.g. [98, 27, 90, 2].

It is well-known that the time-optimal instruction scheduling problem for basic blocks is for most target architectures NP-complete [61]. While smaller scheduling problems can, today, be solved to optimality using integer linear programming and similar techniques, the general case is typically solved by heuristic algorithms.

One classical heuristic solution of the scheduling problem is Graham’s greedy algorithm for scheduling tasks [47], known as “list scheduling”. It maps tasks to a set of processors, where the tasks can be dependent on each other. The tasks are kept in a list ordered by their dependencies and other priorities so to assign a task to an idle processor it goes through the list and picks the next ready task. If no runnable task is found the processor will idle. Another scheduling algorithm is critical path scheduling [61].

Finlayson et al. [26] show how compiler support can help reducing power consumption without adversely affecting performance when introducing internal registers, that are read and written explicitly, instead of pipeline register. These internal registers reminds a lot of REPLICAs explicit functional unit result registers.

When it comes to VLIW scheduling for architectures with chained functional units we are only aware of the following two, which we compare in detail.

3.6.1 Original virtual ILP algorithm

Scheduling for a more generalized version of this type of architecture has been proposed by [31]. The algorithm proposed there proceeds in a different way than we do. It is more focused on filling the available memory slots. It sees the next free slot (both memory slots and other ones) and then tries to find a suitable instruction that is independent of all the remaining unscheduled instructions. From our point of view, this has some disadvantages:

- a lot of instructions are inspected which then turn out to have the wrong type.
- it is only taken into account that this instruction depends on other ones, not that other instructions (in this chain) might depend on this one and therefore have to fit into the same super-instruction/ VLIW.

Our algorithm, on the other hand, scans through the set of ready instructions and checks if they can be scheduled. This seems to be more flexible if dependencies in both directions have to be considered.

3.6.2 Earlier ILP scheduler for the REPLICA compiler

Together with the general compiler back-end, Åkesson [3] also implemented an optimization pass that aims at exploiting more ILP. The implemented algorithm is based on [31] but looks for the next instruction (as we do) instead of the next slot. As the necessity to manage instructions in a different stage was recognized, different containers exist for holding instructions:

- those that have to be scheduled immediately (within the same super-instruction)
- those that have to be scheduled whenever there is enough space
- those that have to be scheduled at the earliest in the next super-instruction.

One can see that this corresponds to the different latency constraints defined in the dependency graph. This design, however, turns out to be rather cumbersome when we try to use short-cuts and skip write-backs. In order to do this scheduling, a dependency graph was required as well in [3]. The implementation however took in many cases a too conservative approach (e.g., it ignored the fact that output buffers are only valid inside a super-instruction) which resulted in too many unnecessary dependencies that rendered rescheduling more difficult or even impossible. [3] only support one basic REPLICA configuration.

3.7 Conclusion and future work

We have shown that our implementation of a parametrizable instruction scheduling algorithm for a VLIW processor with chained functional units produces high quality code for different hardware configurations. The high parametrization of the compiler makes it, together with the simulator, useful for evaluating different hardware configurations.

Future work includes the extension of scheduling beyond basic blocks, for example for loops using software pipelining techniques. It can be interesting to evaluate it for the REPLICA architecture since the case of chained functional units is a different one compared to standard VLIW architectures.

Another interesting idea would be to try to formulate and solve the scheduling problem as an integrated code generation problem of instruction selection, scheduling and register allocation together, for instance using *integer linear programming* both at basic block level and beyond. In earlier work Eriksson [24],[25] models integrated code generation for clustered VLIW DSPs using this technique. One example of similarity is that clustered VLIW DSPs have different register files which give constraints on which registers can be used by the different functional units; in our case we have the transient functional registers which are exposed to the programmer

and can only be used from left to right and are only valid during one super-instruction step. Both lead to strong coupling between register allocation and instruction scheduling, for which integrated code generation provides higher code quality, see Eriksson [24].

Chapter 4

Evaluation of REPLICA PRAM mode

This chapter is based on [51, 50].

4.1 Motivation to evaluate PRAM

To keep up with the never decreasing demand of more computation power both researchers and hardware manufacturers have turned to parallel computing [89]. One reason is that higher clock frequencies is a dead end with current technology due to power and heat constraints. For example high-end personal computers had already in 2003 clock frequencies up to 4GHz, today, ten years later they are about the same [92]. Still the current technology evolves so that more transistors, and hence more logic, can be built on a single chip, following Moore's law [92]. This has lead to the multicore trend that we face today – processor chips contain multiple processor cores that execute computational threads in parallel. Another approach used for speeding up sequential programs is to exploit *instruction level parallelism* (ILP) by executing multiple instructions in multiple functional units. Unfortunately, the amount of easily extractable ILP is limited in most applications. As the step towards exploiting massive *thread-level parallelism* (TLP) in applications now must be taken, the trade-off between ILP and TLP must be balanced cost-effectively.

It is claimed [42, 92] that current CPUs are inefficient in certain types of parallel functionalities making use of frequent inter-thread communication/synchronization and that so-called *Emulated Shared Memory* (ESM) architectures would solve these problems. ESMs use multithreading to hide the latency of the shared memory subsystem instead of caches and provide highly cost-efficient synchronization mechanisms. The motivation for shared memory emulation comes from the theoretical *Parallel Random Access Ma-*

chine (PRAM) model [58] that helps a programmer to focus on the essence of parallel computation without having to worry about data locality and various architecture/implementation-dependent properties of the machine, see [58] for more information about ESM. We show that an advanced ESM architecture, REPLICAS, performs very well compared to the Intel CPU and Nvidia GPUs, even though it has not yet been provided with latest performance enhancement techniques such as SIMD instructions. In several cases the REPLICAS architecture only needs to be clocked at a few hundred Megahertz to match them.

Here, we compare preliminarily the performance of three hardware implemented ESM architectures with a state-of-the-art CPU and GPU. The benchmarks are selected to cover different patterns of parallel computation. For the CPU and GPU we used fast library implementations of the benchmark functionalities while most of ESM the implementations were written by us and therefore potentially less optimized, see Table 4.3.

Some related work includes [78] and [17] for SB-PRAM, [9] where XMT is compared to GPUs, and [67] compares GPUs and CPUs. Vuduc et al. in [93] also compares CPUs to GPUs for problems that are irregular, and states that a GPU is in this case roughly comparable to one or two CPUs (including the on-chip cores). [4] investigates CPU programming using OpenCL.

4.2 Hardware Architectures

In order to figure out the performance potential of ESM architectures with respect to current CPUs and GPUs, we selected five different architectures for comparison – REPLICAS (ESM), SB-PRAM (ESM), XMT (asynchronous ESM), Intel Xeon X5660 CPU, and Nvidia Tesla M2050 GPU.

In Section 2 we have already given an introduction to the REPLICAS architecture, XMT and SP-PRAM.

4.2.1 Intel Xeon CPU

Xeon X5660 is a 2.8 GHz commercially off-the-shelf available Intel CPU with 6 dual-threaded cores with SIMD units. It has three levels of cache; 32 kB data cache, 256 kB mid-level cache per core and 12 MB last-level cache shared between the cores [55].

4.2.2 NVidia Tesla GPGPU

Tesla M2050 is a 575 MHz commercially off-the-shelf available NVidia GPU with 14 streaming multiprocessors, with totally 448 CUDA cores and 3 GB memory in total. Each streaming multiprocessor can handle up to 1536 resident threads, i.e., the maximal number of resident threads is $14 \times 1536 = 21504$. GPUs can profit from multithreading to hide memory latency. However, we do not consider it here; instead we refer to [10].

Benchmark	Description	Mem. access	Control flow input dependent	Berkeley dwarfs [7]
prefix sum	Prefix sums of an array.	regular	No	No
matmul	Product of two matrices.	regular	No	Yes
smatvec	Product of a sparse matrix and a vector.	irregular	Yes	Yes
breadth first	Breadth first search from a graph.	irregular	Yes	Yes

Table 4.1: List of benchmarks

Configuration	#Cores	#Thrds	#FUs	Model
REPLICA-4	4	2048	10	ESM
REPLICA-16	16	8192	10	ESM
REPLICA-64	64	32768	10	ESM
SB-PRAM	64	2048	4	ESM
XMT	64	1024	4	AESM
Xeon CPU	6	6		SMP
TESLA GPU	448	21504		Hybrid

Table 4.2: List of architectures tested. FUs stands for Functional Units.

4.3 Evaluation of PRAM mode

We evaluated the performance of the architectures of Section 4.2 by measuring the execution time of four benchmarks representing different types of parallel computing patterns (see Table 4.1) on seven configurations of the architectures (see Table 6.1). We motivate our selection of benchmarks as follows: three of the benchmarks that we use, namely matrix matrix multiply, sparse matrix multiply and breadth first search, belong to the well-known Berkeley dwarfs [7]. Moreover, we have two irregular memory access algorithms; sparse matrix vector multiply and breadth first search, which would suit ESM well. On the other hand, prefix sum and matrix matrix multiply are regular memory access algorithms and therefore suit cache based systems better. The control flows of sparse matrix vector multiply and breadth first search are input dependent, which is not the case for prefix sum and matrix matrix multiply.

Benchmark	Intel Xeon	Nvidia Tesla	XMT	SB-PRAM	REPLICA
prefix sum	-	Thrust::inclusive_scan [75]	From [21]	-	-
matmul	OpenBLAS [99]	CuBLAS [75]	Based on [74]	-	-
smatvec	Sparse Library (SL) [100]	CUSP [11]	Based on [74]	-	-
breadth first search	Rodinia [87]	Rodinia [87]	-	-	-

Table 4.3: Implementations used for the different architectures. - means that we implemented it ourselves for this study.

Name	#Nodes	#Edges
graph4096.txt	4096	24576
graph65536.txt	65536	393216
graph1MW_6.txt	1000000	5999970

Table 4.4: Graphs from the Rodinia benchmark suite [87] used in our BFS benchmarking

REPLICA benchmarks were written in the baseline REPLICA language [72], compiled in our LLVM 3.0 based REPLICA compiler capable of generating and optimizing target code for the different hardware configurations [64] and executed in an in-house developed cycle-accurate simulator modeling processors, interconnect and on-chip memories.

For SB-PRAM we used the Fork95 compiler (fcc version 2.0) and SB-PRAM simulator pramsim 2.0 assuming an ideal memory system.

For XMT we used the cycle-accurate XMT Simulator version 0.82.112 with 1024 TCUs and 64 clusters modeling the processors, interconnect and on-chip caches, and the xmtcc 0.82.0 compiler with O2 optimizations.

All tests for Intel Xeon X5660 were done using Debian squeeze with Linux kernel 2.6.32-5-amd64 (x86_64) and gcc version 4.4.5 with O2 optimizations with hyperthreading switched off. For the matrix matrix multiplication we used the highly optimized parallel library OpenBLAS stable release version 0.2.8 [99].

The tests for Tesla were done on ARCH Linux with Linux kernel 3.10.10-1-ARCH (x86_64) using nvcc Cuda compilation tools, release 5.0, V0.2.1221 [75]. For the prefix sum benchmark we used `Thrust::inclusive_scan` and for the matrix matrix multiplication we used the highly optimized CuBLAS library that comes with CUDA 5.0 Toolkit [75].

In Table 4.3 we show the benchmark implementations we used for Intel Xeon, Nvidia Tesla, XMT, SB-PRAM and REPLICA. Note that we used library implementations for all of the Berkeley Dwarfs listed in Table 4.1 for Intel Xeon and Nvidia Tesla.

For breadth first search we used graphs from the Rodinia benchmark suite 2.4 [87]. For CPU and GPU implementations we used the target specific BFS algorithms provided by the Rodinia benchmark suite 2.4 [87]. Among our benchmarks BFS on graphs are the most irregular problems with respect to memory accesses.

Name	#Cols/Rows	#non-zeros
Internet	124651	207214
Lugn2	109460	492564
ASIC_680ks	682712	2329176
t2em	921632	4590832

Table 4.5: Sparse matrices used from The University of Florida Sparse Matrix Collection [16].

Figure 4.1 shows selected performance results normalized as processed elements per execution time in clock cycles (and normalized to performance of Xeon for sparse matrix vector multiply since the number of elements is not meaningfully defined for sparse matrices), assuming the same clock frequency for CPU and ESMs and 4.8 times lower frequency for GPU¹. From these results we can make the following observations:

The highest performance was obtained from the REPLICAs and XMT architectures except in matrix matrix multiply where the Tesla GPU achieved the best performance if data transportation time to GPU and back to CPU was not taken into account. The second best was the Xeon CPU for smaller matrices and REPLICAs for larger matrices. The lowest performance was provided by Tesla with data copying. The main reason for lower than expected ESM performance in this benchmark is that, while Xeon and Tesla are using highly assembler optimized library algorithms, ESMs use the standard triple-nested loop algorithm and no hand optimization. There exist faster algorithms also for ESM but due to the high amount of work we did not try to include them in this preliminary study. Due to the relatively high degree of parallelism, the GPU, which has the highest number of cores by a large margin, provides the best performance.

As expected, among the 64-core ESMs, REPLICAs and XMT provide better performance than SB-PRAM in all tests. This is because SB-PRAM has a less optimized hardware architecture and less aggressively optimizing compiler than REPLICAs and XMT. XMT outperformed REPLICAs only in cases where the amount of TLP is limited in breadth first search and sparse matrix vector multiply. This is because REPLICAs need more threads to hide the latency of the interconnect and XMT has clever load balancing between the computational tasks. Unfortunately the former means that the silicon implementation of XMT would have lower clock rate although they are assumed to be the same in this study. Xeon CPU with only a fraction of cores of Tesla had higher performance in small prefix sum and breadth-first-search benchmarks if the plain kernel execution time is taken into account and in most benchmarks if the data transportation time to GPU and back is included.

¹This factor comes from the difference in clock speed between the Xeon CPU and Tesla GPU.

Figure 4.1: (a)-(c) Performance as elements per execution time in clock cycles for prefix sum, matrix matrix multiply and breadth first search. (d) Normalized performance for sparse matrix vector multiply (Xeon=1.0) since the number of elements is not well defined for sparse matrices. TESLA-K refers to kernel-only execution while TESLA-T includes also CPU-GPU data transportation. Adapted from [51, 50].

We also considered the necessary clock rate of REPLICA architectures to match the performance of CPU and GPU (see Figure 4.2). From these results we can make the following observations:

The trend for prefix sum is that the larger problem sizes we have, the lower clock frequency is needed by REPLICA to match CPU and GPU, from 3.52GHz down to less than a MHz.

Matrix-matrix multiply needs around 6 GHz for a 16-core REPLICA to match the CPU for the largest problem size, and up to 22GHz to match the GPU if transfer time to the GPU is not considered. The overall lowest frequency needed to match the CPU for matrix-matrix multiply is 1.6GHz.

For sparse matrix-vector multiply, the needed clock frequencies range between 35MHz to 2.3GHz to match CPU and GPU.

For BFS the lowest clock frequency needed was 27MHz to match the Xeon CPU, using a 64-core REPLICA for the largest graph. On the smallest graph the four core REPLICA gave the worst result; 1.3GHz to match Xeon CPU.

In the case of matrix-matrix multiply we only show the comparison

Figure 4.2: Clock frequencies needed for REPLICAs CMPs to match the performance of Xeon CPU and Tesla GPU. Note that Figure a) Prefix Sum has a logarithmic scale on the Y-axis. Adapted from [50, 51].

against the Xeon CPU in Figure 4.2. The reason is that if we consider the data transfer time to the GPU the needed REPLICAs clock frequency is just a few Megahertz, while it is up to hundred GHz without transfer time.

In Figure 4.3 we show the speedups that we get if we clock the REPLICAs at 2.0 GHz; for all benchmarks except matrix-matrix multiply we get a speedup for the different REPLICAs versions. It ranges from 0.86x up to 102x. The lowest speedup (slowdown) among these three benchmarks was for sparse-matrix-vector multiply in the case of the smallest matrix (internet) when we compare the REPLICAs four-core version to the Xeon CPU (which has 6 cores and is clocked higher).

Finally we compared REPLICAs to an otherwise similar system but having an ideal shared memory (so-called PRAM configuration). The slowdown for REPLICAs was less than 1% for prefix sums calculation. This consideration could not be done for the other ESM architectures since XMT does not define the PRAM model and the SB-PRAM results were already PRAM conformant since the simulator did not model the interconnection network².

REPLICAs has a scalable multi-mesh network and has been estimated to be clockable at least at 2 GHz with an old 65 nm technology. Estimating the maximum clock frequency, needed silicon space and resulting power consumption of ESMs with current technology is beyond the scope of this

²Note that [78] reported a similar slowdown of 1-2% measured on the SB-PRAM prototype compared to the pramsim simulated execution times.

Figure 4.3: Speedup for REPLICA CMPs clocked at 2.0 GHz compared to Xeon CPU and Tesla GPU. Note that Figures a), c) and d) have a logarithmic scale on the Y-axis. Adapted from [50, 51].

work.

4.4 Conclusions

We have done a preliminary quantitative comparison of ESM architectures against off-the-shelf CPUs and GPUs by benchmarking. It is commonly known that some types of algorithms, based on their data access scheme, suit some type of architectures better than others. Here we have chosen different benchmarks with large variation to actually avoid biasing some architecture. Some of our tests use both data and benchmarks that are well known, also some of our implementations for the commercially available architectures are written so they make use of state-of-the-art vendor provided libraries.

Assuming that the ESM architectures execute at the same clock rate as the CPU (4.8 times faster than the GPU), the fastest ESM architectures (REPLICA and XMT) perform way better than both CPU and GPU except in matrix multiplication where the latter use asymptotically faster and more optimized algorithms. At the same time also most programs for ESM machines were significantly simpler and shorter than those of CPU and GPU. In Listing 4.1 we give an example of the sparse matrix vector multiply kernel implemented for REPLICA that we used in the benchmark.

Listing 4.1: Sparse matrix vector multiply kernel for REPLICa. Note that variable names that end with `_` are shared. The matrix is compressed using Compressed Sparse Row (CSR) format.

```

counter_ = _number_of_threads;
r=_thread_id;
while(r<ROWS)
{
    sum = 0;
    rowStart = row_ptr_[r];
    rowEnd = row_ptr_[r+1];
    for(c=rowStart; c<rowEnd; ++c)
    {
        sum += val_[c] * x[col_ind_[c]];
    }
    y_[r] = sum;
    aprefix(r,ADD,&counter_,1);
}
_synchronize;

```

The best performance was obtained by REPLICa except with some small-size breadth first searches and sparse matrix vector multiplication by XMT and with all problem sizes by Tesla GPU in matrix matrix multiplication.

This work also reveals some potential weaknesses of the ESM architectures, including the modest architectural implementation of SB-PRAM, limited performance of XMT with large data sets, and problems of REPLICa with low-parallelism and dynamic load balancing cases. The latter ones could be potentially solved by using the NUMA mode [44] combining multiples threads to single a NUMA thread for each processor but we did not test it in this study.

Chapter 5

Exploiting NUMA mode

This chapter is based on [44, 43].

5.1 Introduction

In this chapter we propose a number of hardware and software techniques to support the NUMA computing in CESM architectures in a seamless way. The hardware techniques include three different NUMA shared memory access mechanisms and the software ones provide mechanisms to integrate and optimize NUMA computation into the standard PRAM operation of the CESM. The hardware techniques are evaluated on our REPLICA CESM architecture and compared to an ideal CESM machine making use of the proposed software techniques.

5.2 Configurable ESM architectures

In order to solve the performance and programmability problems of current parallel/multicore machines, the concept of shared memory emulation has been introduced and a number of academic architectures has been proposed [58, 29, 91, 40]. In this section we will discuss the idea of shared memory emulation and adding NUMA support for it to support configurability in those architectures.

Shared memory emulation

The main problems of current architectures are that synchronization (of asynchronous execution) takes hundreds of clock cycles and that cache-based latency hiding scales weakly. To solve these problems one would need a fast/low-overhead synchronization mechanism and scalable latency hiding/-tolerance mechanism. The standard way to solve these problems, known as

shared memory emulation, is to use wave synchronization to provide lock-step synchronous execution and to employ multithreading with a pipelined memory system along with a machine consisting of P processors (each T_p -threaded) connected to M memory modules via a high-bandwidth network of diameter ϕ_{net} .

The idea of wave synchronization is to separate references belonging to consecutive steps of execution during which each thread of the processor executes a single instruction. This is done by sending a set of synchronization references between the steps. Routing of synchronization references happens in an elastic wave-like manner so that synchronicity is maintained [69, 29].

Latency hiding using excessive multithreading is based on overlapping relatively long latency memory references in a pipelined memory system so that when a thread refers to a memory location, other threads are executed e.g. in an interleaved manner until the reply is received [29]. Hot spots and congestion are minimized by using randomized hashing of memory locations [18].

As these two methods are combined the synchronization cost is dropped so that the overhead of lock-step synchronicity in execution time becomes $1/T_p$ and the latency will be hidden with a high probability assuming $T_p \geq 2\phi_{net}$ [82] like the current single core architectures are emulating the model of sequential computation efficiently with a high probability.

5.3 Adding NUMA support

The utilization fraction U_p of a P -processor ESM system with T_p threads per processor as the function of software parallelism T_b (the number of threads in execution at the current moment of time) is

$$U_p = \frac{\min\{T_b, P \times T_p\}}{P \times T_p}$$

We can easily see that if the software parallelism is low U_p gets weak. Most current multicore architectures do not have this problem since they are not using multithreading for latency hiding but coherent caches to exploit access locality in programs (where available) or just try to tolerate natural latency defined by the distance of memory access making memory access non-uniform [68, 81].

In earlier work, [38], the idea of adding the plain NUMA model support for the ESM machine by allowing a set of threads in a processor core to join together as a single bunch that is sharing a single register set and executing instructions consecutively even inside a step instead of executing the same instruction for each thread was introduced. This is done by reorganizing the thread storing mechanism, adding indirection to the register set addressing, merging the ESM pipeline with the standard NUMA pipeline, and providing non-uniform locality-aware access to memory modules. (see

Figure 5.1: Indirecting the thread storage in ESM and merging a T_p -threaded ESM pipeline and standard 4-stage NUMA pipeline to obtain CESM architecture. Adapted from [43, 44].

Figure 5.1) The obtained class of architectures is called configurable emulated shared memory (CESM). We have also introduced the PRAM-NUMA model of computation capturing the details of this solution in more theoretical way [39]. It consists of T processors grouped as P groups of T_p processors, a word-wise accessible global shared memory, P local memory blocks, a metric defining distance between the processor groups and target memory blocks, and distance-aware interconnection network (see Figure 5.2).

5.4 NUMA realization alternatives

The original PRAM-NUMA model of computation [39] defines separate networks and memory systems for the different modes of the machine, which is impractical from the point of view of writing unified programs making use of both modes. In order to simplify hardware implementation and programming, we have proposed unifying the modes by embedding the NUMA system into the PRAM system so that there is no need for a dedicated NUMA network while dedicated NUMA memories are retained as local memory modules [40]. This solution does, however, not define a simple way to unify memory allocation and execution control mechanisms in the different modes of the processor and leaves low-level hardware organization open. Our new idea is to use the PRAM shared memory system to implement storage also for shared NUMA variables while the private PRAM variables are moved to the local memories to reduce the load of the network. Technically there are three obvious ways to implement shared memory load accesses for the NUMA mode – freeze processor, freeze bunch, and load with explicit receive. In the following we describe the main principles of these alternatives.

Figure 5.2: The PRAM-NUMA model of computation. (P =processor, L =local memory, T_p =number of processors per group, T =total number of threads, P =number of processor groups.). Adapted from [43, 44].

- *Freeze processor (FP)* The whole processor core freezes until the reply arrives. As a consequence, the rest of the threads let them be PRAM threads or other NUMA bunches on the same core will also be halted. This resembles adding wait states until the memory reply is received in a standard pipelined processor.
- *Freeze bunch (FB)* The currently executed bunch freezes but the rest of the threads and bunches continue execution. This is implemented by re-executing the tail of the load instruction during the execution slots of the bunch until the reply is received from the memory system. This resembles freezing the current instruction and all the dependent instructions in a standard superscalar processor.
- *Load with explicit receive (LER)* The original load instructions are divided into two new memory instructions – load and receive. The new load instruction sends the shared memory reference on its way to the memory system. The reply is fetched by the receive instruction. If the reply has not arrived at the time of executing the receive, the current bunch freezes until the reply is received, just like in the freeze bunch alternative. With a help of explicit receive it is possible schedule the load and receive separately or even to overlap multiple shared load operations unlike in the other alternatives while the shared store operations are automatically overlapped in all alternatives. A practical upper bound for a number of overlapping references is the number of threads per the NUMA bunch divided by two (assuming both send and receive instructions are executed in the same functional unit) and the upper bound for the latency between a load and corresponding receive is a single step. The former is potentially bounded also by the

number of available registers.

Since these solutions are using the PRAM memory system, the memory wait logic is taking care that the replies are received within the current step of execution. In the case of contention in the network, it can happen that the thread slot initiating the load is trying to leave the memory waiting stages causing the whole processor core to freeze until the reply is received. The synchronization wave is ultimately guaranteeing that all the references made during a step of execution have completed before the references of the next step are applied.

5.5 Programming considerations

Earlier work proposed a parallel application development scheme for ESM machines consisting of a strong model of computation, a C-like TLP programming language *e* [34] or *REPLICA* [71], ILP-TLP optimization algorithm, and application development flow [36]. The scheme allows programmers to write applications with a help of supporting theory of parallel algorithms [56, 58]. Orchestration of threads is arranged in standard PRAM-style by providing thread identifier, number of threads, and synchronous and asynchronous variants of C-style constructs. Since the NUMA mode of the CESM machine does not execute threads synchronously nor provide access distance independent latency hiding like the PRAM model, the scheme does not directly support NUMA programming of CESM machines. In the following we describe how NUMA mode programming can be closely unified to the ESM programming scheme, give some programming examples, explain what kind of support the programming scheme requires, as well as consider compilation and optimization issues for PRAM-NUMA.

Orchestrating parallelism in the NUMA mode

The CESM machine boots up in the PRAM mode, supports switching groups of threads to the NUMA mode and back to PRAM mode, and allows for multiple simultaneous PRAM and NUMA executions. This suggests that the control of execution mode could be implemented as standard blocks at the programming language level. For this we propose using two control constructs that switch the execution to NUMA mode and back. The first construct is

numa(*s*)

that bunches all the threads of current thread group in all participating processor into NUMA bunches and executes statement *s*. After that the mode is switched back to PRAM. Sequential portions of computation can be supported with a construct

sequential(*s*)

that bunches all the threads of current thread group in the participating processor having the lowest Id into a single NUMA bunch and executes statement *s* while the other threads are waiting. This mechanism would just allow switching to NUMA mode and back to PRAM mode but prevents more complex schemes including nesting modes (e.g. switching to NUMA mode, splitting to subgroups to execute them in the PRAM mode). Since there can be multiple PRAM thread groups executing in parallel, it would be possible to set up multiple NUMA bunches that run in parallel with the remaining PRAM groups if any.

In order to provide similar programming interfaces for the both modes, we propose using the PRAM thread orchestration mechanisms also in the NUMA mode where applicable. Controlling individual bunches could e.g. be done with the same *thread identifier* and *number of threads* variables and synchronization could happen with the same barrier construct. Since the cost of synchronization is high in the NUMA mode, we do provide only the standard asynchronous control constructs for NUMA and leave synchronous ones limited to the PRAM mode only.

In the original ESM development scheme passing data between the different parts of the program happens with standard programming language mechanisms, e.g. global variables, stack frames and pointers. To support easy data exchange between the modes we propose reusing the REPLICA memory model consisting of unified shared memory and private memory making use of local memory modules also for the NUMA mode (see Figure 5.3 for the resulting memory organization map as a portion of code executed in the PRAM mode calls a NUMA mode portion in a *T*-threaded CESM). This means that exchange between NUMA bunches happens via the shared memory system only and exchange between the modes happens via the shared memory and bunch leaders private subspace. Then a programmer can directly use the shared variables defined in the PRAM mode but without the notion of locality and refer also to the single set of private variables that belong to the bunch leader. Switching back to the PRAM mode passes the modifications done to shared variables and bunch leader's private variables. All the variables declared inside the NUMA portion are be accessible in the PRAM mode before or after the portion.

Programming example

Consider the computational problem of determining the prefix sums of an array of *N* integers. Let us denote that array with symbol *source*. The standard (sub-optimal) way to compute the prefix sums in parallel is the known as the recursive doubling logarithmic algorithm, where each element of the array gets added by the element 2^i positions left on iteration *i*, $i = 0 \dots \log(N) - 1$. Since the number of processors *P* in the NUMA mode is

Figure 5.3: The memory organization of a T -threaded CESM machine making use of the NUMA mode. Adapted from [43, 44].

often way smaller than N , one needs to compute each iteration by an N/P -iteration inner loop. The algorithm assumes also that data is accessed truly parallel in each outer iteration meaning that a temporary array *temp* has to be used to store intermediate results during inner loop execution and the obtained data must be copied back to the original array. Figure 5.4 shows the implementation of this algorithm in the e language.

The data is initialized in the PRAM mode and the logarithmic prefix computation is done inside the *numa(s)* construct. The execution time of this algorithm is $O(N \log(N)/P)$.

```

#define size N
int source[size]; // Allocate data array from the shared memory
int temp[size]; // Allocate space for a shared temporary array
int main()
{
    int i;
    int j;
    source[_thread_id] = _thread_id; // Initialize source_
                                     // in parallel in the PRAM mode
    numa(
        for (i=1; i<size; i<=<=1)
        {
            for (j=_thread_id; j<size; j+=_number_of_threads)
                if (j-i>=0) temp[j]=source[j] + source[j-i];
            synchronize;
            for (j=_thread_id; j<size; j+=_number_of_threads)
                source[j]=temp[j];
            synchronize;
        }
        synchronize;
    );
}

```

Figure 5.4: Non-optimized NUMA prefix computation in the e language in which symbol “_” at the end of the variable name declares it shared. Adapted from [43, 44].

5.6 Supporting the NUMA mode

The above NUMA programming scheme requires implementation of the numa construct and barrier synchronization routine on a top of the ESM programming scheme.

In order to switch from the PRAM mode to the NUMA mode, the current state of the PRAM thread group is stored into block-local variables and restored from these variables as the execution of NUMA block ends. There is also need to allocate room for a synchronization variable from the shared stack and compute the new *_thread_id* and *_number_of_threads* variables reflecting the bunch id and number of bunches in the NUMA block. Just before entering to the statement *s* the PRAM group is switched to the NUMA mode. After the execution of the statement, the bunches are deformed and the state of the PRAM group is restored from the local variables. The code for the numa construct switching CESM to the NUMA mode, executing the statement *s* and switching back to the PRAM mode is shown in Figure 5.5.

```

#define numa( s ) {
    int _old_thread_id = _thread_id;
    int _old_number_of_threads = _number_of_threads;
    int _old_group_id = _group_id;
    int _old_shared_stack = _shared_stack;
    int _processor_id = _thread_id/_threads_per_processor;
    shared_stack -= 4;
    _group_id = _shared_stack;
    _update_region_numa;
    join_marker;
    {
        s
    }
    split_instruction;
    write_back(32,_private_space_start);
    _thread_id = _old_thread_id;
    _number_of_threads = _old_number_of_threads;
    _group_id = _old_group_id;
    _shared_stack = _old_shared_stack;
}

```

Figure 5.5: Implementation of the **numa** construct. The *_update_region_numa* routine computes the new values for *_thread_id* and *_number_of_threads* variables and switches execution to the NUMA mode. The *split_instruction* routine switches execution back to the PRAM mode. Adapted from [43, 44].

Synchronizing the bunches in the NUMA mode is done with a NUMA-specific library routine *_RTL_SYNCHRONIZE_NUMA* that first checks if the previous synchronization is still going on and waits for that if necessary. Then the synchronization variable is decremented by one for each arriving bunch. To manage the situation in which two or more bunches do this simultaneously we use REPLICA processor's fast multiprefix operations that work partially also in the NUMA mode. The arrived bunches wait until the synchronization variable reaches zero meaning that all the bunches have arrived and continue after that. In order to manage asynchrony of bunches in exiting the routine and reinitializing the synchronization variable the threads decrement the synchronization variable once more and the last bunch leaving the routine does the initialization. The REPLICA assembler code for NUMA-specific barrier synchronization is shown in Figure 5.6.

```

_RTL_SYNCHRONIZE_NUMA
    ADD0    R29,C0    OP0    -4    WB1    A0                ; Save registers
    ADD0    R29,C0    OP0    -8    ST0    R2,R1    WB1    A0
    ADD0    C0,R32    OP0    __group_id ST0    R3,R1    WB1    A0
    LD0     R1        WB1    M0                ; R1 <- pointer to shared __group_id

L10_RTL20
    LD0     R1        WB2    M0                ; R2 <- The synchronization variable __group_id
    SLE0    R2,C0    OP0    0                ; If the synchronization variable " 0 then the previous barrier is still goin on
    NOP0
    NOP0
    BNEZ    C0,R1    OP1    L10_RTL20        ; Wait until it is reinitialized
    MSUB0   C0,R1    OP0    1                ; Decrement the synchronization variable of the group

L20_RTL20
    LD0     R1        WB2    M0                ; R2 <- The synchronization variable __group_id
    SGT0    R2,C0    OP0    0                ; Test if all threads of the group have arrived
    NOP0
    NOP0
    BNEZ    C0,R1    OP1    L20_RTL20        ; If not, continue waiting
    MPSUB0  C0,R1    OP0    1    WB2    M0    ; Decrement the synch variable, R2 <- previous value
    ADD0    C0,R32    OP0    __number_of_threads    WB3    A0    ; R3 <- number of threads
    LD0     R3        WB3    M0
    ADD0    R2,R3    WB2    A0                ; R2 <- number of threads + previous value of the synchronization variable
    SNE0    R2,C0    OP0    1                ; If the sum ↑ 1 then do not reinitialize the synchronization variable
    NOP0
    NOP0
    BNEZ    C0,R1    OP1    L30_RTL20        ; Reinitialize the synchronization var
    ST0     R3,R1

L30_RTL20
    ADD0    R29,C0    OP0    -4    WB1    A0                ; Restore registers
    ADD0    R29,C0    OP0    -8    LD0    R1        WB1    A0    WB2    M0
    NOP0
    NOP0
    LD0     R1        WB3    M0    JMP    R31                ; Return

```

Figure 5.6: The NUMA-specific barrier synchronization routine in REPLICAS assembler. Adapted from [43, 44].

5.6.1 Compiler support

We support the REPLICAS hardware with a tailored compiler tool chain. The code generation part is based on the LLVM compiler framework [65]. Earlier versions of our REPLICAS compiler only supported to generate code for PRAM mode [72, 64]. For the previous architectures, [38, 41], the so called e-compiler was developed and used [35, 32]. To a large extent it can be used for the current REPLICAS architecture as well, and we use it for some of the evaluation benchmarks in Section 5.7, since we try to support both compilers under a transition period.

Here we present the first native REPLICAS compiler version that supports both PRAM and NUMA compilation. The earlier PRAM compiler, [64], can generate code for different configurations of the REPLICAS processor, e.g. for different numbers of functional units (ALUs, MUs etc.) placed in a chain etc. but has no NUMA support.

The first step in the code generation phase, both for PRAM and NUMA mode, is to generate code for a minimal configuration. In the PRAM case it is then translated and optimized for larger configurations. For NUMA mode we still use this initial minimal configuration as a starting point, but instead of optimizing for a more functional units etc., our scheduling algorithm enforces the stricter scheduling constraints such as no chaining of functional units. One example is that a compare instruction and a depending branch

instruction can not be scheduled in the same VLIW instruction. Another is that, after a branch instruction two NOPs are needed. These two examples do not apply in PRAM mode.

To change from PRAM mode to NUMA mode a special instruction, **JOIN**, is used. To go back to PRAM mode **SPLIT** is used. The compiler detects the basic blocks that are encapsulated between the **JOIN** and **SPLIT** instructions and marks them to be NUMA basic blocks. If a **JOIN** or **SPLIT** instruction is found inside a basic block, the block is divided into two new ones.

For the case of *load with explicit receive* (LER), the REPLICA compiler can in NUMA mode translate a load instruction to a new load instruction together with a receive instruction. This desired behavior can be specified with a special compiler flag. It is important to mention that the LER is only for shared memory while private memory is handled with traditional load instructions. To distinguish between loading shared and private memory the compiler recursively analyzes if an address computation uses shared memory or not inside a NUMA block. Of course, there are rare cases where it is not possible to do it statically, then it is up to the programmer to manually adapt the code.

5.6.2 NUMA optimizations

In order to provide decent performance in the NUMA mode there exists a number of optimizations familiar from all NUMA systems, e.g. synchronization minimization, locality maximization, as well as some that are specific to CESM architectures, e.g. scheduling of receive instructions and overlapping two or more shared memory loads.

Since synchronizations (e.g. with the barrier algorithm) take a long time to execute in the NUMA mode compared to a single instruction execution, performance increases if the computation is reorganized so that the number of synchronizations is minimized. A typical way to do this is to divide the data at hand into blocks and process blocks inside processors so that synchronizations are not used. Unfortunately this is not simple and not even always possible. For the prefix example described above there exists a way to do synchronization minimization by blocking although there are a relatively large number of synchronizations in the unoptimized algorithm. For this we divide the shared array to P blocks, compute prefix sums of blocks with a sequential algorithm in P parallel bunches, compute the prefix of the block sums in the bunch 0 with a sequential algorithm, and offset the blocks with the obtained prefixes of the blocks with a sequential algorithm executed in parallel in all bunches (see Figure 5.7).

Locality optimization for the NUMA mode means, in the prefix sums example, dividing the shared data array *source* into P private (local) parts that are processed in the same way. This maximizes the locality since the most data (except computing the prefix sum of the block sums and distributing the results of it) is then processed locally and the only place in which

```

#define size N
volatile int source_[size]; // Allocate from the shared memory
int main()
{ int i, blocksize, start, stop, prev;
  numa(
    blocksize=size/procs;
    start = _thread_id * thrds;
    stop = start + blocksize - 1;
    for (i=start; i<=stop; i++) // Initialize blocks in parallel
      source_[i]=i; // with a sequential algorithm
    synchronize;
    for (i=start+1; i<=stop; i++) // Determine prefixes of
      source_[i]+=source_[i-1]; // blocks in parallel
    synchronize;
    if (_thread_id==0) // Prefix for block sums sequentially
    { for (i=start+thrds+thrds-1; i<procs*thrds; i+=thrds)
      source_[i]+=source_[i-thrds]; }
    synchronize;
    prev = start - 1;
    if ( prev>=0 ) // Add results of prefix sum
    { for (i=start; i<stop; i++) // of block sums to blocks
      source_[i]+=source_[prev]; }
    synchronize;
  );
}

```

Figure 5.7: Synchronization optimized NUMA prefix computation. Adapted from [43, 44].

shared access is needed is computing prefixes of block sums. The resulting program is shown in Figure 5.8.

For suitable algorithms executed in LER-enabled CESM it is possible to schedule the receive instructions so that their distance to corresponding load instructions is maximized or made long enough for partially hiding the latencies of loads. The result is even better if two or more loads are overlapped e.g. with software pipelining. Figure 5.9 shows the main loop of the block benchmark in the unoptimized case, after applying maximization of distance between corresponding load and receive instructions, and after overlapping consecutive shared loads.

For more detailed information on quantitative performance effects given by these optimizations, see the evaluation in the next section.

5.7 Evaluation

In order to evaluate the performance, difficulty of programming, and complexity of the hardware and software techniques proposed in Sections 5.4 and 5.5, we applied them to the REPLICA *chip multiprocessor* (CMP) framework being developed at VTT.

Performance

We measured the execution time of 6 simple benchmarks (see Table 5.1) in 12 configurations of REPLICA making use of the techniques, the standard ESM (the PRAM mode of REPLICA), CESM (REPLICA making use of the fastest available mode) and corresponding PRAM assuming idealized memory system (see Table 5.2). Note that the benchmarks *block*, *prefix* and *rand* are variable sized (problem size= T_{total}) while *barrier*, *numa* and *edge* are fixed sized.

	Symbol	FP- P	FB- P	LER- P	ESM- P	CESM- P	PRAM- P
#processors	P_{total}	P	P	P	P	P	P
#threads	T_{total}	P	P	P	PT_p	PT_p	PT_p
#registers/thread	R_t	R	R	R	R	R	R
#FUs	F_{total}	3	3	3	F	F	F
FU organization	F_{org}	parallel	parallel	parallel	chained	chained	chained
Memory scheme	M_{scheme}	NUMA	NUMA	NUMA	PRAM	PRAM- NUMA	ideal PRAM
Network diameter	ϕ_{net}	$2P^{0.5}$	$2P^{0.5}$	$2P^{0.5}$	$2P^{0.5}$	$2P^{0.5}$	1

Table 5.1: Configurations used in the evaluation (P=number of processors 4 ... 16 in the evaluation).

Benchmark	Description
block	Move a block of T_{total} integers in memory.
barrier	Synchronize the threads in the current group of threads (dependent of the mode).
numa	Switch the current thread group from the PRAM mode to NUMA mode and back.
edge	Detect edges of an RGB image of 640x30 pixels. For locality optimization divide the shared array evenly to P blocks, move them to local memory for processing and copy the results back to the original array.
prefix	Calculate the prefix of T_{total} integers.
rand	Calculate the sequence of T_{total} random numbers using the linear congruential technique.

Table 5.2: Benchmarks

The benchmarks were written in the e language, compiled with the e compiler *ec* [32] applying options *-ilp*, *-o2*, synchronization minimization and simulated on our CMP tool *IPSMSim* [30]. To determine the effect local

memory versus shared memory the NUMA mode tests were done with and without locality optimization maximizing the locality of memory references by using local memory where possible.

The results of measurements are shown in Figures 5.10 – 5.11. From the results we can make the following observations:

- The LER alternative is the fastest shared memory NUMA technique by a small margin while FP is by far the slowest. This is because freezing the whole processor unbalances the timing of the synchronization wave leading to relatively long delays.
- As expected the PRAM mode is often much faster than the NUMA mode but in the *barrier* and strictly sequential *rand* benchmarks all NUMA executions are much faster than the PRAM mode. This is because the ESM can not execute sequential code efficiently and because the number of threads in the PRAM mode is much higher than in the NUMA mode.
- CESM can benefit from the NUMA mode but the cost of switching the machine to the NUMA mode for fast computation and back is substantial ruling fine-grained NUMA exploitation impractical, see e.g. results for *numa*.
- Locality optimizations speed up execution. If there is enough computation per element, like in the *edge* benchmark, this applies even if data is originally in the shared memory. In that case data needs to be divided into local blocks, processed locally and copied back to the shared memory.
- The LER alternative provides the programmer with an option to overlap also load operations speeding up operation significantly compared to non-overlapped operation. This opens up interesting optimization possibilities for programmers and compilers. More accurate evaluation of this is however left to our future work.

In order to figure out the potential performance of the LER-specific software pipelining optimizations, we applied them to the block benchmark. We measured the execution time of the shared memory block in the baseline configuration (BASE), where no optimization is done but the receive instruction is placed just after each shared load instruction, after maximizing the distance between load and receive instructions but not overlapping two or more loads (MAX-DIST), after overlapping each shared memory load with the next one and then maximizing the distance between load and corresponding receive (OVERLAP-1). For comparison purposes we measured the execution time of local memory versions of the block benchmark and the baseline version assuming ideal shared memory. The programs were compiled with the REPLICA compiler and the software pipelining optimizations

(MAX-DIST and OVERLAP-1) were done by hand. The results are shown in Figure 5.12.

We can do the following observations from these results:

- For this simple but very widely used block copy functionality, maximizing the distance between load and corresponding receive increases the performance by 46%-60% and overlapping consecutive loads increases the performance by 89%-124%.
- Overlapping the consecutive loads drops the execution times very close to that of the localized algorithm and ideal shared memory emphasizing the potential of this optimization.

The cost of applying MAX-DIST and OVERLAP-1 techniques was two and four extra registers, respectively. This indicates that overlapping a high number of shared memory loads is not possible without special hardware support.

5.7.1 Code size and programmability

In order to roughly characterize the difficulty of programming, we determined the size of code of all benchmarks. The results are shown in Figure 5.13. We can make the following observations:

- The size of the code is higher for the NUMA execution than it is for the PRAM execution. The difference is biggest in applications making use of frequent exchange of data, e.g. *prefix*, generating a lot of synchronizations in asynchronous NUMA execution. The synchronization optimization increases this difference due to application of the blocking technique.
- Optimizing the code with locality optimization increases the code size depending on the benchmark and its dependencies.

5.8 Conclusions

We have proposed a number of hardware and software techniques to support NUMA computing in CESM architectures in a seamless way. The hardware techniques include different NUMA shared memory access mechanisms and the software ones provide a way to integrate NUMA computation into the standard PRAM operation of CESM. According to the evaluation making use of our REPLICA CMP framework, the proposed solutions can be used to provide relatively unified programming scheme for the CESM architecture making use of both the PRAM and NUMA modes. As expected the PRAM mode is faster in most cases but there are a few clear exceptions, e.g. strictly sequential code in which NUMA performs better. Since the number

of threads in the NUMA mode is much smaller than in the PRAM mode the barrier synchronizations in the NUMA mode are much faster. This does however not mean that the NUMA mode would be typically better for algorithms requiring tightly synchronous execution since the implied wave synchronization of the PRAM mode eliminates the most barriers and multi-threading provides latency hiding that is not available in the NUMA mode. The code size for the NUMA mode is higher than that for the PRAM mode. The difference is most substantial for benchmarks making use of frequent exchange of data between the computational threads. This points out that while NUMA mode can solve some performance problems, this happens with the cost of programmability. According to our preliminary tests it appeared also that the hardware overhead of the FP, FB, and LER techniques is very small.

Our future work includes extending the current REPLICA language and compiler tool-chain with all NUMA-related constructs, investigating possible combined PRAM-NUMA algorithms and optimization possibilities for NUMA mode execution such as loop unrolling/software pipelining and load instruction overlapping/scheduling.

```

#define size N
volatile int final_[size]; // Allocate from the shared memory
volatile int sums_[procs]; // Allocate from the shared memory
int main()
{ int i, blocksize, start, stop, prev;
  numa(
    int source[thrds]; // Allocate from the local memory
    int offset;
    blocksize=size/procs; // Divide into blocks
    start = _thread_id * thrds;
    stop = start + blocksize - 1;
    for (i=0; i<blocksize; i++) // Initialize blocks in parallel
      source[i]=i+start; // with a sequential algorithm
    synchronize;
    for (i=1; i<blocksize; i++) // Determine prefixes of blocks
      source[i]+=source[i-1]; // with a sequential algorithm
    sums_[_thread_id]=source[blocksize-1];
    synchronize;
    if (_thread_id==0) // Prefix for block
      for (i=1; i<procs; i++) // sums sequentially
        sums_[i]+=sums_[i-1]; // in a single processor
    synchronize;
    if (_thread_id>0)
    { offset=sums_[_thread_id-1];
      for (i=1; i<blocksize; i++) // Add results of prefix
        source[i]+=offset; } // sum of block sums to blocks
    synchronize;
    for (i=0; i<blocksize; i++)
      final_[i+start]=source[i];
  );
}

```

Figure 5.8: Locality and synchronization optimized NUMA prefix computation. Adapted from [43, 44].

<p>NON-OPTIMIZED-VERSION</p> <pre> _BB5_2 OP0 _source_ WB8 00 OP0 2 SHL0 R1,00 WB7 A0 OP0 _target_ WB9 00 ADD0 R8,R7 WB8 A0 ADD0 R2,R1 WB1 A0 ADD0 R9,R7 WB7 A0 LDO R8 WB8 M0 WB8 M0 RECO R8 STO R8,R7 SLT R1,R6 OP0 _BB5_2 BNEZ 00 NOP0 NOP0 </pre> <p>DISTANCE-MAXIMIZED-VERSION</p> <pre> ; Prefix OP0 _source_ WB8 00 OP0 2 SHL0 R1,00 WB7 A0 OP0 _target_ WB9 00 ADD0 R8,R7 WB8 A0 ADD0 R2,R1 WB1 A0 ADD0 R9,R7 WB7 A0 LDO R8 WB10 M0 WB11 R7 ; First load SLT R1,R6 ; Main body _BB5_2 OP0 _source_ WB8 00 OP0 2 SHL0 R1,00 WB7 A0 OP0 _target_ WB9 00 ADD0 R8,R7 WB8 A0 ADD0 R2,R1 WB1 A0 ADD0 R9,R7 WB7 A0 LDO R8 WB10 M0 WB11 R7 ; From the previous iteration WB12 M0 RECO R12 ; From the previous iteration STO R12,R13 WB12 R10 WB13 R11 ; From the previous iteration LDO R8 WB10 M0 WB11 R7 SLT R1,R6 OP0 _BB5_2 BNEZ 00 NOP0 NOP0 ; Postfix WB10 M0 RECO R10 ; Final recive STO R10,R11 ; Final store </pre>	<p>OVERLAPPED-VERSION</p> <pre> ; Prefix 1 OP0 _source_ WB8 00 OP0 2 SHL0 R1,00 WB7 A0 OP0 _target_ WB9 00 ADD0 R8,R7 WB8 A0 ADD0 R2,R1 WB1 A0 ADD0 R9,R7 WB7 A0 LDO R8 WB10 M0 WB11 R7 ; First load odd SLT R1,R6 ; Prefix 2 OP0 _source_ WB8 00 OP0 2 SHL0 R1,00 WB7 A0 OP0 _target_ WB9 00 ADD0 R8,R7 WB8 A0 ADD0 R2,R1 WB1 A0 ADD0 R9,R7 WB7 A0 WB12 R10 WB13 R11 LDO R8 WB10 M0 WB11 R7 ; First load even SLT R1,R6 ; Main body _BB5_2 OP0 _source_ WB8 00 OP0 2 SHL0 R1,00 WB7 A0 OP0 _target_ WB9 00 ADD0 R8,R7 WB8 A0 ADD0 R2,R1 WB1 A0 ADD0 R9,R7 WB7 A0 WB12 M0 RECO R12 ; From the previous iteration STO R12,R13 WB12 R10 WB13 R11 ; From the previous iteration LDO R8 WB10 M0 WB11 R7 SLT R1,R6 OP0 _BB5_2 BNEZ 00 NOP0 NOP0 ; Postfix 1 WB12 M0 RECO R12 ; Final receive odd STO R12,R13 ; Final store odd ; Postfix 2 WB10 M0 RECO R10 ; Final receive even STO R10,R11 ; Final store even </pre>
---	---

Figure 5.9: The main loop of the block benchmark without optimizations, after maximizing distance between receives and corresponding loads, and after overlapping consecutive loads. Adapted from [43, 44].

Figure 5.10: Execution time without NUMA optimizations. Adapted from [43, 44].

Figure 5.11: Execution time with locality and synchronizations optimizations. Adapted from [43, 44].

Figure 5.12: Execution time with LER optimizations compared ideal shared memory and local memory executions. Adapted from [43, 44].

Figure 5.13: Length of the measured code segment of the benchmarks in e source code lines. Adapted from [43, 44].

Chapter 6

Automated mode selection

This chapter is based on our publications [53] and [52].

6.1 Introduction to execution mode selection

In the multicore era we do not only face the problems that parallel programming brings; modern architectures and hardware platforms also expose the advantages and problems of managing heterogeneity. Today's computer systems usually have multicore processor chips and dedicated accelerators such as GPUs. To utilize these systems efficiently, boils down to selecting where and how to run a program. How to achieve high performance for real applications is not straight forward, and predicting performance is even harder since aspects such as data locality and movement has to be considered.

If we already have transferred data to the accelerator, it might be beneficial to continue executing the next sub-task(s) on the accelerator even though it had not been worth switching just because of the sub-task, when data transfer time is considered. This leads to a global optimisation problem.

In this study we use the VLIW massively hardware multithreaded emulated shared memory (ESM) architecture REPLICA. Recall that each core has 512 hardware threads and the processor pipeline is designed so that the high number of threads effectively can hide the latency of accessing the emulated shared memory. Since it realizes the PRAM (parallel random access machine) model [58] it is very convenient to program. To get full performance an ESM needs programs with large enough thread level parallelism (TLP). To solve the problem with low TLP REPLICA can be reconfigured at run time so that the time slot of several hardware threads are bunched together and access on-chip memory modules in NUMA mode such that the PRAM emulation is switched off and the overhead from plain ESM is removed [38, 41]. Switching between PRAM and NUMA mode take only a moderate number of clock cycles as overhead.

For the programmer, NUMA mode means that there are fewer threads and the memory latency becomes "visible" and has to be taken care of manually to utilize the hardware fully. The main reason for having NUMA mode is to be able to run sequential legacy programs and programs with low thread level parallelism faster since they do not suit PRAM mode very well, Section 5. To switch to NUMA mode the programmer can join all the threads on a core at runtime, so each core becomes single threaded, and can execute faster.

It is not always obvious which parallel programs will run faster in NUMA mode, one reason is that hashing of memory addresses is not exposed to the programmer. To tackle this we use state-of-the-art machine learning methods.

A software component represents a particular software functionality which can have one or several implementation variants that are considered equivalent concerning their input and output [15]. Even though different variants produce the same output from the same input their (runtime) behavior might differ a lot, depending both on algorithmic implementations and the hardware executing the actual implementation variant.

We have in Section 5, introduced REPLICAs PRAM-NUMA programming model and given some basic examples and evaluations. We have also earlier done a preliminary evaluation of REPLICA PRAM capabilities see Section 4, where one conclusion was that PRAM mode is very good for irregular memory access and control flow problems in contrast to commercially available state-of-the-art CPUs and GPUs. However, as stated in Section 4 REPLICA PRAM mode was in several cases outperformed by cache based CPUs and GPUs when it comes to regular memory accesses and control computations.

The main goal of this chapter is to define an initial model that predicts when to use NUMA mode and when to use PRAM mode in terms of performance. We also try to tackle the problem of global software composition for REPLICAs two runtime modes in the general setting of multiple regions (computations and switching points). We want to both derive lightweight cost prediction functions and that the optimization problem for global composition should be easy and fast to solve. We show that our cost models can be expressed compactly with a few arithmetic operations, and that our optimization problem can be solved with low runtime overhead using well known shortest path algorithms.

Since PRAM mode already is very fast for irregular problems but possibly suboptimal for regular, see Section 4, we here focus on regular data parallel problems namely, generic stencil computations. In Section 5 we also showed that locality and latency optimizations could be beneficial in NUMA mode, these optimizations suite of course regular problems well.

A secondary goal of this part is to take two popular state-of-the-art machine learning tools, one based on decision trees and one on symbolic regression, to see if they can be used for modeling this kind of performance

Mode	Cores	Threads per core	Pre mem- ory ALU	MU	Post Mem- ory ALU	Compare unit	Chained FU
PRAM	4	512	1	1	0	1	Yes
NUMA	4	1	1	1	0	1	No

Table 6.1: Configurations used in this chapter.

optimization problems of hetrogenous architectures. For both methods we use the same training and evaluation data sets.

As mentioned before, the main motivation for NUMA mode is to be able to run sequential legacy programs and programs with low thread level parallelism faster than in PRAM mode. To switch to NUMA mode REPLICa has a specific assembly instruction `JOIN` that joins all threads in a thread group to a NUMA bunch. To switch back to PRAM mode we use the `SPLIT` instruction. In our C based REPLICa baseline language, see 5 or [43], we have a construct `numa(s)` that switches the processor to NUMA mode, executes the statement `s`, and switches back to PRAM mode. The construct also orchestrates setting up the stack pointers, thread ids etc. and restores them. The overhead of switching to NUMA and back is around 16000 clock cycles [43]. When the processor runs in NUMA mode, no chaining of functional units is possible, the number of functional units is fixed to the ones given in Table 6.1 even though it can have more in PRAM mode. This is taken care of by the compiler, see Section 5.6.1. Programming for NUMA mode is like programming for traditional NUMA multicore processor with global shared memory and private local memory.

We have selected a configuration (number of ALUs and MUs) in PRAM mode that looks most similar to the fixed one in NUMA mode, to be able to highlight the differences that comes from NUMA mode itself and not from having a fat PRAM. See Table 6.1 for the specific processor configurations.

In Section 5 we gave three paradigms for accessing shared memory in NUMA mode. In the following we only use LER:

- Load with explicit receive (LER): After an asynchronous shared load, an explicit receive instruction `RECO` need to be issued; in between, other instructions can execute.

If a `RECO` is issued directly after a load the bunch will freeze until data has arrived just like in the case of "freeze bunch". It is important to note that loading from shared memory with the LER paradigm occupies the memory unit twice, once for the load instruction (`LDO`) and once for the receive (`RECO`). With LER the result of the load will be stored in a register and used as an input to the receive. The used register will be kept alive until the receive is done. If we are short of the registers the compiler will insert spill code to local memory (stack) wich can reduce performance and it might have ben better to "freeze" the bunch instead. For this study we still only

focus on LER, since it gives more opportunities for optimizations.

6.2 Parameterized Benchmark

In section 4 we have shown that PRAM mode suits programs with irregular memory access and control flow very well, while regular problems do not benefit from PRAM mode. In that study we only focused on PRAM mode, still our experience is that in order to be able to make (any) practical use of NUMA the problems needs to be very regular in terms of memory accesses and control flow. We need to reduce loads and stores from shared memory, but also reading and writing to local memory is considered expensive in NUMA mode.

To explore when NUMA mode can be beneficial we introduce a parameterized benchmark that is very regular. It can be considered a stencil operation. Compared to other regular algorithms, such as vector and matrix operations, stencil operations are more generic especially regarding how much computation there is per data element. We apply register pipelining [20] in order to load each element once (recall that REPLICA has no caches), and obtain a typical software pipeline code structure consisting of prologue (filling the pipeline), computational kernel (steady state) and epilogue (draining the pipeline). We have the following parameters:

- N : Problem size (number of array elements to update)
- P : Prologue size
- C : Number of instructions for local computations with no memory access
- LLS : Number of local loads and stores.

P models how many times we run the prologue, e.g. the prefetching of data in shared memory that we have to do before the kernel can start executing. The computational kernel loop run for N iterations, once for each data element. Inside the kernel we do LLS local loads and stores. Our experience shows that in order to be able to get any performance out of NUMA mode we can only access a shared memory data element once, otherwise the performance is ruined. If we need the same data again, we must keep it locally (in local memory but registers are preferred). The C parameter resembles the distance between the load and the receive (RECO instruction), and can be seen as how much local (register based) computation we need to do, to hide the latency of the shared memory access.

Optimizations done for NUMA, such as register pipelining [20] to avoid memory accesses (both shared and local) are in our experience often also useful in PRAM mode. To make a fair comparison between NUMA and PRAM mode we run the exactly same program with the same optimizations (locality, registers etc) for both NUMA and PRAM. The only difference is

Figure 6.1: Distribution of N (size), P (prologue size) and LLS (local loads and stores) parameters in the evaluation set S_E . LLS is shown in grayscale. Adopted from [53].

that we, of course, switch to NUMA and the back-end compiler compiles NUMA code to fit the NUMA pipeline (no chained FU). In NUMA mode we also access shared memory using the LER concept (loads with explicit receive instruction). In PRAM mode we also divide the work over all the available HW threads, e.g. 4×512 . In NUMA mode we have joined all the threads per core, so we only have 4 cores (threads) to divide the work among.

6.3 Machine learning models

Using a random number generator (with lower and upper limits on each parameter) we generated different training set and evaluation sets. The parameter space is well covered, see the distribution of the parameters for the evaluation set S_E in Figure 6.1. We generated programs in REPLICA baseline language (an extended version of C), compiled them and run them on the cycle accurate REPLICA simulator.

Initially we started with two training sets, S_1 and S_2 and their union S_3 , where $|S_3| = 45$ and $|S_1| \approx |S_2|$. S_1 and S_2 are disjoint.

Using S_3 we derive formulas for both NUMA and PRAM execution times using the Eureqa Pro framework [84, 83]. We assume that the execution time, t_n , for NUMA is $t_n = f_n(N, C, P, LLS)$. and t_p for PRAM in a similar way. Since we want to make a predictor for when to switch to NUMA mode it is natural to use the speedup, eg. only switch to NUMA if we get speedup larger than one, e.g. $\frac{t_p}{t_n} > 1$.

For C5.0 [80] we tried S_1 , S_2 and S_3 as training sets, however the results were not good enough so we decided to increase the size of the training data.

The new sets S'_1 , S'_2 and $S'_3 = S'_1 \cup S'_2$ contains the old sets in the following way: $S_1 \subset S'_1$, $S_2 \subset S'_2$ where $S_3 \subset S'_3$, $|S'_3| = 98$ and $|S'_1| \approx |S'_2|$

For evaluation we use the set S_E , it is shown in Table 6.6. It is independent from the training sets and $|S_E| = 20$ ¹. Figure 6.1 depicts S_E distribution of the parameters N (size), P (prologue) and LLS (local loads and stores) where LSS is shown with grayscale. Since C5.0 did not use the C parameter we do not show its distribution in the picture.

6.3.1 Eureqa Pro

A first model using S_3

With S_3 we got the following execution time models.

NUMA: $t_n = a_n * P + b_n * N + c_n * N * C + d_n * LLS^2$, where $a_n = 0$, $b_n = 9.65313889909994$, $c_n = 0.224715696133425$ and $d_n = 23005.0142108128$.

PRAM: $t_p = a_p * P + b_p * N + c_p * N * C + d_p * LLS^2$ where $a_p = 3561.80442755642$, $b_p = 5.91351736227114$, $c_p = 0.232529731521942$ and $d_p = 8662.82648471137$.

As stated earlier we use the estimated speedup larger than one, $\frac{t_p}{t_n} > 1$, as a predictor. The generated expressions for the execution time, are quite similar for PRAM and NUMA, one interesting detail is that the P (prologue) is not used for NUMA and that $c_n \approx c_p$.

An updated model using S'_3

When we double the training set size, adding more training cases, using S'_3 , we get different formulas for estimating the PRAM and NUMA execution times.

NUMA: $t_n = a_n * LLS + b_n * N + c_n * N * LLS * \sqrt{N} + d_n * S * LLS^2$ where $a_n = 44201.802640319$, $b_n = 8.01445073789093$, $c_n = 0.0765782960682957$ and $d_n = -0.000128963606572698$.

PRAM: $t_p = a_p * LLS + b_p * P + N * LLS + c_p * P * LLS + d_p * N * C + e_p * LLS^2$ where $a_p = 107691.057352967$, $b_p = 2919.18227717314$, $c_p = 107.696490731206$, $d_p = 0.26706472514021$ and $e_p = -1605.81303793423$. Note that the coefficient for $S * LLS$ is 1.

Binary model using S'_3

Deriving the execution times as functions of N , P , C and LLS seems unstable for Eureqa Pro, i.e. very depending on the specific training sets (see Table 6.4). However, we are only interested in relative performance. To handle this we introduced a binary model using Eureqa Pro and S'_3 . It gives 1 as result if NUMA should be used and 0 for PRAM:

$$NUMA = P > 0.000905177867360653 * N + 0.055818680067241 * P * LLS \quad (6.1)$$

¹Due to large simulation times, a significantly larger number of training and evaluation samples was not feasible.

6.3.2 C5.0 decision trees

Running C5.0 on S'_1 we get the following decision tree:

```

LLS > 8: PRAM
LLS <= 8:
...P > 400: NUMA
  P <= 400:
    ...N <= 174851: NUMA
      N > 174851: PRAM

```

Running C5.0 on S'_2 we get the following decision tree:

```

LLS > 15: PRAM
LLS <= 15:
...N <= 177417: NUMA
  N > 177417:
    ...P <= 643: PRAM
      P > 643: NUMA

```

Running C5.0 on S'_3 we get the following decision tree:

```

LLS > 8: PRAM
LLS <= 8:
...N <= 210765: NUMA
  N > 210765:
    ...P <= 500: PRAM
      P > 500: NUMA

```

Note that no C5.0 model uses the C parameter and all the generated predictors are quite similar to each other.

6.4 Global optimization model

Consider a program that consists of K consecutive regions, $R_1, R_2, \dots, R_K$ as shown in Figure 6.2. Each region may contain parallelism, and we separate any two subsequent parallel regions from each other by having a (global) barrier synchronization in between. This is a natural way to program the REPLICA architecture; in PRAM mode each thread participates to in doing some computation. In the general case the work may be unbalanced across the PRAM threads, and hence we might need barrier synchronization at region exit to restore the lock-step synchronous thread execution with strict shared memory consistency across the entire chip. One region maps to a particular component which can have one or several implementation variants. Each region can potentially either execute in PRAM or NUMA mode, depending on existing implementations.

Since each region can (potentially) execute in either NUMA or PRAM mode, we model the possible combinations as a *directed acyclic graph* (DAG), which we call a selection DAG. In the selection DAG, nodes (P_k, N_k) represent different implementation variants which correspond to regions (R_k) executed in the different modes (PRAM, NUMA), and edges correspond to control flow transitions (sequencing). See Figure 6.3 for the selection DAG

Figure 6.2: Example of regions. Adopted from [52].

for a program with four regions in sequence. In this case there exists exactly one PRAM and one NUMA implementation per region. The edges are shown in Figure 6.3. An edge can only exist between a node in region R_k and its direct successor nodes in region R_{k+1} which can be either a PRAM node or a NUMA node. We have the following edges with weights: PRAM-to-PRAM $w_{P_k P_{k+1}}$, NUMA-to-NUMA $w_{N_k N_{k+1}}$, PRAM-to-NUMA $w_{P_k N_{k+1}}$, and NUMA-to-PRAM $w_{N_k P_{k+1}}$, for all $k = 1, \dots, K - 1$. In our case with the REPLICAS architecture the switching cost from PRAM to NUMA is fixed and denoted S , and also the cost of switching back from NUMA to PRAM is fixed and denoted B . Of course we always have the costs to execute the region. This means that, $w_{P_k P_{k+1}} = c(P_{k+1})$, $w_{N_k N_{k+1}} = c(N_{k+1})$, $w_{P_k N_{k+1}} = c(N_{k+1}) + S$ and $w_{N_k P_{k+1}} = c(P_{k+1}) + B$. See Figure 6.3 for an example.

Using the largest training set (which contains 98 cases, e.i S'_3) from section 6 we obtained from Eureqa pro the following predictors for the execution time (in clock cycles)²:

$$t_{PRAM} = 3533P + 13P * LLS + 3N * LLS + \frac{72449C + 3N - 568831}{LLS} \quad (6.2)$$

$$t_{NUMA} = 186037LLS + 5N * LLS + \frac{107467C}{LLS} - 1755527 - 3N \quad (6.3)$$

Using the above cost models and measurements for deriving the constant switching costs back and forth, we can for a specific problem assign costs on the DAG edges. Since we focus on execution time we want to find the shortest possible execution path from the start node to the end node, this is a well known problem and we use Dijkstras famous algorithm to solve it at runtime [19]. The result is a mode assignment for all regions including with an overall linear asymptotic runtime overhead.

6.5 Evaluation and comparison of Eureqa Pro and C5.0 models for mode selection

Table 6.3 shows the mode selection results of evaluating the Eureqa Pro models based on S_3 and S'_3 compared to actual results from test runs in the simulator. It might be interesting to note that the average speedup error is 15.1% for S_3 while it is 37.5% for the model based on S'_3 .

²Compared to the predictors obtained in section 6.3.1 we have here constrained Eureqa to used integers constants only.

Figure 6.3: Selection DAG representing that each region can execute in either PRAM or NUMA mode. Adopted from [52].

Table 6.4 shows the predicted mode for all cases in S_E for the models based on S'_1 , S'_2, S'_3 and S_3 . The columns are sorted by the number of misclassifications and also show the average real speedup gained when using the corresponding predictors.

Case	N	C	P	LLS	Case	N	C	P	LLS
1	307200	104	300	4	11	202099	56	273	4
2	153600	12	150	24	12	240096	56	841	4
3	307300	16	500	8	13	131407	40	2463	8
4	38400	12	1000	8	14	277705	96	2720	44
5	38400	24	1000	16	15	55311	56	999	1
6	222401	16	765	8	16	12155	88	212	8
7	62709	72	59	1	17	203909	48	362	1
8	286382	16	3475	92	18	262394	80	1437	32
9	78714	56	2034	44	19	49693	80	1467	24
10	32431	88	966	4	20	94664	0	1871	1

Table 6.2: The set of test cases, S_E , for evaluation and comparison.

For our problem and training sets Eureqa Pro using our first execution time based models seems to be some what unstable; it produces both their best and worst predictor depending on the training set, see Table 6.4. Eureqa Pro gives the best predictor with a smaller training set S_3 than the larger S'_3 . With our binary Eureqa Pro predictor based on set S'_3 we get only one misclassification, it seems as good as C5.0 for the same set. All predictors are very simple and can be implemented with a few instructions so the overhead of invoking them at runtime is very low.

All predictors give on average a speedup between 1.91 and 2.23. This is not a "magical range", it comes from the evaluation set S_E . In most cases misclassifications do not "hurt" so much since they are border cases and running in "wrong" mode will only affect performance marginally. If we remove the three worst predictors in terms of misclassifications and use the

Case	Real speedup	Eureqa S'_3 speedup	Eureqa S_3 speedup error %	Eureqa S'_3 speedup	Eureqa S'_3 speedup error %
1	0.98	0.99	1.5	1.83	86.6
2	0.42	0.45	7.3	0.17	60.4
3	0.98	0.96	3.0	0.66	33.2
4	5.45	2.29	58.1	1.33	75.7
5	0.84	0.96	14.7	0.95	13.0
6	1.26	1.23	2.3	0.42	66.9
7	0.99	1.00	1.2	1.25	26.8
8	0.41	0.45	7.7	0.61	46.4
9	0.56	0.55	2.2	0.52	7.3
10	3.79	3.35	11.5	2.69	28.9
11	1.02	1.02	0.1	0.60	40.8
12	1.31	1.35	2.5	0.93	29.5
13	3.03	2.89	4.6	0.98	67.6
14	0.47	0.64	37.8	0.70	49.7
15	3.92	3.68	6.1	3.68	6.1
16	2.04	0.89	56.3	1.74	14.8
17	1.19	1.14	4.5	1.19	0.0
18	0.46	0.66	45.1	0.51	12.1
19	0.77	0.78	2.1	0.85	10.5
20	11.60	7.72	33.5	3.12	73.1

Table 6.3: Real speedup from simulator, estimated speedup for Eureqa Pro using S'_3 and S_3 and the speedup error in % compared to the real speedup.

Case	Correct	Eureqa S'_3	C5.0 S'_2	C5.0 S'_1	Eureqa S_3	C5.0 S'_3	Eureqa bin S'_3
1	PRAM	NUMA	PRAM	PRAM	PRAM	PRAM	PRAM
2	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
3	PRAM	PRAM	NUMA	PRAM	PRAM	PRAM	PRAM
4	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
5	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	NUMA
6	NUMA	PRAM	NUMA	NUMA	NUMA	NUMA	NUMA
7	PRAM	NUMA	NUMA	NUMA	PRAM	NUMA	PRAM
8	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
9	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
10	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
11	NUMA	NUMA	PRAM	PRAM	NUMA	NUMA	NUMA
12	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
13	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
14	PRAM	NUMA	PRAM	PRAM	PRAM	PRAM	PRAM
15	NUMA	PRAM	NUMA	NUMA	NUMA	NUMA	NUMA
16	NUMA	PRAM	NUMA	NUMA	PRAM	NUMA	NUMA
17	NUMA	NUMA	PRAM	PRAM	NUMA	NUMA	NUMA
18	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
19	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM	PRAM
20	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA	NUMA
	Misclassifications	6/20	4/20	3/20	1/20	1/20	1/20
	Average real speedup	1.92	2.21	2.21	2.01	2.23	2.22

Table 6.4: Evaluation result using S_E . Misclassifications are marked in boldface. Average speedup for all 20 cases using the corresponding predictor.

Evaluation	N	C	P	LLS	Correct	Eureqa S'_3	C5.0 S'_2	C5.0 S'_1	Eureqa S_3	C5.0 S'_3	Eureqa bin S'_3
1	307200	60	8	1	PRAM	NUMA	PRAM	PRAM	PRAM	PRAM	PRAM
2	307200	80	8	1	PRAM	NUMA	PRAM	PRAM	PRAM	PRAM	PRAM

Table 6.5: Parameters and evaluation results using a 1D average computation example. Misclassifications are marked in boldface.

remaining predictors together with a majority voting algorithm we would eliminate all misclassifications and get an average speedup of 2.23. In Table 6.4 the speedup for C5.0 on S'_3 is also 2.23, even though it makes one misclassification, as this misclassification only changes the average speedup with 0.03% which can be explained by that there the PRAM and NUMA execution times are very similar. We also evaluated our different models on two different parallel 1D-average computations, the parameters and results are shown in Table 6.5. The only predictor that misclassifies is Eureqa on S'_3 .

6.6 Evaluation and results for optimized global composition

We use the same evaluation set as in previous section, see Table 6.6, from where we randomly pick different components. As a hardware implementation of a REPLICa version with NUMA mode is not available yet, the evaluation is done using our REPLICa cycle-accurate simulator, which however constrains the feasible evaluation set size and number of regions due to long simulation times. We used our REPLICa compiler with NUMA support that was introduced in section 5.6.1 (or in [43]). We evaluate composition for sequence of three and four regions, see Figures 6.4 and 6.5. For both cases we do twenty tests each where we randomly pick three respectively four components from our set of twenty components. Twenty tests can be considered very few compared to the amount of possible combinations. However, we are constrained by the simulation time.

In the case of composing for sequences of three regions, see Figure 6.4, we get a speedup between 0.84 and 2.9 compared to running in PRAM mode only. These speedup figures include the overhead of running the predictor, calculating the shortest path and switching to the selected execution mode where it is necessary. The average speedup (including overhead) for all tests in Figure 6.4 is 1.4. It is important to notice that in the cases where we get a speedup of 0.98 and 0.99 (i.e., slight slowdowns) are when it is optimal to stay in PRAM mode all the time and the slowdowns come from the overhead costs.

In the case of four regions, see Figure 4, we use the exactly same predictors as for three, i.e. Equations (6.2) and (6.3). When evaluating composition of four components we get speedups between 0.60 and 2.23 when including overhead, and average speedup including overhead is 1.17, see Figure 6.4. We only get speedup gains (including overhead) in thirteen out of twenty cases when combining four components. One reason might be that the more components we add the more unstable becomes the total execution time of the whole path, especially since the algorithm tries to find the cheapest/shortest way. Another reason might be that we were "unlucky" when randomly selecting components from the evaluation set. Both for the

Component	N	C	P	LLS
1	307200	104	300	4
2	153600	12	150	24
3	307300	16	500	8
4	38400	12	1000	8
5	38400	24	1000	16
6	222401	16	765	8
7	62709	72	59	1
8	286382	16	3475	92
9	78714	56	2034	44
10	32431	88	966	4
11	202099	56	273	4
12	240096	56	841	4
13	131407	40	2463	8
14	277705	96	2720	44
15	55311	56	999	1
16	12155	88	212	8
17	203909	48	362	1
18	262394	80	1437	32
19	49693	80	1467	24
20	94664	0	1871	1

Table 6.6: The set of synthetic components, S_E , for evaluation and comparison.

case of three and four components the overhead is almost constant.

6.7 Optimizing mode selection for loops

Assume we have a loop with K iterations whose body can execute in PRAM and NUMA mode just as in the case of REPLICA. In Figure 6.6 we show the loop unrolled and that it is possible to enter and exit the body in P (PRAM) and N (NUMA) mode. In the figure we do not show which mode we start in.

The body of the loop is modeled as a black box component having two input modes and two output modes. Each combination of input and output modes represent an implementation variant of the component, where we enter and exit in specific modes. The inputs and the outputs are connected using edges which are assigned edge costs. The edge costs represent the execution time of the component variant. Since we treat the component as a black box we cannot know if there are internal mode switches. However, this is not a problem since we only care about the cheapest path cost going

Figure 6.4: Speedup when combining three components.

Figure 6.5: Speedup when combining four components.

Figure 6.6: Example of a (unrolled) loop which can execute in two possible modes (P and N).

to and from the available modes. In the example in Figure 6.6, we have the following possible cheapest path costs:

- C_{PP} : we enter in PRAM mode and exit in PRAM mode.
- C_{NN} : we enter in NUMA mode and exit in NUMA mode.
- C_{PN} : we enter in PRAM mode and exit in NUMA mode.
- C_{NP} : we enter in PRAM mode and exit in NUMA mode.

We also have the switching costs:

- S the cost to switch to N from P .
- B the cost to switch to P from N (back).

Naturally, there is zero cost to stay in the current mode which is also shown in Figure 6.6.

In Figure 6.6 N_j and P_j represent the minimal execution cost for ending up in mode N or P respectively when executing j iterations, i.e. the shortest path given the initial start values.

Recall Dijkstras algorithm for finding the shortest path between two nodes in a graph [19]. Here we have a special case of graph where we only go "forward" and either remain in the same execution mode or we switch. Applying Dijkstras algorithm [19] on this regular case leads to the following recurrence relations:

$$P_j = \min(C_{PP} + \min(P_{j-1}, N_{j-1} + B), C_{NP} + \min(N_{j-1}, P_{j-1} + S)), 1 \leq j \leq K \quad (6.4)$$

$$N_j = \min(C_{NN} + \min(N_{j-1}, P_{j-1} + S), C_{PN} + \min(P_{j-1}, N_{j-1} + B)), 1 \leq j \leq K \quad (6.5)$$

$$\text{Final solution for } K \text{ iterations: } \min(N_K, P_K) \quad (6.6)$$

Case when starting in mode P:

$$\begin{aligned} N_0 &= S \\ P_0 &= 0 \end{aligned} \quad (6.7)$$

Case when starting in mode N:

$$\begin{aligned} N_0 &= 0 \\ P_0 &= B \end{aligned} \quad (6.8)$$

Equation 6.6 gives the final solution, e.g the cheapest cost for executing the loop K times³.

Equation 6.7 gives the initial starting mode and here we explicitly state that we start in mode P since there is only a switching cost if we go to mode N . If we do not care which mode we start in and are only interested in the shortest execution time of the loop we set $N_0 = P_0 = 0$.

One interesting observation is that the number of modes reflects the (maximal) unrolling factor of the loop that is needed for the optimal steady state⁴. This means that we at most need to unroll the loop two times (in the case of two modes) to come back to the same state. The main motivation can be seen in Equations 6.4 and 6.5. In each step (for N_j and P_j) we have two options where we (according to the algorithm) always take the minimal one. In the next step (for N_{j+1} and P_{j+1}) we do the same and since the previous step made an optimal decision we will either make a new selection or the same. However, in step $j + 2$ we will either make the same decision as in step j or $j + 1$ due to that the decisions are already optimal and we in this step have the exact same options to choose from. This leads to a fixed set of possible solutions. A similar argumentation can be done for any number of modes.

Our observation is very useful in practice since we do not need to fully unroll the loop and run Dijkstras algorithm to find the optimal execution path. Instead, we only need to unroll it M (number of mode) times and find the optimal solution for this unrolled loop body. Of course this requires much less computations than the fully unrolled loop if K is (much) larger than the number of modes.

For two modes the possible minimal solutions for $\min(N_K, P_K)$ can be described as two dimensional patterns: chain (unroll factor one) see Figure 6.7, saw (unroll factor one) see Figure 6.8. Finally we have zigzag (unroll factor two) see Figure 6.9. If we only care about steady state and do not care about if we start in mode P or N we have the following costs when the steady state is executed K' times:

- N-chain: $K' * C_{NN}$
- P-chain: $K' * C_{PP}$
- NP-zigzag: $\frac{K'}{2}C_{NP} + \frac{K'}{2}C_{PN}$
- PN-zigzag: $\frac{K'}{2}C_{PN} + \frac{K'}{2}C_{NP}$
- PN-saw: $K' * C_{PN} + B * (K' - 1)$
- NP-saw: $K' * C_{NP} + S * (K' - 1)$

³Note that here we do not care which mode (P or N) we exit the loop in.

⁴Here steady state denotes the case where we enter in the same mode as we exit and is recurring. To enter the steady state we might need a some prologue iterations, analogously we might need an epilogue

Figure 6.7: Example of N-chain and P-chain patterns.

Figure 6.8: Example of PN-saw and NP-saw patterns.

Depending on the edge costs (C_{NN} , C_{PP} , C_{PN} , C_{NP} , S and B) the different solutions will occur. For simplicity we assume that K' is even. However, the solutions when K' is odd will be similar. Using the same argumentation as in the case of deriving the maximal unrolling factor we know that $\frac{K}{2} - 2 \leq K' \leq K$: If the maximal unroll factor (2) is needed we need to execute the new loop $\frac{K}{2}$ times. We can also come in a case where we need to take the first iteration of the loop into the prologue and the last iteration into the epilogue, hence we need to subtract two iterations. If we do not need to unroll the loop at all and no prologue and epilogue is needed then $K' = K$.

6.8 Possible extension towards structural composition

As we now have a way to handle the execution costs of loops we can wrap a loop in a component. A full loop itself can have two input modes and two output modes, just like a general component. Using the method described

Figure 6.9: Example of PN-zigzag and NP-zigzag patterns. Note that they basically are the same but have different starting modes.

in the previous section we can decide the minimal C_{NN}, C_{PP}, C_{NP} and C_{PN} for the entire loop using the patterns (chain, saw and zigzag variants) together with their prologues and epilogues.

For a given program we can use a so-called region tree to show how combinations of components, sequences, loops over components, nested loops of components etc. relate. An example region tree of a function is shown in Figure 6.10.

In Figure 6.10 S represents the start node (start of the current function)⁵ and E the end of the program. L represents a loop-header and B a normal base level component. We call components in region tree for R-components to distinguish them from components (regions) in the compiler IR (AST). Between each two nodes there are two edges, each representing control flow with the execution modes (P and N). C is the computation, to be performed at run time, to calculate the possible shortest execution time of the function. As soon as we know all parameters at the entry of C to calculate the shortest execution time. The computation of the shortest path in Section 6.4 is actually the C computation in that case, even though we there only have a sequence of components. There we store, in an array, for each component (region) which mode it should be executed in. The computation done in C can be considered overhead, however the computation cost should also be part of the total execution time. In the general case we need a dispatch table to store the solution information. The table should contain which component variant should be invoked for each component call. Before each B and L is executed a lookup in the dispatch table is done to check in which mode to execute in. Doing this lookup is also considered overhead.

If we use our technique of unrolling the loop M times from the previous section instead of unrolling the loop fully, we do not need to have an entry for each loop iteration in the table; only for the prologue, steady state and epilogue. In the simplest form, the actual transformation of the loop into prologue-steady-state-epilogue form can be done at compile time. The size of the dispatch table will be linear in the number of R-components, since the unroll factor is a constant, M , (in REPLICAs case $M = 2$) and the prologue and epilog are also constant per loop. The dispatch table size also depends on M^l where l is the maximal nesting depth of the loops and also is a constant. The size of the table will be $\mathcal{O}(nM^l)$ where n is the number of R-components.

Instead of unrolling the loop into prologue, steady-state and epilogue in the code we can achieve the same effect by doing the unrolling in the dispatch table. An advantage is that code size will not grow as in the case of code unrolling. To support unrolling of loops in the dispatch table, the table can be implemented as a tree, representing the region tree. In this case the dispatch table size will also be $\mathcal{O}(nM^l)$ where l is the depth of the tree. Hence, from the point of dispatch table size and code size there is no gain by unrolling the code itself.

⁵REPLICA always boots up in PRAM mode.

Figure 6.10: Example of a region tree of a function.

We have in this section given some proposal how we can calculate the shortest path for a sequence of components, including nested loops over components, and achieve optimized structural composition at runtime. An implementation is subject to future work.

6.9 Related work

Software composition is wide research area, for example it ranges from how to compose web services concerning quality of service (QoS) [73] to speedup scientific computations on GPU-based systems [14]. The problem of selecting the best runtime mode for REPLICA is very related to the implementation selection problem for heterogenous systems such as CPU-GPU based systems; in both REPLICA and CPU-GPU case there are overhead costs of switching and data transfers costs. Similarities such as small local memory also exist.

The parallel programming language framework PetaBricks [6] uses auto-tuning that effectively explores the search space to select from multiple user provided implementations the best one, depending on problem parameters [6]. Their compiler can generate OpenCL code that can execute on GPUs [79].

SkePU is a C++ skeleton programming library mainly for mapreduce problems with back-ends for both CPU and GPUS. It supports implementation selection using machine learning methods to adopt skeletons to a given platform [22].

One example where C5.0 has been used for performance optimization of heterogeneous systems is [70], they use it to prune the search space when doing off-line tuning of component composition.

Danylenko et al. compared different machine learning approaches for context-aware composition in [12]. They consider decision tress, decision diagrams, naive bayes and suport vector machine classifiers. They evaluate their results on three different multicore machines.

In [48] Grewe and O'Boyle show how to select optimized mappings of

OpenCL tasks on a heterogeneous CPU-GPU system to get good load balancing. They base their training on the support vector machine (SVM) model, and is based on static features (number of floating point operations etc) just like we do. However hybrid execution is not possible on REPLICA as the same hardware is used by both PRAM and NUMA mode.

Elastic computing is a framework that supports heterogenous computing, such multicore CPUs combined with FPGA accelerators. It separates functionality and implementations using elastic functions which can be executed with different parameters (input size etc) on different target architectures [97]. They use linear regression based to predict execution time based input size and other metrics captured by the specific performance model for each component.

Dastgeer et. al have developed the PEPPIER composition tool which can be used to compose components for heterogeneous CPU and multi-GPU systems in a performance aware way [13]. The componets are written in C/C++ and annotated for composition with tunbale parameters. It does not perform global optimized composition, instead they rely on a greedy local optimization scheme. The tool both include a composition and optimization framework and a runtime system based on StarPU [13].

Kessler and Löwe do optimized composition of performance-aware parallel components using look-up tables containing the expected best performance given parameters such as problem size, available cores etc. For evaluation they use different implementation variants of sorting [62].

Grewe and O’Boyle use support vector machines (SVM) to get good load balancing of OpenCL tasks with hybrid execution on heterogenous CPU-GPU systems [48].

In [23, Chapter 5] Ericsson suggests, in a very general way, to use a dependency graph similar to our selection DAGs as in Figure 6.3 with the variations as "parallel" nodes. He also suggests that a shortest path calculation can be used to minimize execution time. However, he does not go into any details. For example he just states that user-specified cost functions can be used to assign weights on the edges in the graph, but not how to derive them.

In [14, Chapter 6] Dastgeer describes his Global Composition Framework (GCF) where sequences of component calls are optimized using a bulk scheduling heuristic. The heuristic checks which of the predicted run times is faster if *all* calls are executed in bulk, either on CPU or GPU. This is very naïve compared to our approach; it would basically mean that we would not have any cross-edges between the modes in our selection DAG and the problem would be trivial to solve (i.e. no shortest path calculation needed).

As far we as know Eureka Pro has not been used before for performance prediction and implementation selection and optimized composition before, however in [46] Goel uses Eureka for per-core power estimation and power aware scheduling for CMPs which is a related problem area.

6.10 Conclusion

We used state-of-the-art machine learning methods, decision trees and symbolic regression, and tools based on them, namely C5.0 and Eureqa Pro. Using the same training data we derive models to predict if to run the REPLICA architecture in PRAM or NUMA mode for a certain parameterized computation type (parallel stencil operation). We also combine different components in an optimized way using derived cost predictors and shortest path computation. It is important to notice that the derived execution times are average execution times and not worst case. Our techniques and methods focus on speed-up in the average case. Without machine learning it had not been possible to derive predictors of when to use PRAM or NUMA mode.

We also showed how it is possible to handle loops without unrolling them fully. An interesting observation is that the number of execution modes reflects the (maximal) unrolling factor of the loop that is needed for the optimal steady state.

6.10.1 Mode selection

The best predictors give a misclassification rate of 5%. Combining the three best ones using a majority voting algorithm misclassifications can be eliminated fully, at least on the test case. Average gained speedup ranges between 1.92 and 2.23 for all classifiers on the test cases.

For C5.0 it seems that adding more training data improves the accuracy while for Eureqa Pro more training data can generate more instable models. However, Eureqa Pro can be as good as C5.0 if the right training data is used and then it makes correct classification for the case where all other predictors are wrong. The derived formulas for PRAM and NUMA execution time are not accurate enough to predict the execution time, however they are accurate enough to be used for deciding if PRAM or NUMA mode should be used.

All the derived predictors are very simple, and can be implemented with a few computation and comparison instructions. The overhead of invoking them at runtime, if some parameters such as size are unknown statically, will be very small. As far as we know, Eureqa Pro has not been used for this type of performance predictions before.

6.10.2 Global optimization

For the case of multiple components or regions, we have proposed a way to optimize execution mode and variant selection for REPLICA by combining machine learning (symbolic regression) for generating cost models together with shortest path calculation. One goal was to both have cost models/predictors which are fast to evaluate and also an optimization problem that is fast to solve.

We evaluated our approach using parameterized generic stencil operations and derived cost models for them using the tool Eureka pro [84, 83].

For the case of composing three components, from our evaluation set, we achieved speedups between 0.84 and 2.9 including overhead of evaluating the cost functions, solving the shortest path problem and launching the selected component versions. The overhead cost is only a few percent. The average speedup for the twenty tests of composing three components was 1.4.

However, the evaluation of four components was not as fruitful. We achieved speedups in thirteen out of twenty cases, in the range of 0.60 and 2.23 including overhead, the average speedup was 1.17.

Our approach is very generic, basically any cost models/predictors can be used. Also it is very easy to extend the model to support more modes than two. Also, if there are different/more implementations for the same regions they would be inserted as parallel nodes in the graph.

As far as we are aware, we are the first to combine lightweight cost functions for implementation variants generated by machine learning with the shortest-path problem to optimize composition of multiple components

6.11 Future work

Future work includes using the same methods on other problem types than stencil-like algorithms. It would also be interesting to test this on heterogeneous systems such as CPU-GPU based ones, especially if we apply our technique for loop unrolling. Another interesting problem would be to derive parameterized models of algorithms using a pattern matching framework such as PRT [85] and combine it with machine learning. Each pattern could then be annotated with a predictor for the pattern implementations' best performance for specific parameter values and for a given type of hardware.

Chapter 7

Concluding remarks and future work

In this chapter we will give some concluding remarks of the results presented in this thesis and suggest some possible future work.

The first chapters in this thesis are very tightly coupled to the REPLICA architecture. The chapters contain evaluations of both PRAM and NUMA mode which also motivate the existence of both modes. The chapters also introduce the compiler and show how code generation is done to utilize PRAM and NUMA mode. The REPLICA specific optimizations provided by the back-end focus on how to exploit the chained functional units, but only on basic block level. Future work would be to go beyond basic blocks.

The evaluations show that PRAM mode is very suitable for irregular memory access patterns and control flow, and also that NUMA mode can accelerate some programs with regular memory access patterns. However, for a particular program it is not always obvious which mode, PRAM or NUMA, will show best performance. In Chapter 6 we tackle this by providing a case study for generic stencil computations using machine learning derived cost models in order to automatically select at runtime which mode to execute in. It is extended to also cover sequences of kernel invocations.

In contrast to the first chapters the methods and techniques presented in Chapter 6 for mode selection are more general and not coupled to the REPLICA architecture specifically, even though we do our case study on the REPLICA architecture. Our so-called parallel regions are defined by global synchronization points, which suits many parallel programming paradigms. Even though we only did our case studies for two runtime modes, it is easy to extend the model by having more parallel tracks of nodes, each representing a different execution mode or a possible hybrid execution variant. Future work includes to evaluate this on other architectures, for example CPU-GPU systems. Using state-of-the-art machine learning is a pragmatic way to derive cost functions for the implementation variants that maps to

each region. These cost functions can in many cases be lightweight and fast to evaluate at runtime. The shortest path formulation of the optimization problem of mode selection for a sequence of kernel invocations is also lightweight and fast to solve at runtime. Finally, we show how it is possible to handle loops with out unrolling them fully. An interesting observation is that the number of execution modes reflects the (maximal) unrolling factor of the loop that is needed for the optimal steady state. We also give some ideas how extend our model to allow nesting regions and regions sequences inside loops. The next natural step would be to implement and evaluate the models for loops and nested regions both for CPU-GPU systems and REPLICA.

Chapter 8

Glossary

- **AESM** Asynchronous emulated shared memory
- **ALU** Arithmetic logic unit
- **BFS** Breadth first search
- **CESM** Configurable emulated shared memory
- **CMP** Chip multiprocessor
- **CPU** Central processing unit
- **CRCW** Concurrent read concurrent write
- **DAG** Directed acyclic graph
- **DSP** Digital signal processor
- **DWT** Discrete wavelet transform
- **ESM** Emulated shared memory
- **FB** Freeze bunch
- **FP** Freeze processor
- **FPGA** Field-programmable gate array
- **FU** Functional unit
- **GPU** Graphics processing unit
- **IDCT** Inverse discrete cosine transform
- **IDWT** Inverse discrete wavelet transform
- **ILP** Instruction level parallelism

- **IR** Intermediate representation
- **LER** Load with explicit receive
- **LLVM** Low level virtual machine
- **MIMD** Multiple instruction multiple data
- **MPU** Messages passing interface
- **MTCU** Master thread control unit
- **MU** Memory unit
- **NUMA** Non uniform memory access
- **PRAM** Parallel random access machine
- **SIMD** Single instruction multiple data
- **SMP** Symmetric multiprocessing
- **SPMD** Single program multiple data
- **SVM** Support vector machine
- **TCU** Thread control unit
- **TLP** Thread level parallelism
- **UMA** Uniform memory access
- **VILP** Virtual instruction level parallelism
- **VLIW** Very long instruction word
- **WB** Write back
- **XMT** eXplicit multi threading

Bibliography

- [1] F. Abolhassan, J. Keller, and W.J. Paul. On the Cost-Effectiveness of PRAMs. In *Proceedings, 3rd IEEE Symposium on Parallel and Distributed Computing, ACM Special Interest Group on Computer Architecture, and IEEE Computer Society*, pages 2 – 9, 1991.
- [2] Alexander Aiken and Alexandru Nicolau. A development environment for horizontal microcode. *IEEE Trans. Software Eng.*, 14(5):584–594, 1988.
- [3] D. Åkesson. An LLVM Back-end for REPLICA: Code Generation for a Multi-core VLIW processor with Chaining. Master’s thesis, Linköping, 2012.
- [4] Akhtar Ali, Usman Dastgeer, and Christoph Kessler. OpenCL for programming shared memory multicore CPUs. In *Proceedings of the 5th Workshop on MULTIPROG, in conjunction with HiPEAC*, 2012.
- [5] Christoph Kessler Amin Shafiee Sarvestani, Erik Hansson. Towards domain specific automatic parallelization. In *Proc. MCC’12 Fifth Swedish Workshop on Multicore Computing*, November 2012.
- [6] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.
- [7] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiawicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10), October 2009.
- [8] OpenMP Architecture Review Board. Openmp application program interface. <http://www.openmp.org/mp-documents/spec30.pdf>, 2008. [Online; accessed 02-September-2013].

- [9] G.C. Caragea, F. Keceli, A. Tzannes, and Uzi Vishkin. General-purpose vs. gpu: Comparison of many-cores on irregular workloads. In *Proceedings of the Second Usenix Workshop on Hot Topics in Parallelism*, 2010.
- [10] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. Guided region-based GPU scheduling: Utilizing multi-thread parallelism to hide memory latency. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 441–451, 2013.
- [11] Steven Dalton and Nathan Bell. Cusp. <http://people.cs.kuleuven.be/~albert-jan.yzelman/software.php>. [Online; accessed 12-December-2013].
- [12] Antonina Danylenko, Christoph Kessler, and Welf Löwe. Comparing machine learning approaches for context-aware composition. In Sven Apel and Ethan Jackson, editors, *Software Composition*, volume 6708 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin Heidelberg, 2011.
- [13] U. Dastgeer, Lu Li, and C. Kessler. The PEPPER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-Based Systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion*., pages 711–720, Nov 2012.
- [14] Usman Dastgeer. *Performance-aware Component Composition for GPU-based systems*. PhD thesis, Linköping University, Software and Systems, The Institute of Technology, 2014.
- [15] Usman Dastgeer and Christoph Kessler. Conditional component composition for GPU-based systems. In *MULTIPROG-2014 workshop at HiPEAC-2014*, January 2014.
- [16] Tim Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. <http://www.cise.ufl.edu/research/sparse/matrices/index.html>. [Accessed 20-September-2013].
- [17] Roman Dementiev, Michael Klein, and Wolfgang J. Paul. Performance of mp3d on the sb-pram prototype. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par'02 Parallel Processing*, volume 2400 of *LNCS*. Springer Berlin Heidelberg, 2002.
- [18] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. *SIAM Journal on Computing*, 23(4):738 – 761, August 1994.

- [19] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [20] Evelyn Duesterwald, Rajiv Gupta, and MaryLou Soffa. Register pipelining: An integrated approach to register allocation for scalar and subscripted variables. In Uwe Kastens and Peter Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*, pages 192–206. Springer Berlin Heidelberg, 1992.
- [21] James Alexander Edwards and Uzi Vishkin. Brief announcement: speedups for parallel graph triconnectivity. In *Proceedings of the 24th ACM symposium on parallelism in algorithms and architectures*, SPAA '12, pages 190–192, New York, NY, USA, 2012. ACM.
- [22] Johan Enmyren and Christoph W. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.
- [23] Morgan Ericsson. *Composition and Optimization*. PhD thesis, Växjö University, School of Mathematics and Systems Engineering, 2008.
- [24] Mattias Eriksson. *Integrated Code Generation*. PhD thesis, Linköping University, Department of Computer and Information Science, The Institute of Technology, 2011.
- [25] Mattias V. Eriksson and Christoph W. Kessler. Integrated modulo scheduling for clustered vliw architectures. In *HiPEAC*, pages 65–79, 2009.
- [26] Ian Finlayson, Gang-Ryung Uh, David Whalley, and Gary Tyson. An overview of static pipelining. *IEEE Computer Architecture Letters*, 11(1):17–20, 2012.
- [27] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, 1981.
- [28] Arno Formella, Jörg Keller, and Thomas Walle. HPP: A high performance PRAM. In *Euro-Par, Vol. II*, 1996.
- [29] M. Forsell. A Scalable High-Performance Computing Solution for Network on Chips. *IEEE Micro*, 22(5 (September-October)):46–55, 2002.
- [30] M. Forsell. Advanced simulation environment for shared memory network-on-chips. In *Proceedings of the 20th IEEE NORCHIP Conference*, pages 31–36, Copenhagen, Denmark, November 2002.

- [31] M. Forsell. Using parallel slackness for extracting ilp from sequential threads. In *Proceedings of the SSGRR-2003s*, L'Aquila, Italy, 2003.
- [32] M. Forsell. Compiling thread-level parallel programs with a C-compiler. In *Proceedings of the IV Jornadas sobre Programacion y Lenguajes*, pages 215–226, Malaga, Spain, November 2004.
- [33] M. Forsell. Designing NOCs with a parallel extension of C. In *FDL'04*, pages 463–475, 2004.
- [34] M. Forsell. E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs. *WSEAS Trans. on Computers*, 3(3):807–812, jul 2004.
- [35] M. Forsell. Ec - a compiler for the e-language. In *Proceedings of the 2004 International Symposium on System-on-Chip*, pages 157–160, 2004.
- [36] M. Forsell. Parallel Application Development Scheme for General Purpose NOCs. In *Proceedings of the 2005 ECTI International Conference (ECTI-CON)*, Pattaya, Thailand, pages 819–822, 2005.
- [37] M. Forsell. Realizing multioperations for step cached mp-socs. In *Proc. SOC'06*, pages 77–82, 2006.
- [38] M. Forsell. Configurable emulated shared memory architecture for general purpose MP-SOCs and NOC regions. *Networks-on-Chip, International Symposium on*, 0:163–172, 2009.
- [39] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. In *Proceedings of the 12th Workshop on Advances in Parallel and distributed Computational Models (in conjunction with the 24th IEEE International Parallel and Distributed Processing Symposium, IPDPS'10), April 19, 2010, Atlanta, USA*, pages 1–8, 2010.
- [40] M. Forsell. *TOTAL ECLIPSE – An Efficient Architectural Realization of the Parallel Random Access Machine*, pages 39–64. IN-TECH, Vienna, 2010. Editor: Alberto Ros.
- [41] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. *International Journal of Networking and Computing*, 1(1):21–35, 2011.
- [42] Martti Forsell. Replica. <http://www.vtt.fi/sites/replica/?lang=en>. [Online; accessed 02-September-2013].
- [43] Martti Forsell, Erik Hansson, Christoph Kessler, Jari-Matti Mäkelä, and Ville Lepänen. Numa Computing with Hardware and Software Co-support on Configurable Emulated Shared memory Architectures. *International Journal of Networking and Computing*, 4(1), 2014.

- [44] Martti Forsell, Erik Hansson, Christoph Kessler, Jari-Matti Mäkelä, and Ville Leppänen. Hardware and software support for numa computing on configurable emulated shared memory architectures. In *15th Workshop on Advances on Parallel and Distributed Processing Symposium (APDCM 2013), IPDPS-2013 Workshop proceedings*, May 2013.
- [45] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of 10th ACM STOC*, pages 114–118. Association for Computing Machinery, New York, 1978.
- [46] Bhavishya Goel. Per-core Power Estimation and Power Aware Scheduling Strategies for CMPs, 2011. 70.
- [47] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.
- [48] Dominik Grewe and Michael F.P. O’Boyle. A static task partitioning approach for heterogeneous systems using opencl. In Jens Knoop, editor, *Compiler Construction*, volume 6601 of *Lecture Notes in Computer Science*, pages 286–305. Springer Berlin Heidelberg, 2011.
- [49] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.
- [50] Erik Hansson, Erik Alnervik, Christoph Kessler, and Martti Forsell. A quantitative comparison of emulated shared memory architectures to current multicore cpus and gpus. In *Proceedings of MCC’2013*, 2013.
- [51] Erik Hansson, Erik Alnervik, Christoph Kessler, and Martti Forsell. A quantitative comparison of pram based emulated shared memory architectures to current multicore cpus and gpus. In *Architecture of Computing Systems (ARCS), 2014 27th International Conference on*, pages 1–7, Feb 2014.
- [52] Erik Hansson and Christoph Kessler. Global optimization of execution mode selection for the reconfigurable PRAM-NUMA multicore architecture REPLICA. In *Proc. of 6th International Workshop on Parallel and Distributed Algorithms and Applications (PDAA’14) in conjunction with CANDAR’14*, December 2014.
- [53] Erik Hansson and Christoph Kessler. Optimized selection of runtime mode for the reconfigurable PRAM-NUMA architecture REPLICA using machine-learning. In *Accepted for the 7th International Workshop on Multi-/Many-Core Computing Systems (MuCoCoS’14) in conjunction with Euro-par 2014*, August 2014.
- [54] Erik Hansson, Joar Sohl, Christoph Kessler, and Dake Liu. Case study of efficient parallel memory access programming for the embedded

- heterogeneous multicore dsp architecture epuma. In *Proceedings of the 2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, CISIS '11, pages 624–629, Washington, DC, USA, 2011. IEEE Computer Society.
- [55] Intel. Intel Xeon Processor 5600 series, data sheet, volume 1, 2011.
 - [56] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley, Reading, 1992.
 - [57] J.A. Kahle, M.N. Day, H.P. Hofstee, C.R. Johns, T.R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM journal of research and development*, 49(4/5):589, 2005.
 - [58] Jörg Keller, Christoph Kessler, and Jesper Larsson Träff. *Practical Pram Programming*. John Wiley & Sons, Inc., New York, NY, USA, 2000.
 - [59] B.W. Kernighan and D.M. Ritchie. *The C Programming Language*, ("ANSI C"). Prentice Hall,(ISBN: 0-13-110362-8), 1988.
 - [60] C. Kessler and E. Hansson. Flexible Scheduling and Thread Allocation for Synchronous Parallel Tasks. In *Proc. of 10th Workshop on Parallel Systems and Algorithms (PASA'12)*, 2012.
 - [61] Christoph. Kessler. Compiling for vliw dsps. In Shuvra S. Bhattacharyya, Ed F. Deprettere, Rainer Leupers, and Jarmo Takala, editors, *Handbook of Signal Processing Systems*, pages 603–638. Springer US, 2010.
 - [62] Christoph Kessler and Welf Löwe. A framework for performance-aware composition of explicitly parallel components. In *Proc. ParCo-2007 conference, Jülich/Aachen, Germany, Sep. 2007. In C. Bischof et al. (eds.): Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing Series, Volume 15, IOS Press*, pages 227–234, February 2008.
 - [63] C.W. Kessler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *Int. Journal of parallel programming*, 1997.
 - [64] Martin Kessler, Erik Hansson, and Christoph Kessler. Exploiting instruction level parallelism for replica - a configurable vliw architecture with chained functional units. In *Proc. 18th Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'12)*, July 2012.
 - [65] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

- [66] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 330–335, Washington, DC, USA, 1997. IEEE Computer Society.
- [67] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *Comput. Archit. News*, 38(3), June 2010.
- [68] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, 1992.
- [69] V. Leppänen. *Studies on the realization of PRAM, Dissertation 3*. Turku Centre for Computer Science, University of Turku, 1996.
- [70] Lu Li, Usman Dastgeer, and Christoph Kessler. Pruning strategies in adaptive off-line tuning for optimized composition of components on heterogeneous systems. In *Accepted for Proc. Seventh International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2) at ICPP*, 2014.
- [71] J. Mäkelä, E. Hansson, and M. Forsell. REPLICA language specification, 2013. In preparation.
- [72] Jari-Matti Mäkelä, Erik Hansson, Daniel Åkesson, Martti Forsell, Christoph Kessler, and Ville Leppanen. Design of the language replica for hybrid pram-numa many-core architectures. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications, ISPA '12*, pages 697–704, Washington, DC, USA, 2012. IEEE Computer Society.
- [73] D.A Menasce. Composing Web services: a QoS view. *Internet Computing, IEEE*, 8(6):80–90, Nov 2004.
- [74] D. Naishlos, J. Nuzman, C. W Tseng, and Uzi Vishkin. Evaluating the XMT parallel programming model. *High-Level Parallel Programming Models and Supportive Environments*, pages 95–108, 2001.
- [75] Nvidia. Cuda toolkit. <https://developer.nvidia.com/cuda-toolkit>. [Online; accessed 11-September-2013].
- [76] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7):28–32, 2010.

- [77] Gregoire Pau. Fast discrete biorthogonal cdf 5/3 wavelet forward and inverse transform (lifting implementation). <http://www.embl.de/~gpau/misc/dwt53.c>. [Online; accessed 06-September-2010].
- [78] Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real PRAM programming. In *Proc. Euro-Par'02*, August 2002.
- [79] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. *SIGPLAN Not.*, 48(4):431–444, March 2013.
- [80] Ross Quinlan. C5.0 release 2.07 GPL Edition [software]. <http://www.rulequest.com/download.html>, 2014.
- [81] S. Fuller R. Swan and D. Siewiorek. Cm* – A Modular Multiprocessor. In *Proceedings of NCC*, pages 645–655, 1977.
- [82] A.G. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.
- [83] M. Schmidt and H. Lipson. Eureka (version 0.99.5 beta) [software]. <http://www.nutonian.com>, 2014.
- [84] Michael Schmidt and Hod Lipson. Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324(5923):81–85, 2009.
- [85] Amin Shafiee Sarvestani, Erik Hansson, and Christoph Kessler. Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *International Journal of Parallel Programming*, pages 1–19, 2012.
- [86] Amin Shafiee Sarvestani, Erik Hansson, and Chrustoph Kessler. Extensible pattern recognition in DSP programs using Cetus. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with 20. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT11)*, 2011.
- [87] Kevin Skadron. The Rodinia benchmark suite, version 2.4. https://www.cs.virginia.edu/~skadron/wiki/rodinia/index.php/Main_Page. [Online; accessed 11-September-2013].
- [88] D.B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys (CSUR)*, 30(2):123–169, 1998.
- [89] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [90] M. Tokoro, E. Tamura, and T. Takizuka. Optimization of microprograms. *IEEE Trans. Comput.*, 30(7):491–504, July 1981.

- [91] U. Vishkin. Towards Realizing a PRAM-on-Chip Vision, 2007. Workshop on Highly Parallel Processing on a Chip (HPPC), August 28, Rennes, France (see <http://www.hppcworkshop.org/HPPc07/talks.html>).
- [92] Uzi Vishkin. Using simple abstraction to reinvent computing for parallelism. *Commun. ACM*, 54:75–85, January 2011.
- [93] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar’10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.
- [94] P. Wadler and S. Blott. How To Make Ad-Hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’89, pages 60–76, New York, NY, USA, 1989. ACM.
- [95] X. Wen and U. Vishkin. FPGA-based Prototype of a PRAM-on-chip Processor. In *CF ’08: Proceedings of the 2008 conference on Computing frontiers*, pages 55–66, New York, NY, USA, 2008. ACM.
- [96] Xingzhi Wen. *Hardware design, prototyping and studies of the explicit multi-threading (XMT) paradigm*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 2008.
- [97] John Robert Wernsing and Greg Stitt. Elastic computing: A framework for transparent, portable, and adaptive multi-core heterogeneous computing. *SIGPLAN Not.*, 45(4):115–124, April 2010.
- [98] Graham Wood. Global optimization of microprograms through modular control constructs. In *Proceedings of the 12th annual workshop on Microprogramming*, MICRO 12, pages 1–6, Piscataway, NJ, USA, 1979. IEEE Press.
- [99] Zhang Xianyi, Wang Qian, and Zaheer Chothia. OpenBLAS, version 0.2.8. <http://xianyi.github.io/OpenBLAS/>. [Online; accessed 13-September-2013].
- [100] A. N. Yzelman. Sparse library. <http://people.cs.kuleuven.be/~albert-jan.yzelman/software.php>. [Online; accessed 12-December-2013].

Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

- No 17 **Vojin Plavsic:** Interleaved Processing of Non-Numerical Data Stored on a Cyclic Memory. (Available at: FOA, Box 1165, S-581 11 Linköping, Sweden. FOA Report B30062E)
- No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.
- No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.
- No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.
- No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.
- No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.
- No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.
- No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.
- No 73 **Ola Strömfors:** A Structure Editor for Documents and Programs, 1986.
- No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.
- No 104 **Shamsul I. Chowdhury:** Statistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.
- No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.
- No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.
- No 113 **Ralph Rönquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.
- No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.
- No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.
- No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.
- No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.
- No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.
- No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.
- No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.
- No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.
- No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.
- No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.
- No 177 **Peter Åberg:** Design of a Multiple View Presentation and Interaction Manager, 1989.
- No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.
- No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.
- No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.
- No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.
- No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.
- No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.
- No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.
- No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.
- No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.
- No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.
- No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.
- No 253 **Torbjörn Näslund:** SL DFA-Resolution - Computing Answers for Negative Queries, 1990.
- No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.
- No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.
- No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.
- No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.
- No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.
- No 326 **Andreas Kägedal:** Logic Programming with External Procedures: an Implementation, 1992.
- No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.
- No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.
- No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.
- No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.
- No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.
- No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.
- No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

- No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
- No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
- No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
- No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
- No 398 **Johan Boye:** Dependency-based Groudnness Analysis of Functional Logic Programs, 1993.
- No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
- No 406 **Anna Moborg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
- No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
- No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
- No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
- No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
- No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
- FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
- FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
- No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
- No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
- No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
- No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
- No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
- No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
- FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
- No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
- No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
- No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
- No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
- No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
- No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
- No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
- No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
- FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
- No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
- No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
- No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
- No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
- No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
- No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
- FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
- FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
- No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
- No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
- No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
- No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
- No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
- No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
- No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
- FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
- No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
- No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
- No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
- No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
- No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
- No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

- No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
- No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
- No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
- No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
- No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
- No 591 **Niclas Wahlöf:** A Default Extension to Description Logics and its Applications, 1996.
- No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
- No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
- No 598 **Rego Granlund:** C³Fire - A Microworld Supporting Emergency Management Training, 1997.
- No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
- No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
- No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
- FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
- FiF-a 6 **Tommy Wedlund:** Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
- No 615 **Silvia Coradeschi:** A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
- No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
- No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
- No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
- No 629 **Gunilla Iwefors:** Krigsspel och Informationsteknik inför en oförutsägbart framtid, 1997.
- No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
- No 639 **Jukka Mäki-Turja:** Smalltalk - a suitable Real-Time Language, 1997.
- No 640 **Juha Takkinen:** CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
- No 643 **Man Lin:** Formal Analysis of Reactive Rule-based Programs, 1997.
- No 653 **Mats Gustafsson:** Bringing Role-Based Access Control to Distributed Systems, 1997.
- FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
- No 674 **Marcus Bjärelund:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
- No 676 **Jan Håkegård:** Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
- No 668 **Per-Ove Zetterlund:** Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
- No 675 **Jimmy Tjäder:** Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
- FiF-a 14 **Ulf Melin:** Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
- No 695 **Tim Heyer:** COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
- No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
- FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
- No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
- No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
- No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
- No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
- No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
- No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
- No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
- No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
- FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
- FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
- No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
- No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
- FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
- No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
- No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
- No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
- No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
- No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

- No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
- No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
- No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
- No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
- FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
- No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
- No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
- No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
- No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
- No 800 **Anders Subotic:** Software Quality Inspection, 1999.
- No 807 **Svein Bergum:** Managerial communication in telework, 2000.
- No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
- FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
- No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
- No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
- No 823 **Lars Hult:** Publika Gränssytor - ett designexempel, 2000.
- No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
- FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
- No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
- No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
- FiF-a 37 **Ewa Braf:** Organisationers kunskapsverksamheter - en kritisk studie av "knowledge management", 2000.
- FiF-a 40 **Henrik Lindberg:** Webaserade affärsprocesser - Möjligheter och begränsningar, 2000.
- FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
- No 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
- No 863 **Dan Lawesson:** Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
- No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
- No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
- No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
- FiF-a 47 **Per-Arne Segerkvist:** Webaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
- No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
- No 906 **Lin Han:** Secure and Scalable E-Service Software Delivery, 2001.
- No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
- No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
- FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
- FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
- No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
- No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
- No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
- No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
- No 938 **Bourhane Kadmiry:** Fuzzy Control of Unmanned Helicopter, 2002.
- No 942 **Patrik Haslum:** Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
- No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
- FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
- No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
- No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
- No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
- FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
- No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
- No 982 **Diana Zentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
- No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
- No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
- No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

- No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
- No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
- No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
- No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
- FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
- No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
- No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
- No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
- No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
- No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
- No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
- No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
- FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
- No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
- No 1034 **Arja Vainio-Larsson:** Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
- No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befärade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
- FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
- No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
- No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
- No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
- FiF-a 71 **Emma Eliason:** Effekttanalys av IT-systems handlingsutrymme, 2003.
- No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
- No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
- FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
- No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
- No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
- FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
- No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
- No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensive Data Mining Models, 2004.
- No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
- No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
- No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
- FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
- No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
- No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
- No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
- No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
- No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
- No 1149 **Vaida Jakonienė:** A Study in Integrating Multiple Biological Data Sources, 2005.
- No 1156 **Abdil Rashid Mohamed:** High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
- No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
- No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
- FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
- No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
- No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
- No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
- FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
- No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
- FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
- No 1172 **Petter Ahlström:** Affärsstrategier för seniorbostadsmarknaden, 2005.
- No 1183 **Mathias Cöster:** Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
- No 1184 **Åsa Horzella:** Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
- No 1185 **Maria Kollberg:** Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
- No 1190 **David Dinka:** Role and Identity - Experience of technology in professional settings, 2005.

- No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
- No 1192 **Nicklas Bergfeldt**: Towards Detached Communication for Robot Cooperation, 2005.
- No 1194 **Dennis Maciuszek**: Towards Dependable Virtual Companions for Later Life, 2005.
- No 1204 **Beatrice Alenljung**: Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
- No 1206 **Anders Larsson**: System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
- No 1207 **John Wilander**: Policy and Implementation Assurance for Software Security, 2005.
- No 1209 **Andreas Käll**: Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
- No 1225 **He Tan**: Aligning and Merging Biomedical Ontologies, 2006.
- No 1228 **Artur Wilk**: Descriptive Types for XML Query Language Xcerpt, 2006.
- No 1229 **Per Olof Pettersson**: Sampling-based Path Planning for an Autonomous Helicopter, 2006.
- No 1231 **Kalle Burbeck**: Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
- No 1233 **Daniela Mihailescu**: Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
- No 1244 **Jörgen Skågeby**: Public and Non-public gifting on the Internet, 2006.
- No 1248 **Karolina Eliasson**: The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
- No 1263 **Misook Park-Westman**: Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
- FiF-a 90 **Amra Halilovic**: Ett praktikerspektiv på hantering av mjukvarukomponenter, 2006.
- No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
- No 1277 **Viacheslav Izosimov**: Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
- No 1283 **Håkan Hasewinkel**: A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
- FiF-a 91 **Hanna Broberg**: Verksamhetsanpassade IT-stöd - Design teori och metod, 2006.
- No 1286 **Robert Kaminski**: Towards an XML Document Restructuring Framework, 2006.
- No 1293 **Jiri Trnka**: Prerequisites for data sharing in emergency management, 2007.
- No 1302 **Björn Häggglund**: A Framework for Designing Constraint Stores, 2007.
- No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
- No 1305 **Magnus Ingmarsson**: Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
- No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
- No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
- No 1309 **Ola Leifer**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
- No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
- No 1313 **Zhiyuan He**: System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
- No 1317 **Jonas Elmqvist**: Components, Safety Interfaces and Compositional Analysis, 2007.
- No 1320 **Håkan Sundblad**: Question Classification in Question Answering Systems, 2007.
- No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
- No 1329 **Martin Magnusson**: Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
- No 1331 **Mikael Asplund**: Restoring Consistency after Network Partitions, 2007.
- No 1332 **Martin Fransson**: Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
- No 1333 **Karin Camara**: A Visual Query Language Served by a Multi-sensor Environment, 2007.
- No 1337 **David Broman**: Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
- No 1339 **Mikhail Chalabine**: Invasive Interactive Parallelization, 2007.
- No 1351 **Susanna Nilsson**: A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
- No 1353 **Shanai Ardi**: A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
- No 1356 **Erik Kuiper**: Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
- No 1359 **Jana Rambusch**: Situated Play, 2008.
- No 1361 **Martin Karresand**: Completing the Picture - Fragments and Back Again, 2008.
- No 1363 **Per Nyblom**: Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
- No 1371 **Fredrik Lantz**: Terrain Object Recognition and Context Fusion for Decision Support, 2008.
- No 1373 **Martin Östlund**: Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
- No 1381 **Håkan Lundvall**: Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
- No 1386 **Mirko Thorstensson**: Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
- No 1387 **Bahlol Rahimi**: Implementation of Health Information Systems, 2008.
- No 1392 **Maria Holmqvist**: Word Alignment by Re-using Parallel Phrases, 2008.
- No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
- No 1401 **Annika Öhgren**: Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
- No 1410 **Rickard Holmsmark**: Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
- No 1421 **Sara Stymne**: Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
- No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
- No 1450 **Fabian Segelström**: Visualisations in Service Design, 2010.
- No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
- No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.

No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011.

No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011.

No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.

No 1485 **Loove Broms**: Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011.

FiF-a 101 **Johan Blomkvist**: Conceptualising Prototypes in Service Design, 2011.

No 1490 **Håkan Warnquist**: Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011.

No 1503 **Jakob Rosén**: Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011.

No 1504 **Usman Dastgeer**: Skeleton Programming for Heterogeneous GPU-based Systems, 2011.

No 1506 **David Landén**: Complex Task Allocation for Delegation: From Theory to Practice, 2011.

No 1507 **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.

No 1509 **Mariusz Wzorek**: Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011.

No 1510 **Piotr Rudol**: Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011.

No 1513 **Anders Carstensen**: The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011.

No 1523 **Jody Foo**: Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012.

No 1550 **Anders Fröberg**: Models and Tools for Distributed User Interface Development, 2012.

No 1558 **Dimitar Nikolov**: Optimizing Fault Tolerance for Real-Time Systems, 2012.

No 1582 **Dennis Andersson**: Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013.

No 1586 **Massimiliano Raciti**: Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013.

No 1588 **Banafsheh Khademhosseini**: Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013.

No 1589 **Amy Rankin**: Resilience in High Risk Work: Analysing Adaptive Performance, 2013.

No 1592 **Martin Sjölund**: Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013.

No 1606 **Karl Hammar**: Towards an Ontology Design Pattern Quality Model, 2013.

No 1624 **Maria Vasilevskaya**: Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013.

No 1627 **Ekhiotz Vergara**: Exploiting Energy Awareness in Mobile Communication, 2013.

No 1644 **Valentina Ivanova**: Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014.

No 1647 **Dag Sonntag**: A Study of Chain Graph Interpretations, 2014.

No 1657 **Kiril Kiryazov**: Grounding Emotion Appraisal in Autonomous Humanoids, 2014.

No 1683 **Zlatan Dragisic**: Completing the Is-a Structure in Description Logics Ontologies, 2014.

No 1688 **Erik Hansson**: Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014.