CONCURRENCY AND SYNCHRONIZATION IN THE MODERN MANY-CORE ERA: CHALLENGES AND OPPORTUNITIES

by

Juergen Ributzka

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Summer 2013

CONCURRENCY AND SYNCHRONIZATION IN THE MODERN MANY-CORE ERA: CHALLENGES AND OPPORTUNITIES

by

Juergen Ributzka

Approved: _____
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____
Babatunde Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____
James G. Richards, Ph.D.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Stephan Bohacek, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Fouad Kiamilev, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Kathleen Knobe, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

After many years on this journey, I am finally closing another big chapter in my book of life. I am excited by the new opportunities that are being presented as I prepare to open the first in a series of brand new chapters, which I will fill with new and amazing stories. Although this has been a huge step in my life I feel that my journey is on the verge of truly beginning, and that the most important and precious moments of life are still ahead of me. In the many years I have been in the USA, I have had the opportunity to meet countless people from all over the world and learn more about their different cultures. During this time all these people left a mark on my life and the sum of all these little encounters defines me and the things I have accomplished during these years. This dissertation is not only my work, but the sum of many people that have supported me in a multitude of ways.

The foundation that made this research possible was laid by Fei Chen and Yuhei Hayashi who created the initial emulation framework used during my dissertation. Special credit must be given to Yuhei, who dedicated a great deal of time to adapt the existing emulation framework to make it applicable to my research. I also owe Monty Denneau much gratitude for making his IBM Cyclops-64 architecture available to us and providing such a wonderful research vehicle.

Another great example that shines with their openness are Kathleen Knobe and Frank Schlimbach from Intel. They provided me with their Concurrent Collections software framework to jump start my research on a well-tested and supported software platform.

I was standing on the shoulders of giants and this played a big role in completing this dissertation, but people very often forget the little and small acts of kindness and help which we get every day and they are equally important for our success. I would like

to thank all the staff including Michael Davis, Karen DiStefano, Kathleen Forwood, Kjeld Krag-Jensen, Debbie Nelson, Lydia Pagnotti, Jo Ann Rucker, Wendy Scott, Deborah Whitesel, and many more.

During my time in the CAPSL research group I met many new colleagues (too many to list them all here) and also new friends. Even though I haven't worked directly with everyone, I wish to extend my gratitude to all of them, because in one way or another we all helped each other.

I also was very fortunate in the help I received from friends in reviewing and advising me in my thesis research. Thank you Asia Dowtin, Aaron Landwehr, Joseph Manzano, Sunil Shrestha, and Pamela Vovchuk for all your help.

I thank all the members who served on my committee for accepting the invitation to counsel and support me in my endeavors. To Stephan Bohacek, Fouad Kiamilev, Kathleen Knobe, and Xiaoming Li, I am thankful for your feedback, assistance, and the time you invested toward helping me to complete this degree.

My advisor, Professor Guang R. Gao, gave me full freedom in my research and work - a great privilege that not many graduate students receive and I feel honored with the trust he put in me. Over the years I also learned many of the interesting stories of his personal life and I really enjoyed the conversations we had. It is good to know that there is a safe place to come back to if sometimes things don't work out as planned, and I will never forget that kindness.

My heart is trapped between two continents. When I am in the USA my thoughts are with my family back home, and when I am at home I am missing all my friends here. Although there is a huge distance between us I take comfort in knowing that they are thinking of me and loving me from afar. It is always good to know that they are just a phone call away, which can bridge this vast ocean between us. I received all I could wish for from my parents to support me on my current journey and beyond.

DEDICATION

This dissertation is dedicated to my parents, Helmut and Ute Ributzka,
for their limitless patience, love, and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Many-core architectures are omnipresent in today's modern life. They can be found in mobile phones, tablet computers, game consoles, laptops, desktops and server systems. Although many-core systems are common in powerful mainstream systems, their core count is still in the lower tens and has not increased much over the last years. Truly massively parallel systems with core counts in the higher tens or even hundreds have only been seen so far in custom-made architectures for High Performance Computing (HPC) systems or innovative new architectures from start-up companies. Nevertheless, the core count has already reached a critical mass that shows the difficulty of increasing performance and reducing power. This requires careful orchestration of the many cores with efficient synchronization constructs such that they reduce the idle time of waiting cores and use power efficient synchronization operations.

This thesis explores the challenges that current many-core architectures face. First, it analyzes synchronization constructs not only on from a hardware perspective but also from a software stack / runtime view. Under this study it investigates the feasibility, usefulness, and tradeoffs of different synchronization mechanisms including fine-grain in-memory synchronization support in a real-world large-scale many-core chip (IBM Cyclops-64). The original Cyclops-64 architecture design is extended at the gate level to support fine-grain in-memory synchronization features and it proposes a new non-strict synchronization method. Next, it performs an in-depth study of a well-known kernel code: the wavefront computation. Several optimized versions of the kernel code are used to test the effects of different synchronization constructs using a chip emulation framework. Furthermore, it compares selected SPEC OpenMP kernel loops using these mechanisms against existing well-known software-based synchronization approaches.

In the wavefront benchmark study, the combination of fine-grain dataflow-like in-memory synchronization with the new non-strict synchronization method yields a thirty percent improvement over the best optimized traditional synchronization method provided by the original Cyclops-64 design. The SPEC OpenMP kernel loops show speedups of three to fourteen times the speed of software-based synchronization methods.

Second, this thesis introduces a new method for data availability tracking at the software layer. It enables hierarchical tiling and dynamic partitioning during runtime under a new parallel programming model and language called Intel's Concurrent Collections (CnC). A prototype of the method is implemented as a proof-of-concept in the CnC programming language framework. The results are very promising and show good efficiency compared to manual hand-tuned code.

More importantly, the new system is automatic and fully integrated into the language and runtime framework. This has several advantages over the hand-tuned code. It reduces the effort on the programmer by reducing the complexity and amount of code that has to be written. On the language side an even more fundamental change occurs. This new model allows a clean separation of the algorithm and the tuning specification. This separation of concerns is a core concept of CnC and this thesis extends CnC to support tileable dense data arrays. Another nice side-effect of this new method is the reduction of runtime calls. These runtime calls are pure overhead and are required by the language framework to ensure the correct execution of the program. Any reduction of these runtime calls is a potential scalability and performance improvement, but this effect depends on a given application.

The future of many-cores brings many challenges with it and there is not a single silver bullet to solve all the issues. This thesis highlights two opportunities to tackle the issues ahead of us. It requires a rethinking of our existing infrastructure starting at the top with powerful and new programming languages designed from the start with concurrency in mind, all the way down to dedicated hardware support that has to be carefully chosen and designed to achieve the desired results.

Chapter 1

INTRODUCTION

Many-core architectures have slowly infiltrated almost all aspects of modern life. Mostly unnoticed, they can now be found in a wide variety of devices including mobile phones, tablet-, laptop-, and desktop computers, and of course in the most powerful supercomputers. Although many-core systems have only recently become visible to the general public, they have been in use by the scientific and high performance computing community for several decades. The first many-core systems that were developed were custom made systems or research systems and their sheer size and cost prohibited home use. It was not until the late 1970s that the first single-core computers became affordable, made it into homes, and therefore became more widely available. With the introduction of the IBM PC in 1981, the era of x86 based systems began. A steady increase in the performance of single-threaded microprocessors (50-60% each year) fueled the advances for the coming decades until everything came to a sudden halt in 2005.

1.1 History of Microprocessor Architecture

The move to many-core architectures was not necessarily a voluntary change for everyone, but physical laws and the resulting limitations (which will be discussed later in detail) made it necessary to leave the successful path of highly sophisticated superscalar out-of-order architecture development and move toward many-core architectures. Today, the many-core architectures model promises to be the holy grail to solve the dilemma we are facing, but it is too early to tell for certain if this direction will be successful or if an even more radical change is necessary.

To provide a deeper understanding of why this change was necessary, we will first take a look at the history of microprocessor architecture to get a better understanding of the challenges architects faced over the past several decades and how they addressed these issues. This look back is also very interesting, since early many-core architectures have striking similarities to earlier, simpler microprocessor architectures.

For the past few decades, the number of transistors in integrated circuits has doubled about every two years. Gordon Moore observed this originally in 1965 [1] and predicted a doubling every year and that this trend would continue for at least another ten years. In 1975 [2] he revised his prediction to a doubling every two years. Later, this observation came to be referred to as Moore's Law. This trend continued not only for ten years, but for decades to come and is still true today. This predictable advance in processing technology fueled the imagination of many architects and greatly benefited microprocessor development and architectural advances.

With each reduction in feature size, transistors were improved in two aspects. First, for every new technology generation, a reduction of transistor dimension by 30% actually leads to a 50% area reduction - effectively doubling transistor density. Second, the smaller transistor also has less delay, which allows for a frequency increase of 40%. The increase in frequency by itself is already a performance boost, but the additional transistors could be put to good use too.

Initially, additional transistors were used to increase bit-level parallelism. As a result, microprocessors scaled quickly, increasing from 4 bit (Intel 4004) to 32 bit (Intel 80386DX) within 15 years. This improved integer performance by providing native support for larger integer data types, and most simple instructions like addition could be performed in just one cycle. Another way to use these additional transistors was by integrating the floating-point unit into the chip itself, instead of using a separate off-chip dedicated floating-point processor.

Looking at Figure 1.1 an interesting phenomenon can be observed: the clock rate of microprocessors is increasing at a superlinear rate that cannot be explained by the 40% transistor frequency increase for each technology generation.

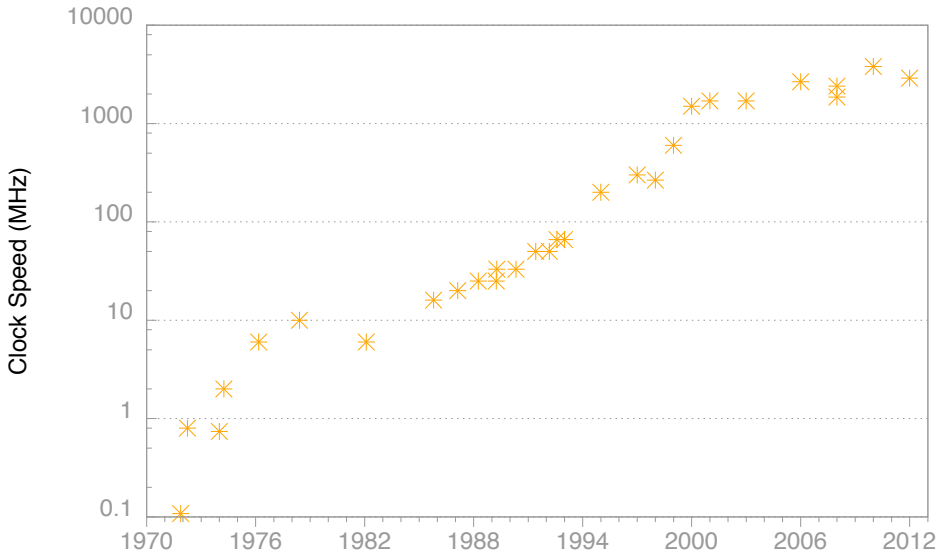


Figure 1.1: Processor Frequency

Before going into further detail let's take a step back and introduce a concept called pipelining. Pipelining can be applied to several different problems, but in this context we will concentrate on pipelining in microprocessors. A very good example to demonstrate this concept is the simple processor pipeline of a Reduced Instruction Set Computing (RISC) processor like MIPS. Without pipelining, a microprocessor would treat each instruction in turn - one at a time. The maximum frequency at which the microprocessor could operate would be limited by the slowest instruction. While processing a single instruction, different parts of the microprocessor are used over time. By using this approach most of the logic sits idle waiting for work. To better utilize the different components of a microprocessor, the lifetime of an instruction can be split up into different phases - henceforth called pipeline stages. The MIPS processor pipeline depicted in Figure 1.2 consists of five stages - Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back.

- Instruction Fetch: The first stage of the pipeline fetches the next instruction directly from memory, or more common for current microprocessors, from the

Instruction Cache (I-Cache). This assumes that the instruction can be obtained within one cycle.

- Instruction Decode: The second stage decodes the instruction, which can be done easily by combinatorial logic in one cycle. Furthermore the required registers are loaded from the register file.
- Execute: The third stage performs the actual work by executing the instruction. This applies only to single-cycle instructions and memory operations. More complicated multi-cycle instructions are passed on to a separate unit that also write back to special dedicated registers.
- Memory: During this stage the actual memory operation, based on the address computed by the previous stage, is performed. For all other operations the result from the execute stage is just forwarded to the next stage.
- Write Back: In this final stage the result is written back to the register file.

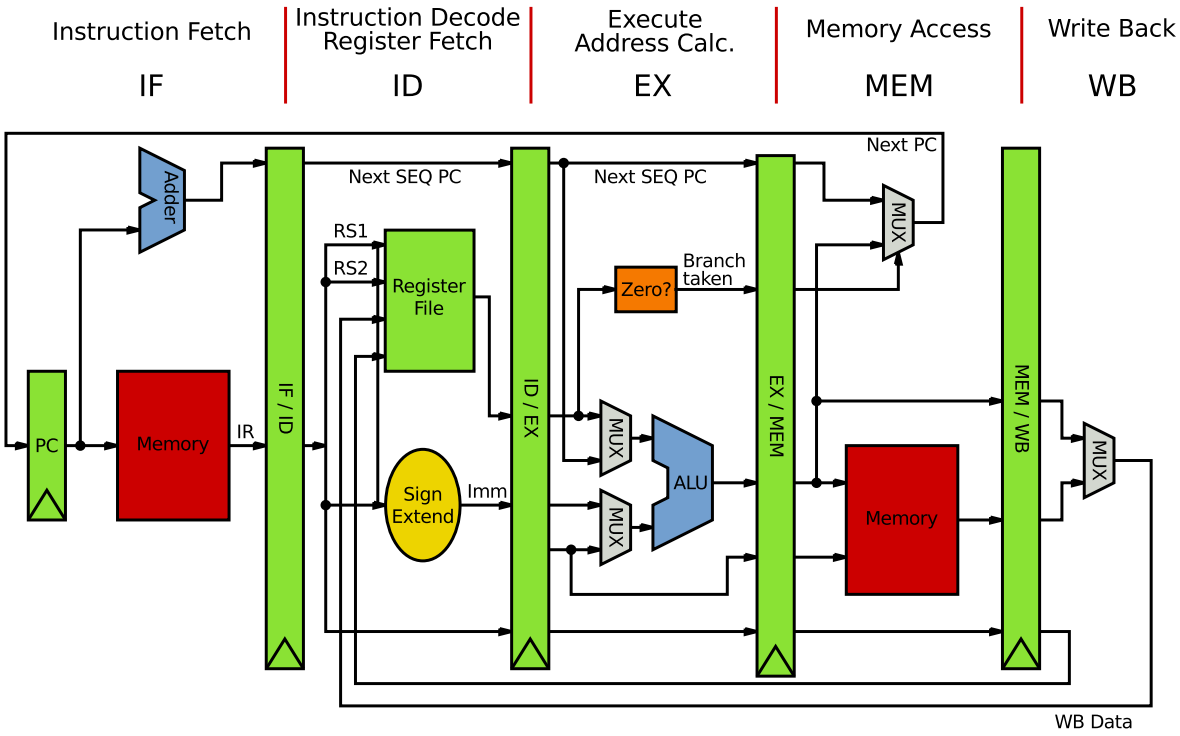


Figure 1.2: MIPS Processor Pipeline

Other architectures like the Cyclops-64 Many-Core architecture, as further explained in Chapter 3, has only a four stage pipeline. There are also architectures with

deep pipelines like the IBM POWER6 processor with 33 pipeline stages on the other side of the spectrum.

Using a pipelined architecture enables a new instruction to be placed into the pipeline every clock cycle (this would be the ideal case) and the different resources are more efficiently utilized. By breaking up the processing of every instruction into smaller stages the processor frequency can be increased. This assumes there is a perfect overlap of the used resources and there are no conflicts/hazards. If for example an instruction depends on the result of the previous instruction it would read the wrong data from the register file, because the register file would be updated afterward. This is commonly referred to as "data hazard". One way to avoid this would be to introduce "bubbles" in the pipeline to guarantee that the value has been written to the register file before its next use. Unfortunately, this would negatively affect performance. Another way would be to forward the result from the output of execution stage directly back to the input of the execution stage for the next cycle. This works only for single-cycle instructions. Memory operations require an additional cycle and in this case the "bubble" cannot be avoided. Ideally, the compiler is aware of this and tries to schedule an unrelated instruction between the load operation and its successor to avoid this "bubble".

Another source of hazards are control flow instructions - hence called control hazards. Control flow instructions, such as conditional and unconditional branches, may change the address of the instruction pointer. Depending on the architecture, the calculation of the next instruction pointer address takes at least one cycle, but could also take several cycles. As a result one or several wrong instructions might have been placed into the pipeline. The pipeline then has to be flushed/drained to evict the erroneous instructions. This can be extremely costly for deep pipelined processors (24-100 clock cycles for the Intel P4). It also wastes energy on useless instructions and may negatively affect the instruction cache too. Control intensive code can therefore reduce the benefit of pipelining. Great effort is put into preventing or reducing such hazards. For conditional branches, the processor keeps a history to predict if the branch will be taken or not. For conditional and unconditional branches, it also keeps a history of past

branch target addresses. Unconditional branches will only be mispredicted if they are not in the Branch Target Buffer (BTB) or if two branch instructions alias to the same location in the buffer. Conditional branches are more difficult to predict and several branch predictors [3] have been implemented to handle even simple repetitive patterns. Unfortunately, it is impossible to create a perfect branch predictor - that would be an oracle. For applications with rather erratic branch behavior, this approach falls apart and a huge amount of work (up to 40% [cite]) is wasted.

Some architectures support a feature called predication (ARM, Itanium, and GPUs). Predication allows the architecture to execute instructions conditionally. They are fetched and placed into the pipeline, but based on the content of a condition register they may or may not be executed. This allows the compiler to schedule both branches of a conditional statement as straight line code, interleaving instructions from both branches, without any branch instructions. Although this approach prevents costly flushes of the pipeline, it is not always beneficial and can lead to even worse performance if not used correctly.

To explain the superlinear increase in frequency one important metric needs to be explained first. The fan-out of 4 (FO4) metric is a process-independent delay metric that describes the delay of an inverter that has to drive four comparable inverters in size on its output. This allows for the fair comparison of a circuit's performance independent of its feature size. It also can help to find a lower bound for a pipeline stage. The clock rate does not only depend on the frequency at which transistors can reliably work, but also at how much work has to be performed in each pipeline stage (ignoring wire delay). The FO4 delay is a metric that allows for the comparison of the different architectures at different feature sizes. The Intel 386 in 1989 had a FO4 delay of around 80 per pipeline stage. This reduced dramatically to 11 FO4 delays for the IBM Cell Processor in 2006 [4]. A lower FO4 delay normally indicates a deeper pipeline with less logic in each pipeline stage. Deeper pipelines allow a processor to run at higher frequencies. The combination of deeper pipelines, with less logic per pipeline stage (small FO4 delay) and the increase in transistor frequency explains this

incredible improvement in microprocessor clock speed of the past several decades. It also explains why we cannot continue on this path anymore. The benefits of reducing the FO4 delay of a pipeline stage diminish because the overhead for the latches to keep and pass on the state to the next pipeline stage stays constant. Hrishikesh et al. [5] estimate the optimal logic depth to be 6 to 8 FO4 delays for floating point and integer pipelines respectively. This also would allow enough logic to implement a 64 bit adder in one pipeline stage [6]. We are already approaching the 8 FO4 delay, so there is little room left to improve the clock frequency with deeper pipelines. This will inevitably stop the superlinear improvement in microprocessor frequency and limit improvements to advances in transistor technology.

1.2 Instruction Level Parallelism Wall

The goal of every architect is to increase the performance of each new design. Performance improvements are not limited to increasing frequency; the idea is to execute as many instructions as possible. Increasing the frequency of the design is one way to achieve this. Another is to execute more than one instruction per cycle. With the increasing transistor budget, architects were able to duplicate certain structures, e.g. the Arithmetic and Logic Unit (ALU). This also required changes to other parts of the microprocessor, e.g. the register file, which requires additional read and write ports to feed more than one ALU simultaneously. There are two architectural philosophies that use this approach - Superscalar and Very Long Instruction Word (VLIW). Superscalar does not require any changes to the instruction format and is therefore backward compatible with previous processors that use the same instruction format. Superscalar microprocessors have a more complicated control logic to make certain that a set of instructions can actually be executed concurrently. Since the instruction format is parallelism-agnostic, the microprocessor has to make sure that the result is the same as it would be if the instructions would have been executed sequential. VLIW, on the other hand, uses a different instruction format that explicitly exposes the different functional units of the microprocessor. Since this format is highly architecture dependent,

backward compatibility is sometimes difficult or impossible to achieve. The addition of a new functional unit in the next version of the architecture requires a modified instruction set that provides an additional slot in the instruction format for the new functional unit. This explicit exposure of concurrency in the instruction format itself makes the processor's control logic less complex.

Regardless of which philosophy is used to take advantage of Instruction Level Parallelism (ILP), it requires a good compiler that is capable of extracting this instruction level parallelism from the application and schedule the instructions carefully to fully utilize the pipeline. The instructions are statically scheduled and the microprocessor relies heavily on the compiler to achieve good performance. In the beginning this task was easier to achieve, because the memory subsystem was more predictable, which is no longer true. Section 1.3 about the memory wall will detail this problem further. It has become rather difficult to correctly estimate the time it takes to load a given datum from memory. This complicates the scheduling for the compiler. Furthermore, due to the limited information available to the compiler, possible optimizations might be prevented and a more conservative schedule is generated that does not expose all the possible instruction parallelism available in the code.

Over time, architects added more functional units to the architecture that greatly increased the theoretical peak performance of the microprocessor, but it also became increasingly more difficult for the compiler to extract more parallelism from the application and provide a good schedule to fully utilize the pipeline. The compiler is limited to the information available to it during compile time, but the microprocessor has additional information available during runtime. In the 1990s microprocessors switched to Out-of-Order (OoO) execution. Out-of-Order Execution allows the processor to look at a set of instructions (the instruction window) and schedule all instructions based on which inputs are available for execution, independent of the order in the instruction stream. However, the results have to be committed in program order to the register file and rolled back correctly during a branch misprediction or exception.

This requires rather complex and power hungry logic. Improvements in the Out-of-Order Execution Engine reaped more instruction level parallelism from applications and increased performance, but the maximum amount of parallelism that can be extracted is limited and application specific. This effect is known as the ILP wall. Since there might not be enough parallelism in one application, the combination of two or more application should yield more instruction level parallelism. To achieve this on the hardware level, architects developed Simultaneous Multithreading (SMT) [7, 8]. This allows the operating system to schedule more than one process simultaneously on the same processor. Hardware structures that contain process specific states, such as the architectural register file, instruction pointer, etc. are replicated to support several concurrent and independent hardware threads. The instructions of the different hardware threads are scheduled simultaneously on the shared microprocessor pipeline. Doing so increases the utilization of the processor pipeline and allows long latency operations to be hidden by overlapping several threads or processes. This increases the computational bandwidth, but applications might now take more time to execute, since they have to share a single pipeline. SMT can also have negative effects on an application, because now more than one application has to share the same instruction cache, data cache, and memory subsystem.

This trend already shows that the ILP in an application is limited and the use of multithreading is required to create more instruction level parallelism.

1.3 Memory Wall

The tremendous strides in microarchitecture performance advances were not matched by memory technology. Both technologies were evolving at an exponential rate, but with different exponents. As mentioned before, microprocessor performance is increasing between 50-60% each year, while memory performance increases only at a rate of 7-10% per year [9, 10]. Figure 1.3 visualizes the widening gap between processor and memory performance. This ever growing gap made it difficult to hide the increasing memory latency with more and more instructions. With increasing transistor budgets,

architects were able to create a very fast on-chip temporary storage. This storage was a cache that automatically obtained the required data from memory when needed, or proactively based on predictors, and wrote it back when the space was needed for other data. The idea is to keep data that might be used more than once in a certain time period (temporal locality), or data that are close to each other, such as in traversal of an array (spatial locality) closer to the processor to reduce the access latency. In current microprocessors there might be more than one level of caches and the caches can be dedicated for data or instructions, or they can be unified. It is common to find a combination of both, where the first level caches are dedicated to either data or instructions and the lower and last level caches are unified to handle both.

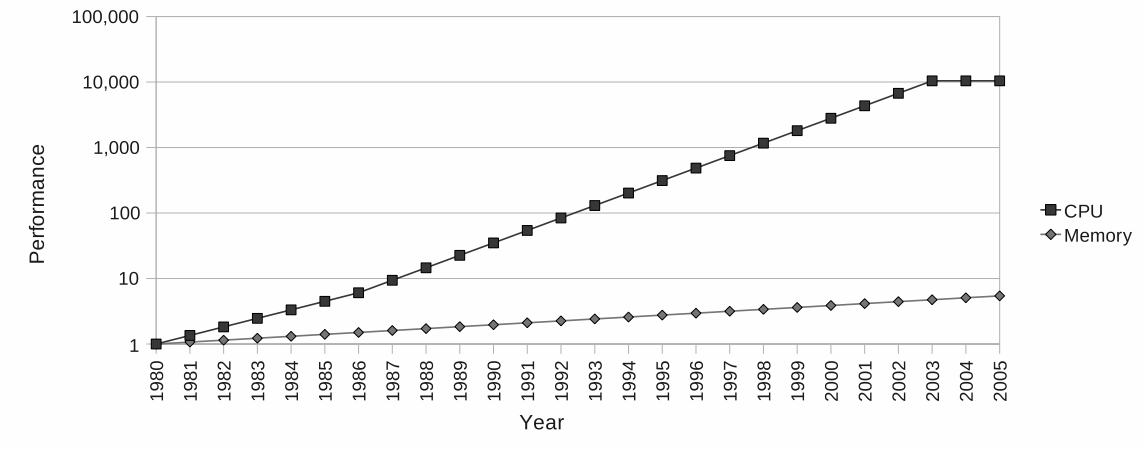


Figure 1.3: Memory Performance Gap

Although caches are transparent to the application running on the microprocessor, knowledge of their existence, configuration, and parameters are crucial for application developers and compilers to obtain maximum performance. Caches can delay the occurrence of the memory wall or even prevent it for certain applications, but other applications that cannot take advantage of caches are already limited by memory bandwidth and delay.

Data caches have become a fundamental part of every modern microprocessor and over 50% of the transistor budget is now dedicated to these memory structures.

1.4 Frequency Wall

Beginning in 2002, microprocessor clock rates started to stagnate and have not improved much since. One of the reasons is, as mentioned above, the lower limit on the logic in a single pipeline stage. As the lower limits of FO4 delay are reached, further deepening of pipelines has diminishing returns. Also, the increasing gap between microprocessor and memory performance further diminishes any gains. These aspects alone would already warrant a reduction or stagnation of frequency. If power is considered as well, then further frequency scaling becomes prohibitive.

1.5 Power Wall

With increasing transistor density, the power consumption initially did not increase because increases in frequency were compensated by voltage reductions. But this only applies to dynamic power - the power that is dissipated for switching a transistor - and this also assumes we can reduce the voltage with every technology step. Another factor that has to be considered is leakage power [11]. This power is dissipated regardless of whether the transistor is switching or not. The only way to eliminate the leakage power is to cut off the power to the circuit. Leakage power used to be a very small factor, so it could be ignored. But with transistor counts now in the billions, there are a lot of transistors doing nothing and still dissipating a huge amount of energy. Soon there will be so many transistors on a single chip that the power to run them all cannot be provided. This phenomenon coined the term dark silicon [12]. With the end of Dennard scaling [13] in 2005, dynamic power will also increase more because we cannot reduce the voltage anymore to compensate for higher frequencies. Since power is becoming the limiting factor in how many transistors can actually be turned on and used, an exploration of reduced supply voltages (also referred to as near

threshold voltages [14]) becomes interesting. The backside of this is another reduction in frequency and therefore performance as well.

1.6 Wire Delay Wall

With every new technology step transistors become smaller and faster. Wires get smaller too, but also increase resistance and capacitance - effectively they are becoming slower than logic [15]. This will have severe repercussions for future architectures. Many modern architectures use sophisticated architectural features that heavily depend on a large set of state. Accessing this state will become more costly in terms of time when the wire delay increases. To maintain the same microprocessor clock rate, the state has to become smaller, which will reduce performance. Or the state stays the same, but frequency is reduced, also leading to less performance.

1.7 The Era of Many-Core begins

There is an abundance of transistors these days, but we have to change the way how we use them. The current path we have traveled on lead to an dead end and all the constraints nudges us slowly but steady into the many-core era. More and more many-core architectures are appearing e.g. IBM Cyclops-64 [16], Tilera Tile [17, 18, 19], and Adapteva Epiphany , to just list a few.

This thesis is structured as following: Chapter 2 introduces several fine-grain synchronization constructs and a few architectures that are famous for their in-memory synchronization support. Furthermore it introduces several asynchronous execution models and runtimes that embrace the philosophy of fine-grain asynchronous synchronization. Chapter 3 gives a detailed description of the Cyclops-64 architecture, which plays a central role in this dissertation. Chapter 4 introduces the hardware based synchronization approach with a motivation example. Chapter 5 and 6 show how the Cyclops-64 architecture was modified to support fine-grain in-memory synchronization for many-core architectures on the architectural level with the Extended Synchronization State Buffer (E-SSB). Chapter 7 describes the different implementation used for

the E-SSB case study. Chapter 8 contrasts the advantages and disadvantages of the different fine-grain in-memory synchronization constructs. Chapter 9 gives a detailed introduction to the emulation system that played an important role in debugging and emulating the modified many-core architecture. Chapter 10 presents the results obtained from the emulation system for the different synchronization constructs presented in this dissertation and provides an in depth analysis of the results. Chapter 11 introduces a high level language framework, which is basis of the research presented in the following chapters. Chapter 12 looks at the problem from a different angle and investigates how synchronization could be used more efficiently on the language and runtime level. Chapter 13 and 14 presents the proposed new item collection and its implementation for the language framework to support automatic tiling for dense data arrays. Chapter 15 evaluates the performance of the new item collections. Chapter 16 presents related work and Chapter 17 concludes this dissertation and provides an outlook for possible future work.

Chapter 2

BACKGROUND

This chapter introduces some notable hardware synchronization constructs, which are the foundation of the fine-grain non-strict synchronization method presented in Chapter 4. It also gives a brief recap of fine-grain asynchronous programming and execution models that relate to the data availability framework presented in Chapter 11.

2.1 Hardware Synchronization Methods

Over the past decades there were several parallel architectures that tackled the synchronization problem in different ways. However, a vast majority of them had one thing in common - the addition of meta-data to describe the state of a datum in memory. This addition affects the operational semantics of memory operations in the system. One of the most recognizable meta-data system is called incremental structure or I-Structure for short.

2.2 I-Structures

The I-Structure was originally proposed by Arvind et al. [20] in 1981 as an extension for the functional language Id as a pure software construct. Only later on in 1987 the I-Structure was first described as a hardware construct [21] and its operational semantics were fully defined. Even though the I-Structure was described as an extension for a functional language and a dataflow-centric hardware implementation proposed, its overall operational semantics and general idea still apply to a broader set of programming languages, like C/C++ or Fortran; and hardware implementations, like stored program architectures.

The I-Structure is a special memory that has additional meta-data associated with each memory location. This meta-data indicates if a datum is **present**, **absent** or if someone is **waiting** for it. When the memory is allocated the memory controller initializes the meta-data to the **absent** state.

A read request contains the address of the memory location to be read and a tag specifying the instruction[1] waiting for the datum. When a read request arrives, the memory controller checks the requested location's state. If the state is **present**, the datum will be immediately returned. If the state is **absent** or **waiting**, then the read operation is deferred and the tag is queued in a linked list of tags in the *deferred read request area*. When the first read operation is deferred, the location's state is changed from **absent** to **waiting**.

When a write request arrives, it contains the address of the memory location to be written and the datum. If the location's state is **absent**, then the datum is written to the location and the state is changed to **present**. If the location's state is **waiting**, then the datum is send to all the instructions specified by the tag(s) in the deferred read request linked list, the datum is written to the location, and the location's state is changed to **present**. Writing the same location more than once results in an error.

2.3 M-Structure

A natural extension of the I-Structure is to relax its single assignment rule. With this in mind, the M-Structure [22] was proposed in 1991. Its operational semantics is very similar to the I-Structure's, but instead of a write operation permanently sealing a location, a read operation will reset the state to **absent**, so that the location can be reused. This allows the iterative update of the same memory location without the need to allocate a new array for each new iteration, as it would have been required with the I-Structure. In the case a write operation encounters a location's state that is **present**, an error is raised.

---

[1] This is a peculiarity of a dataflow machine. Instead of describing a register location to hold the data, a tag describes the instruction that will consume the data.

2.4 Heterogeneous Element Processor (HEP)

The Denelcor HEP computer system [23] was introduced in 1978 as a large scale scientific parallel computer representing a Multiple Instruction, Multiple Data streams (MIMD) architecture [24]. I was the first commercial multithreaded multiprocessor. The architecture was designed by Burton Smith and had a variety of unusual and interesting features. A HEP system may consist of up to 16 Process Execution Modules (PEM), up to 128 memory modules, up to four I/O cache module, and up to 4 external I/O modules connected via a switch network. Each Process Execution Module - the processor - has its own 64 bit register file with 2048 registers. In addition it also has a 64 bit constant register file with 4096 entries. Each processor also has its own separate instruction memory that can hold up to 1024k instructions. Each instruction is 64 bit wide. The processor supports 16 tasks, but only seven can be used by user programs, the others are reserved for the operating system. A task specifies the base and limit for the register files and the memory. This allows for hardware protection between tasks, but tasks can also be overlapped to work cooperatively. A processor can maintain up to 128 processes in hardware and up to 64 processes can be assigned to a single task. The other 64 processes are reserved for the operating system tasks. Each process has a corresponding process status word (PSW). This PSW does not only contain the instruction pointer, but also offset values into the register and constant memory file. These could be used for reentrant programming. Furthermore it contains additional process related information and a user trap mask. The processor has a separate control and execution pipeline/loop. The PSWs circle through the control loop in a queue and a delay is added to limit the rate. This is because the execution pipeline has eight stages and only one instruction per process is allowed in the pipeline to avoid data hazards. This also means that at least eight processes must be active at any given time to fully utilize the pipeline at 10 MIPS. The processor time-multiplexes processes every execution cycle in a round-robin fashion. This approach is called fine-grained multithreading, which is similar to simultaneous multithreading. Processor that use this kind of hardware thread scheduling are also called barrel processors. For memory

operations the scheduler function unit (SFU) sends the request over the switch network to obtain or store the data and removes the PSW from the control loop. It reinserts the PSW into the control loop once the data returned and has been written to the register file. To efficiently facilitate concurrent applications each memory location (including the register file) has additional state information. These states are full, empty, and reserved. In combination with special memory operations this could be used to facilitate fain-grain producer-consumer synchronization or critical sections for mutual exclusion. If an operation fails it circles through the control loop and is retried until it succeeds.

2.5 Tera MTA / Cray XMT

The Tera MTA architecture [25, 26], now known as Cray MTA, was derived from the Horizon architecture [27, 28, 29]. Although both architectures stem its roots from the HEP architecture, which is not a big surprise considering that Burton Smith was involved in all three of them. The Tera MTA kept several features of the HEP system and extended them. One big difference is the separation of the register file for each process, which is now called stream. Originally, in the HEP system different processes could access and share the same registers. This is no longer possible and every stream has its own set of 32 64-bit general registers, eight 64-bit target registers (used for branching), and one 64-bit stream status word that includes the instruction pointer. All 128 streams can now be used for user processes. The instruction word is still 64-bit wide, but it encodes now three instructions to increase ILP. The processor issues an instruction each clock tick from the next ready stream. The memory state bits have been extended. Not only does each 64-bit memory location have a full/empty bit, they also have now two user trap bits and one forwarding bit. Although they are called user trap bits, they are meant to be used by the language implementer to implement certain features like breakpoints, demand-driven evaluation, etc. The forward bit is used to implement automatic indirection that is transparent to the user. The memory controller will continue dereferencing until it reaches a memory location

with the forwarding bit not set. The pointer itself can be used to disable forwarding too by explicitly disabling it. The semantics of the full/empty bit has not changed. There are no data caches and memory addresses are randomized to prevent hot spots. The Cray MTA 2 system was running at 220 Mhz and had a modified Cayley graph as network topology. The Cray XMT (Eldorado) system increased the frequency to 500 Mhz and used a simpler 3D torus network topology.

The **full/empty** bit can be used for fine-grain synchronization on a producer-consumer basis like a M-Structure or for a feature called *futures*, that interprets the **full/empty** differently. In the context of this thesis only the operational semantics of the **full/empty** bit used to facilitate producer-consumer synchronization are considered. This method is also called *sync* mode.

A read request contains the address of the memory location[2]. When a read request in *sync* mode arrives, the memory controller checks if the requested location's **full/empty** bit is full (equivalent to the I/M-Structure state **present**), clears the **full/empty** bit, and returns the datum. If the location's **full/empty** bit is empty (equivalent to the I/M-Structure state **absent/waiting**), the memory controller will retry the operation for a predefined number of times. When it fails to complete the operation a trap is triggered in the processor that issued the memory operation. It is now the software's responsibility to decide how to proceed. It could retry the memory operation or suspend the issuing thread (stream).

When a write request arrives, it contains the address of the memory location to be written and the datum. While in *sync* mode, the memory controller checks if the requested location's **full/empty** bit is empty, sets the **full/empty** bit, and writes the datum. If the location's **full/empty** bit is full, the memory controller will retry the operation for a predefined number of times. When it fails to complete the operation, a trap is triggered in the processor that issued the memory operation. The runtime will

[2] Please note that the tag equivalent mechanism here is the same as used in stored program architectures (i.e. destination register and processor) unlike the dataflow tag used by the I-Structure

decide what to do with the outstanding memory operations (for example to retry it, to retire it or to delay it).

2.6 Tilera Tile Architecture

The Tilera Tile architecture is a mesh-based many-core architecture based on the Raw architecture[17]. Tilera released their first processor, called TILE64, in 2004. The microprocessor consists of 64 tiles connected via a 2D mesh network, four DDR2 memory controllers, two 10-gigabit ethernet interfaces, two PCIe interfaces, and other misc I/O. Each tile contains a microprocessor, cache, and a network switch. The microprocessor is a 32-bit in-order 5-stage pipeline 3-way VLIW processor with three functional units and a 64 entry 32-bit register file. The pipeline drives two integer arithmetic units and one load-store unit. There is no floating-point unit. The cache has separate 8 KiB level 1 instruction and data cache and a unified 64 KiB level 2 cache. A level 3 cache is emulated by the aggregation of all level 2 caches on a chip. The network switch connects the microprocessor to its four surrounding neighbors in the 2D mesh network. The mesh network has five independent networks for distinct functions. The memory dynamic network (MDN) and the tile dynamic network (TDN) are used for the memory subsystem and are dynamically routed. The MDN is used for access to the DDR2 memory controllers on a cache miss. The TDN is used to access level 2 data cache of the different tiles. The user dynamic network (UDN) and the I/O dynamic network (IDN) are mapped into the register space of the microprocessor and are dynamically routed. The UDN is used for user-level communication and the IDN for inter-tile communication and I/O subsystem access. The static network (STN) is a software routed network for communication between tiles. The chip was fabricated in 90nm technology at runs between 500 and 866 MHz.

The TILEPro64 is an improved version of the TILE64 with additional instructions, double the level 1 instruction cache to 16 KiB, doubled the L2 cache associativity, and add the coherence dynamic network (CDN) for the distributed dynamic cache. The TILEPro32 is the TILEPro64s little brother and features 32 cores per chip.

The latest version, the TILE-Gx72, follows the same principles of the mesh-architecture, but with several notable enhancements. The 72 microprocessors have been upgraded from 32-bit to 64-bit. Although they still lack a dedicated floating-point unit, the microprocessors have now a new set of instructions that can be combined to execute simple floating-point operations (addition, subtraction, and multiply) in hardware. More complicated operations (division, square root, etc) are still performed in software. The level 1 instruction and data cache have been both increased to 32 KiB and the level 2 unified cache to 256 KiB. The memory controllers have been upgraded to DDR3 with ECC support. The system-on-chip also features now dedicated hardware for network package processing and cryptography. The feature size has been reduced to 40nm and the chip is running at 1 GHz.

2.7 Fine-Grain Asynchronous Programming and Execution Models

The idea of fine-grain asynchronous programming and execution models has been around in academia for several decades, but no such models made it to industry until recently. There are primarily two reasons for this: (1) such execution models were not needed for non-HPC general purpose computing, and (2) prototype implementations of such models have tended to be more difficult to program than more conventional models. However, because of the event and rise of many-core architectures in recent years, the study of such models is seeing a resurgence.

On of the more famous fine-grain execution models is Cilk [30] which made it all the way to industry. Cilk stands out through its simplicity in use and easy to understand recursive programming model. Cilk worked well for a certain set of applications (e.g. recursion, fork-join based), but it also had its own set of limitations, such as not being able to express consumer-producer problems. Other models developed around the same time tried to be more flexible at the cost of simplicity. For instance, Earth [31], a dataflow inspired execution model, encompassed the Cilk programming model space, and beyond. Earth was the inspiration for the recent proposed codelet execution model [32, 33] and the foundation of SWARM.

There is a myriad of old and new fine-grain asynchronous execution models or programming models out there, including StreamIt [34], Cilk Plus, TBB [35], CnC [36], Chapel [37], UPC [38], Habanero [39], ParalleX [40] to just mention a few of them. So far it is not clear which execution model/programming model will win and be widely adopted, since many of them are not completely defined yet and still in flux. Although, models have started to converge with respect to certain features. This thesis will focus on Intel's Concurrent Collections (CnC) programming model and framework as a basis for a proof-of-concept implementation of the data availability query system for dense data.

Chapter 3

IBM CYCLOPS-64

At the end of 1999 IBM announced the Blue Gene project - a five-year effort to build a massive parallel supercomputer geared toward biomolecular applications such as protein folding [41]. The envisioned machine was planed to provide 1 PFLOP (10^{15} floating-point operations per second), which was 50 times the computational performance of all supercomputers available at that time. To achieve this ambitious goal the chief architect Monty Denneau designed a new cellular architecture that followed a radical new philosophy. Contrary to existing architectures with out-of-order microprocessors and large multi-level caches, the Cyclops-64 architecture followed a different path. In order to exploit parallelism and provide the required computational performance the architecture features 160 independent homogeneous cores on a single chip. To accomplish this the cores were simplified to in-order RISC microprocessors and dispensed of all data caches. Two cores share one pipelined floating-point unit that can produce a fused multiply-accumulate result every cycle. Later on an already existing architecture - the PowerPC architecture - was chosen to build the Blue Gene/L system, but Monty's architecture continued on as Blue Gene/C and then as Cyclops-64. Although the Cyclops-64 architecture has changed over the years, its origin and philosophy can be still traced back to the initial ideas of the Blue Gene project.

3.1 System Architecture

At the lowest level of the Cyclops-64 system hierarchy (see Figure 3.1) we have a single Cyclops-64 chip running at 500MHz, with a peak performance of 80 GFLOPS. Each blade (compute board) contains one Cyclops-64 chip, 1GiB of DDR2 memory, and a FPGA for the system control network. The FPGA connects also to an ethernet

interface, that is used by dedicated I/O nodes. 48 blades connect to a single midplane that provides power and the wiring for the chip interconnect. Three midplanes are placed in a single cabinet. The whole system is comprised of 96 cabinets. Overall the system features a total of 13,824 chips interconnected in a 24x24x24 3D mesh network. The aggregated theoretical peak performance of the 2,211,840 compute cores is over 1.1 PFLOPS.

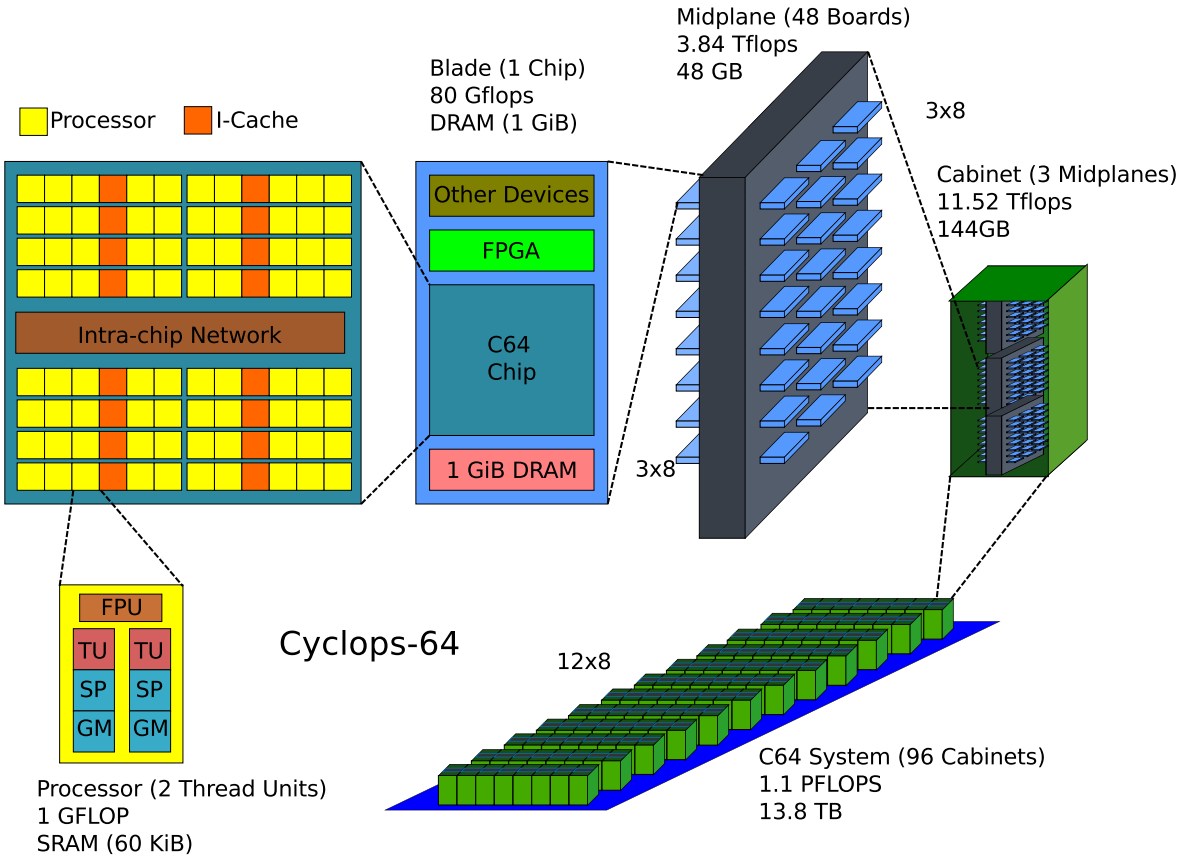


Figure 3.1: IBM Cyclops-64 System Overview

3.2 Chip Architecture

A Cyclops-64 chip (see Figure 3.2) is logically partitioned into 80 processors, containing two integer units called thread units (TUs), one floating-point unit (FPU) that is shared by the two thread units, and two 30 KiB SRAM banks (one for each

thread unit). Each thread unit has a simple 64-bit RISC microprocessor with a four stage pipeline running at 500MHz. The microprocessor is an in-order single-issue processor and uses scoreboarding for out-of-order completion. Each threat unit has its own dedicated 64-bit register file with 64 general purpose registers (GPRs), instruction pointer and special purpose registers. Ten thread units (five processors) share one instruction cache (IC) of 32 KiB. Furthermore, the chip also contains four DDR2 controllers that connect to 1 GiB of off-chip Dynamic Random-Access Memory (DRAM). The on-chip SRAM and the off-chip DRAM are error-correcting code (ECC) protected. Each chip has an integrated network switch that connects to its six neighboring chips using a 3D mesh network topology. There is also an additional interface called host interface that is used to boot-up and configure the chip and to communicate with other I/O components. All the chip components are connected through an on-chip 7-stage crossbar. In summary, the chip's crossbar interconnect possesses a total of 96 ports: 80 for the processors, four ports for the instruction caches, four ports for on-chip DDR2 memory controllers, seven ports for inter-chip communication, and one port for the host interface. The architecture has an explicit three level memory hierarchy that is fully exposed to the programmer. There are no data caches that automatically move data between the different hierarchies and the programmer is responsible to perform this task in software. The three different levels of the memory hierarchy consist of scratch-pad memory, global interleaved shared memory, and DRAM. During chip boot-up each SRAM bank of a thread unit is configured into two distinct regions. One region is configured as scratch-pad memory, the other region contributes to the global interleaved shared memory. This configuration is not required to be symmetrically and it is possible that some threat units contribute all of their memory to the global interleaved shared memory or none at all. This is particularly helpful when a few thread units or SRAM banks are faulty, but the rest of the chip is still working fine. In this case the bad components can be mapped out and the chip can be still used. In the default configuration 15 KiB is assigned to scratch-pad memory and the other 15 KiB contribute to the global interleaved shared memory of 2,400 KiB. The DRAM

and the global interleaved shared memory are interleaved at a 64 byte boundary. A thread unit has direct, low-latency access to its own scratch-pad memory; although the scratch-pad memory of all other thread units can still be accessed through the crossbar. Global interleaved shared memory has always to be accessed through the crossbar. Accesses through the crossbar guarantees sequential consistency for the global interleaved shared memory and the DRAM, but not for the scratch-pad memory if the direct link is used. There are no segments, paging, or virtual memory support. Although there is minimal memory protection support to make a configurable region of memory only accessible in supervisor/kernel mode. A massive parallel architecture requires special synchronization support to allow for efficient orchestration of the several cores on chip. One important feature of this architecture is the hardware synchronization support for fast barriers. Another very important feature is the support of atomic in-memory operations provided by all 160 SRAM memory controllers and all four DDR2 memory controllers. This means that every memory controller has a small Processor-In-Memory (PIM) to perform simple arithmetic operation atomically in memory.

3.3 Microarchitecture

The microarchitecture of a single core is a simple in-order four stage pipeline. In the first stage - the instruction decode stage - the next instruction is fetched from the Prefetch Instruction Buffer (PIB) and decoded. Each core has its own little PIB that holds two sets of 16 instructions. Every instruction is 32-bit wide and can be decoded in a single clock cycle. In the next stage - the register read stage - the requested registers are read from the register file. Most instruction have one or two source registers and the register file has two read ports to serve most instructions in one cycle. Only a few instructions with three source registers like the fused multiply-add floating-point instruction may need an additional cycle to obtain the third source register. The floating-point unit has a register cache of 5 registers to mitigate this one cycle penalty, by using the cached register. This requires careful scheduling and register allocation to take advantage of this feature. In the third stage - the execute

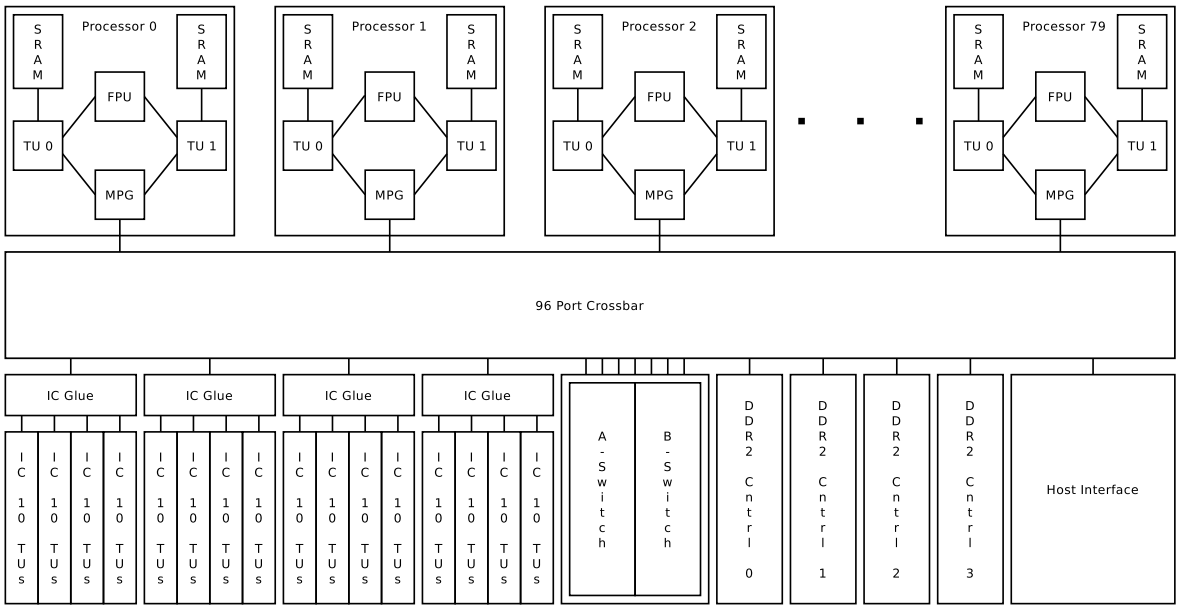


Figure 3.2: IBM Cyclops-64 (C64) Many-Core Architecture: The architecture consists of 80 processors (Processor 0 -79). Each processor has two Thread Units (TUs) called TU 0 and TU 1. Both share one Floating-Point Unit (FPU) and one crossbar port (MPG). Each TU is connected to a SRAM bank, which can be accessed by all other TUs via the crossbar. Ten TUs share one Instruction Cache (IC). The system has four on-chip DDR2 memory controllers to access off-chip memory. The A-Switch is used to connect to the six surrounding neighbors in a 3D-mesh network.

stage - most instructions are performed within one cycle. A few special operations like population count or floating-point comparison require two cycles. Floating-point operations and integer multiplication operations are dispatched to the shared floating-point unit. A Least-Recently-Used (LRU) schema is employed to decide which threat unit is allowed to dispatch the instruction or needs to stall. For memory operations the address is calculated and passed on to the storage interface that will route it to the local memory or to the crossbar. Long latency operations such as memory loads and floating-point operations set the scoreboard bit for the result register. The instruction that depends on a register that is not available yet stalls the pipeline and all instructions that follow. Careful scheduling is required by the compiler to hide these latencies with other instructions. In the last stage - the write back stage - the register file is updated with the new value from the ALU (if applicable). The register file has two write ports that have to be shared between the fix-point unit (ALU), floating-point unit, and load return from memory.

Chapter 4

FINE-GRAIN NON-STRICT SYNCHRONIZATION IN HARDWARE

Many-core architectures are on the rise and an increasing number of applications are modified to take advantage of these new architectures. A certain class of applications is very easy to parallelize, because the parallel computations performed by the applications have no dependence on each other. This small class of applications is often referred to as "embarrassingly parallel applications". Unfortunately, most applications are slightly more complicated and therefore more difficult to parallelize. It is a very common problem that once a programmer tries to split up the work, so that it can be executed on several cores simultaneously, data dependencies have to be identified and taken care of. This task was originally handled by the microprocessor. Even if it was executing instructions out-of-order and concurrently, the control logic was responsible to identify and obey these dependencies in the program. Now, by manually splitting up the application, the programmer has to perform this task. Additional synchronization constructs are now required to ensure the correct execution of a program because the different cores do not work in a lockstep fashion. Each core is independent of every other core and complex system interactions make it impossible to predict the behavior statically during compile time.

4.1 Motivation Example

To better illustrate the problem of concurrency and synchronization on many-core architectures, a simple microbenchmark - the wavefront computation - was chosen. The C code of the kernel is shown in Figure 4.1. First, the algorithm initializes the first column and the bottom row of a 2D array. Next, the remaining elements of the 2D

array are calculated based on the previously determined values from the left, bottom-left and bottom element. This forms a wavefront computation from the bottom-left corner to the top-right corner as shown in Figure 4.2.

```
1 for(i=1; i<N; ++i) {
2   for(j=1; j<N; ++j) {
3     a[i][j] = ( a[i-1][j-1] +
4                 a[i-1][j  ] +
5                 a[i  ][j-1]
6               ) / 3;
7   }
8 }
```

Figure 4.1: Wavefront Computation (C-Code)

Due to the dependence of an element on its previously computed neighbors, parallel versions of the wavefront kernel require synchronization constructs to ensure correctness. Nevertheless, it is still possible to exploit this kernel's parallelism to be executed on a many-core architecture. One possible approach would be to distribute the rows (or a contiguous set of rows) across the available processors on the chip statically in a round-robin fashion and enforce data dependencies via synchronization constructs. However, the choice and the available hardware support for a particular synchronization construct can greatly affect the performance and the scalability of the parallel implementation. To investigate the effects of different synchronization constructs, and in particular fine-grain synchronization, the wavefront computation kernel was implemented with five different synchronization constructs. An explanation of the different implementations is provided in Chapter 7 and a more in-depth analysis of the results can be found in Chapter 10.

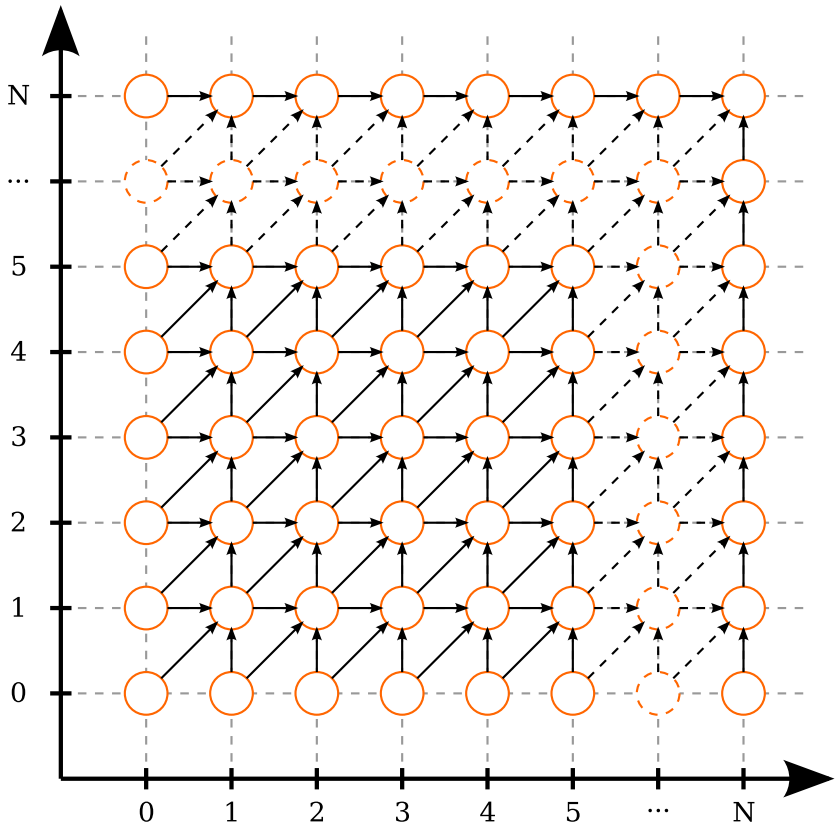


Figure 4.2: Wavefront Computation Dependency Illustration

4.2 Problem Formulation

Many-core architectures usually provide synchronization constructs directly in hardware or other hardware primitives that allow the creation of a variety of synchronization constructs in software. A very common synchronization construct is the barrier, that is usually implemented in software and provided as a library call. This construct has not a neglectable overhead that needs to be considered when partitioning the application to find a good ratio of useful work to synchronization overhead. Other synchronization constructs like signal/wait, locks, or mutexes are also implemented with simpler hardware primitives. Fine-grain synchronization on the other side requires dedicated hardware support, which is not supported by the majority of many-core architectures. Therefore, this thesis not only focuses on the performance and scalability of the different synchronization constructs, but also on the effort and cost of fine-grain synchronization support. The following questions highlight the objectives this thesis tries to answer:

How difficult is it to implement and support non-strict fine-grain synchronization?

New architectural features can be simulated and tested quickly using functional-accurate simulators, but the real complexity and timing is often misunderstood or underestimated. To determine the complexity of fine-grain synchronization constructs, an implementation at the hardware description level (HDL) of a real many-core architecture is performed. Chapter 6 gives a more detailed description of the changes that were necessary to support fine-grain synchronization in the Cyclops-64 many-core architecture.

What are the implications on used chip real estate?

The real hardware cost of a new architectural feature can be estimated to a certain extent, but the final resource usage is unknown until an actual implementation

has been performed. Section 6.1 discusses and describes the additional hardware resources, which are required to support fine-grain synchronization, and how these results are obtained.

What are the performance gains of non-strict fine-grain synchronization?

The effort and cost of adding a new architectural feature has to be validated and justified. In the case of the non-strict fine-grain synchronization construct, a substantial performance increase is expected. Otherwise, it may be more useful to use chip real estate for other features or even more cores. Chapter 10 compares and contrast fine-grain synchronization with other already existing synchronization constructs of the Cyclops-64 many-core architecture.

How to ensure the correctness of the implementation and the given performance prediction with a very high degree of confidence?

The validation of new features and their true performance is difficult to measure with software simulators only. Software simulators may be cycle accurate, but they are also slow and not useful to validate a full chip. Others might be fast, but sacrifice accuracy. Chapter 9 describes the emulation system and how it is used to obtain cycle-accurate performance results of the whole chip with a very high degree of confidence.

4.3 Extended Synchronization State Buffer (E-SSB): An Overview

The Extended Synchronization State Buffer (E-SSB) is a new fine-grain non-strict synchronization method that is not part of the Cyclops-64 architecture. It was inspired by the original Synchronization State Buffer from Zhu et al. [42]. Section 4.4 takes first a look at the original Synchronization State Buffer (SSB) and its semantics. Then Chapter 5 introduces the semantics of the Extended Synchronization State Buffer (E-SSB). Chapter 6 gives a detailed explanation of the implementation in the Cyclops-64 many-core architecture.

4.4 SSB: A Recap

The Synchronization State Buffer (SSB) proposed by Zhu et al. [42] is based on the observation that in any synchronized program only a small number of synchronized variables are needed at any point in time. This means that a small buffer (added to each memory controller) is sufficient to keep the synchronization meta-data of these variables. This reduces the memory overhead of keeping extra bits for each memory location in the system compared to other solutions [43]. Moreover, this buffer could store additional meta-data for a specific datum to enable features such as invisible indirection (pointer forwarding) and debugging/tracing capabilities.

This thesis will only describe the usage of the meta-data as full/empty bits in the context of producer-consumer synchronization. The information saved in a SSB entry is implementation dependent, but it requires at least four parts in the original SSB: (1) a state field to indicate the current synchronization mode; (2) a counter field; (3) a thread identifier field; and (4) an address field to indicate the memory address to which the entry applies. The counter field is used by other SSB instructions as actual counter, but they are not the focus of this paper. Single-Writer-Single-Reader Mode 2 uses the counter field indirectly to encode additional state information.

The original SSB design had two different producer-consumer modes, which are also called Single-Writer-Single-Reader (SWSR). The first mode employs a busy-wait approach for the reader until the data is ready. The second mode utilizes the sleep-wakeup feature of the architecture to reduce crossbar traffic. The operational semantics for the SSB synchronization constructs are described as follows:

SWSR Mode 1: Busy-Wait

A read request contains the address of the memory location to be read and a tag specifying the thread identifier (TID) and register location waiting for the datum. When a read request arrives the memory controller checks if the requested location has an entry in the SSB. If there is a corresponding entry, then the entry is removed, the datum and the status **SUCCESS** are returned. If there is no corresponding entry,

then the status **FAIL** is returned and the read operation has to be retried by the programmer.

When a write request arrives it contains the address of the memory location to be written, the datum, and a tag specifying the TID and register location waiting for the return code. If there is no entry in the SSB for the specified location, then the datum is written to the location, an entry is created with the state **SWSR 1**, and the status **SUCCESS** is returned. If there is already an entry for the specified location, then an interrupt is raised. The corresponding state diagram is shown in Figure 4.3.

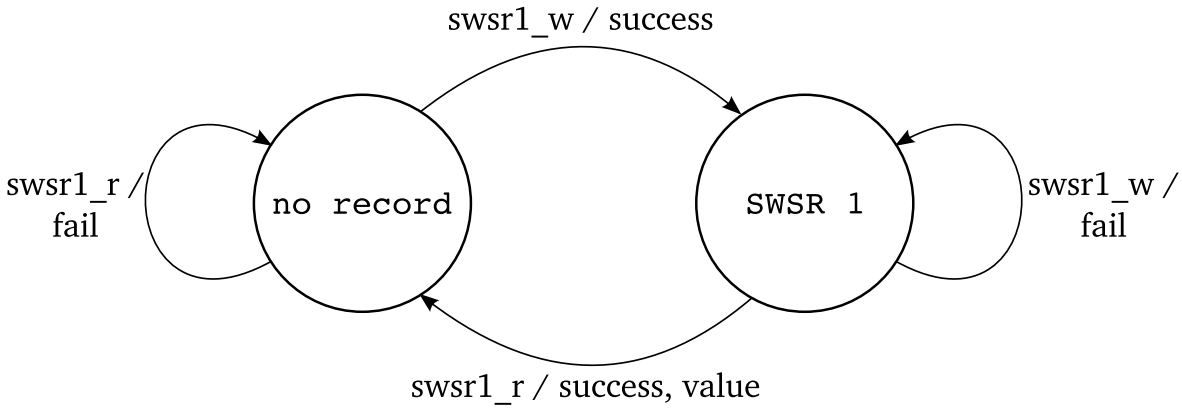


Figure 4.3: SSB 1: Busy-Wait

SWSR 2: Sleep-Wakeup

A read request contains the address of the memory location to be read and a tag specifying the thread identifier (TID) and register location waiting for the datum. When a read request arrives, the memory controller checks if the requested location has an entry in the SSB. If there is a corresponding entry, then the entry is removed, the datum and the status **SUCCESS** are returned. If there is no corresponding entry, then an entry is created with the state **SWSR 2**, and the status **WAIT** is returned. The programmer has to check for this return code and issue a sleep instruction.

When a write request arrives it contains the address of the memory location to be written, the datum, and a tag specifying the TID and register location waiting

for the return code. If there is no entry in the SSB for the specified location, then the datum is written to the location, an entry is created with the state **SWSR 2**, and the status **SUCCESS** is returned. If there is already an entry for the specified location and the state is **WAITING**, then the state is updated to **AVAILABLE** and the TID of the waiting thread is returned with the status indicating that there is a waiting thread. The programmer has to check for this return code and wakeup the thread specified by the returned TID. If there is already an entry and the state is not **WAITING**, then an interrupt is raised. The corresponding state diagram is shown in Figure 4.4.

In the event that the buffer is full and a synchronization operation tries to add a new entry, then an interrupt is generated and the software runtime will take control of the buffer. There is no automatic eviction of entries and flush to memory as a cache would do.

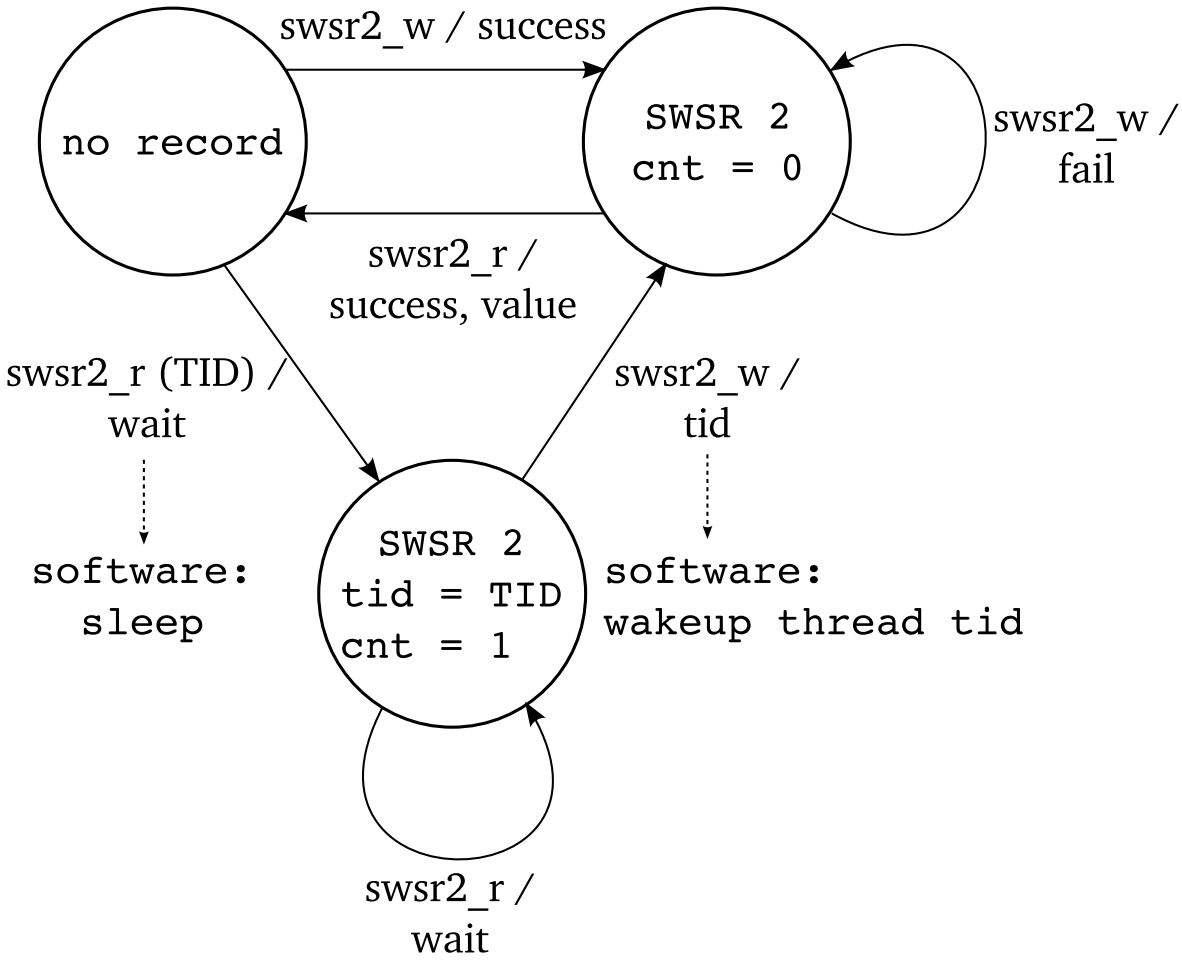


Figure 4.4: SSB 2: Sleep-Wakeup

Chapter 5

DESIGN OF THE EXTENDED SYNCHRONIZATION STATE BUFFER (E-SSB)

This section explains the design principles for non-strict fine-grain synchronization and its operational semantics. The original SSB design is extended with non-strict fine-grain synchronization. The major goal in designing the Extended Synchronization State Buffer (E-SSB) is to improve programmability and ease-of-scheduling for the compiler. The major interest are the Single-Writer-Single-Reader (SWSR) synchronization operations. A third mode, which further reduces the overhead of the synchronization operation with little additional hardware cost and non-strict behavior, is added. For the remainder of this thesis these three different modes are referred to as E-SSB 1, E-SSB 2 and E-SSB 3, respectively. Furthermore, all modes support now several data sizes: byte (1 byte), half word (2 bytes), word (4 bytes) and double word (8 bytes); and signedness (signed and unsigned) of memory operations. To support these new features an E-SSB entry is extended with the following fields: (5) register identifier; (6) size; and (7) signedness.

The operational semantics of the non-strict synchronization is defined as follows:

E-SSB 3: Non-Strict

A read request contains the address of the memory location to be read and a tag specifying the thread identifier (TID) and register location waiting for the datum. When a read request arrives the memory controller checks if the requested location has an entry in the E-SSB. If there is a corresponding entry the entry is removed and the datum is returned. If there is no corresponding entry, then an entry is created containing the state **SWSR 3**, the TID, and register location.

When a write request arrives it contains the address of the memory location to be written and the datum. If there is no entry in the E-SSB for the specified location, then the datum is written to the location and an entry is created with the state **SWSR 3**. If there is already an entry for the specified location and the state is **WAITING**, then the datum is written and also returned to the TID and register location of the waiting thread at the same time. Finally, the entry is removed from the E-SSB. If there is already an entry and the state is not **WAITING**, then an interrupt is raised.

Under this mode, the synchronization memory operations appear as normal load and store operations to the processor. The processor only stalls when a dependency is found between the synchronized operation and another operation. The corresponding state diagram is shown in Figure 5.1.

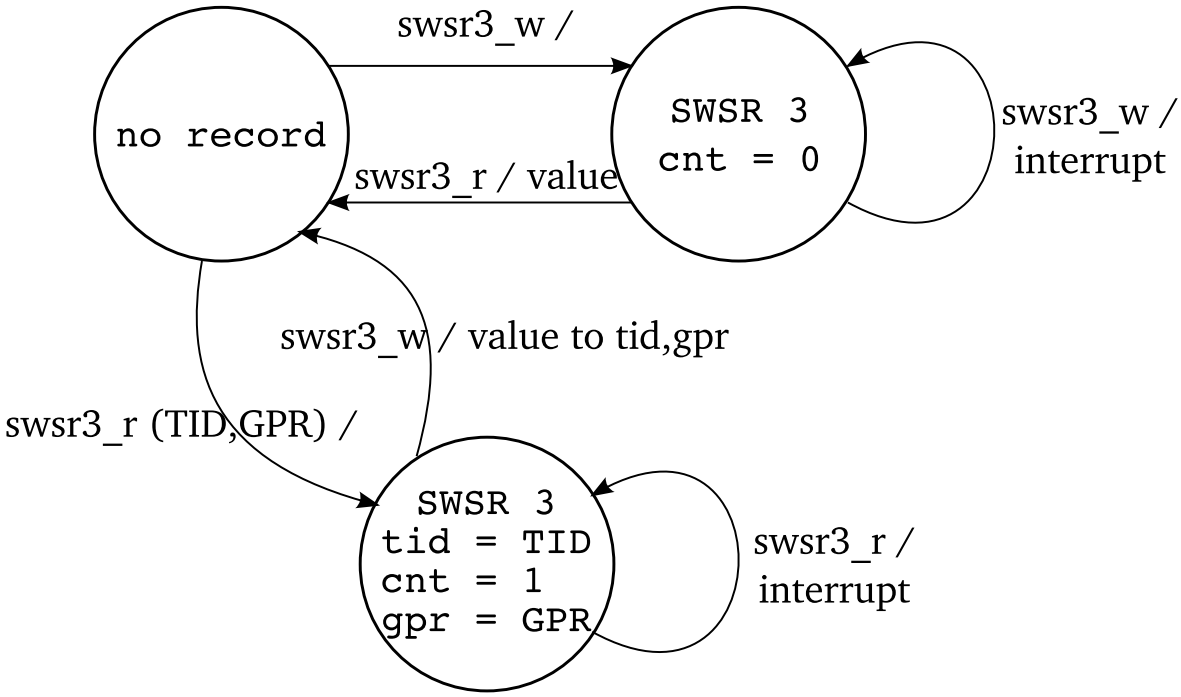


Figure 5.1: E-SSB 3: Non-Strict

Chapter 6

IMPLEMENTATION OF THE EXTENDED SYNCHRONIZATION STATE BUFFER (E-SSB)

This section describes the required architectural changes to implement non-strict fine-grain synchronization in the Cyclops-64 many-core architecture. The Extended Synchronization State Buffer (E-SSB) requires changes mainly in the Thread Unit (TU), because all the required logic related to the on-chip SRAM memory interface is located there. In particular, changes are required on the instruction decoder to support the new E-SSB instructions, the execution unit for interrupt handling and event counting, and the storage interface for routing the new memory operations and the actual implementation of the logic and buffer of the E-SSB. Another module, the crossbar interface, which is shared by two thread units, has to be adapted to support the new E-SSB crossbar packages, but changes to the crossbar itself are not required.

The existing design allows for an easy extension of the instruction decoder to support the new E-SSB instructions. Fortunately the opcode space is not completely exhausted and has enough space left to accommodate a variety of E-SSB instructions. The Cyclops-64 instruction format has a fixed size of 32-bit. The first four most significant bits are used for the primary opcode. The primary opcode could only encode 16 instructions by itself. Fortunately the majority of the instructions only use one or two source registers and at most one result register leaving enough space for an additional opcode field called extended opcode. The combination of the primary and extended opcode fields allow the encoding of a myriad of instructions with a very simple decoding logic. As a matter of fact almost all instructions are encoded using the same primary opcode, but use an additional nine-bit extended opcode field to specify the actual instruction. Memory operations, branch instructions, and instructions with

immediate values require more space for the immediate field and do not have space for an extended opcode field. They are encoded by just using the primary opcode field. The four remaining and unused primary opcodes are used to encode the various E-SSB instructions.

A new instruction format is created that allows the encoding of all instructions mentioned in the original SSB work and the new E-SSB instructions. In a real implementation not all of these instruction would actually be implemented, but for a fair comparison all instructions are implemented at the logic level. The new instruction format accommodates the primary opcode (OP), the return register (RT), the address register (RA), the value register (RB), the E-SSB opcode (EE), the size (Sz) and the signedness (S). The instruction format and the size of each field is shown in Table 6.1.

Table 6.1: Extended Synchronization State Buffer (E-SSB) Instruction Format

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signedness	unused
4	6	6	6	6	2	1	1
OP	RT	RA	RB	EE	Sz	S	0

The EE field is used to encode the actual E-SSB operation. A list of the encoded E-SSB operations is shown in Table 6.2.

The Single-Writer-Single-Reader Mode 3 instructions are also redundantly encoded using the already existing instruction format for memory operations as shown in Table 6.3.

This way they could be easily used as drop-in replacement for normal load and store instruction with no changes to the surrounding code and easily be scheduled without any restrictions. All primary opcodes with the newly added E-SSB instructions are shown in Table 6.4.

A full list of the newly added E-SSB instructions, their encoding, and their semantic description can be found in Appendix A.

Table 6.2: E-SSB Opcodes

EE	E-SSB Operation	Description
0	RLock	Read Lock
1	WLock	Write Lock
2	UnLock	Unlock
3	SWSR1_R	Single-Writer-Single-Reader Mode 1 Read
4	SWSR1_W	Single-Writer-Single-Reader Mode 1 Write
5	SWSR2_R	Single-Writer-Single-Reader Mode 2 Read
6	SWSR2_W	Single-Writer-Single-Reader Mode 2 Write
7	SWSR3_R	Single-Writer-Single-Reader Mode 3 Read
8	SWSR3_W	Single-Writer-Single-Reader Mode 3 Write
9-63	reserved	n/a

Table 6.3: Load/Store Instruction Format

Primary Opcode	Target/Soucre Register	Address Register	Size	Offset
4	6	6	2	14
OP	RT/RS	RA	Sz	D

Table 6.4: Primary Opcodes

OP	Operation	Description
0	E-SSB	E-SSB (uses E-SSB opcodes)
1	LD	Signed Load
2	ST	Store
3	LDU	Unsigned Load
4	BCC	Branch on Condition
5	BAL	Branch and Link Register
6	LDS	Signed Load Synchronized
7	STS	Store Synchronized
8	LDUS	Unsigned Load Synchronized
9	XORI	XOR Immediate
10	ANDI	AND Immediate
11	ORI	OR Immediate
12	CMPI/TRAPI	Compare or Trap Immediate
13	ADDI	Addition Immediate
14	SHORI	Shift left 16 then OR Immediate
15	EXT	all other instructions (uses extended opcodes)

Some of the original SSB instructions require more than one result register. One register is required for the return code and another one for the data. Due to restrictions in the instruction format, crossbar package format, and the register file, the result register and implicitly the following register are used as bundled result registers. For example, the E-SSB instruction **swsr1_rd RT,RA** reads a signed double word value from the address specified in register **RA**. The return code is written to register **RT** and the datum is written to register **RT+1**. The write-back register is selected to be the next register after the return-code register in the register file. The MIPS32 architecture uses a similar approach by pairing two 32-bit floating-point registers for double precision floating-point operations. Although this approach is limited to predefined even-odd register pairs only. The scoreboard of the Cyclops-64 architecture already supports the setting of a vector of bits for its load multiple instructions, so adding this behavior could be easily achieved by leveraging the existing infrastructure of the Cyclops-64 architecture. Crossbar return packages that encode both, return code and

data, require special treatment in the storage interface. A similar special treatment is already applied to load multiple crossbar return packages and a similar treatment was implemented analogous to the existing load multiple return filter.

The execution unit performs the the effective address calculation, which is also analogous to the existing memory operations. In addition the performance counters were extended to count events concerning E-SSB instructions, such as the number of issued E-SSB load and store instructions and the number of failing E-SSB load instructions (E-SSB 1 and 2). E-SSB 3 instructions cannot fail (but raise interrupts if used incorrectly), since they will wait in the memory controller. The interrupt handler is also extended to support the new E-SSB interrupt to expose the invalid use of E-SSB instructions in faulty programs.

The majority of the changes are required in the Storage Interface (SI) of the Thread Unit (TU), because it contains the E-SSB module. This is the actual buffer for the meta-data and the associated control logic. The Storage Interface orchestrates the data routing between different requests coming from the network (crossbar), Thread Unit, and the E-SSB module and the responses coming from the network (crossbar), and the memory controller (SRAM memory controller). The encoding of the crossbar package is adjusted to also support the additional E-SSB memory instructions, but this was done within the already existing encoding space and the size of the crossbar package has not been altered.

The actual implementation of the meta-data buffer is a direct mapped 16-entry 8-way associative buffer and is 47 bits wide for each entry (see Table 6.5). The required fields for an E-SSB entry in this architecture are: state (4 bits), counter (8 bits), address (15 bits), processor id (7 bits), thread id (3 bits), register id (6 bits), size (2 bits), signedness (1 bit), and bits for implementation dependent features (in this case one bit to identify memory operations originating from the local thread unit). The state field is further specified in Table 6.6.

The state field has not been fully utilized, because not all original SSB operations like Single-Writer-Multiple-Reader are implemented. The counter field is only

Table 6.5: E-SSB Entry

Field	State	Counter	Address	PID	TID	GPR	Size	Signedness	Local
Size (bit)	4	8	15	7	3	6	2	1	1

Table 6.6: E-SSB State Encoding

	State	Description
0	Invalid	Free Entry
1	RLOCK	Read Lock
2	WLOCK	Write Lock
3	WRLOCK	Write-Recursive Lock
4	SWSR1	Single-Writer-Single-Reader Mode 1
5	SWSR2	Single-Writer-Single-Reader Mode 2
6	SWSR3	Single-Writer-Single-Reader Mode 3
7-15	n/a	reserved

used for the reader-writer locks and to indicate if the read or write arrived first for the Single-Writer-Single-Reader (SWSR) operations (as described above). If only the SWSR operations would have been implemented, then the counter field could have been removed and additional states could have been added to distinguish the different cases for the SWSR modes. The address field only requires 15 bits, because at this stage it is already known which SRAM block needs to be addressed and 32KiB address space is therefore sufficient. Internally the combined processor- and thread-identifier are represented as a 10-bit value and this implementation follows this representation. The register identifier stores the register location (64 registers per thread unit) to which the value has to be returned to. This is required for the E-SSB Mode 3, when the return of a load is delayed until the store arrives. The size and the signedness of the memory operation is stored to make sure only matching pairs of load-store and lock/unlock are performed. The local bit is an implementation specific detail to identify if the operation

came from the local thread unit or from the network. Depending on the bit value the data is returned directly to the thread unit or directed to the network (crossbar). The E-SSB buffer is created generously large with 128 entries. This allows all experiments to be performed without the need to fall back to a software-based approach, which is not part of this research. The E-SSB module creates special network return packages to accommodate support for E-SSB return codes, interrupts and performance counter events. The interrupt is always raised in the thread unit that issues the memory operation and not in the thread unit where the E-SSB is located. The format of the E-SSB return packages is shown in Table 6.7. **TrCode** species the type of package, which is in this case is E-SSB Return. **Int** is used to raise an E-SSB interrupt. **Sync** is used by the performance counters. It indicates if an E-SSB load operation was successful or not (this only applies to E-SSB Mode 1 and 2 operations). **E-SSB Code** is the return code from the E-SSB module. It indicates success, failure, or if another thread unit is waiting. In the case of an interrupt the field indicates the interrupt reason. Possible reasons for interrupts are faulty programs with size or state violations of the issued operation or non-faulty programs that just filled up the buffer. The **E-SSB Code** is sign-extended to 64-bit and written to the register specified in the **GPR** field. **PID** and **TID** encode the destination and are used by the crossbar for routing. **Error** has the same behavior as for normal memory operations and raises an External interrupt. This normally happens when a user level store tries to access protected memory or a load/store access is outside the valid memory space. The content of the **Data** field is written to register GPR+1.

Table 6.7: E-SSB Return Package

Field	TrCode	Int	Sync	E-SSB Code	PID	TID	Error	GPR	Data
Size (bit)	6	1	1	3	7	3	1	6	64

At last a few changes are required on the crossbar interface. The crossbar

interface arbitrates the network access of two thread units that share a single crossbar port. There are two virtual networks - one for memory load and store request and another one for the load returns. The first type has a buffer in the crossbar interface, because a thread unit and in turn the memory controller might already be busy with a previous network request or a local request from the thread unit. The second type had originally no buffer, because a thread unit is designed to handle a load return request every cycle. With the introduction of the E-SSB return package that writes two registers this is no longer true and a second buffer of the same size is added to the crossbar interface to compensate for this. This buffer is added for pure functional reasons to make the original SSB instructions work. This buffer is not required by the new E-SSB Mode 3 instruction and an actual implementation of the E-SSB without E-SSB Mode 1 and 2 instructions would not need it. Therefore further investigation to find an optimized size for the buffer or to find an alternative solution, which would not require the buffer at all, were not performed. Minor additional changes are required to process the new E-SSB operations network packages, which is analogous to the handling of atomic memory operations network packages.

In summary the existing architecture is surprisingly extension friendly and most changes could be easily integrated by leveraging already existing system components.

6.1 Logic Resource Usage of the Extended Synchronization State Buffer

New architectural features may sometimes be implemented very easily, but the associated hardware cost can be overwhelming and may not be feasible to implement in hardware. A comparison of the Cyclops-64 design with and without E-SSB is performed. The HDL code is converted to VHDL and synthesized with the design compiler, using the generic technology independent libraries (GTECH) to generate a VHDL netlist. Then a tool is used to analyze the VHDL netlist and calculate the number of each design primitive. The design primitives reported for this study are the following basic logic elements: NOT, AND, OR, XOR, Flip-Flops (FF), and SRAM. An exact

gate number cannot be given, because this depends on the specific component libraries of the semiconductor foundry.

Table 6.8: Logic Resource Usage of the Cyclops-64 Architecture

Design Primitive	Original Design	Design with E-SSB	Increase (%)
NOT	6,946,100	7,364,740	6.03%
AND	10,924,586	11,779,946	7.83%
OR	5,812,398	6,257,358	7.66%
XOR	1,171,951	1,200,671	2.45%
FF	2,140,299	2,350,619 (+76,000)	6.28 (9.83)%
RAM(bit)	50,318,560	51,260,640	1.87%

As mentioned in Section 4.4 the implementation of the first two Single-Writer-Single-Reader Modes (E-SSB Mode 1 and E-SSB Mode 2) require additional buffers in the crossbar interface, which is solely responsible for an increase of 76,000 FF in the whole system. The first two modes are only implemented to have a fair comparison for benchmarking. In the final architecture it would not be necessary to implement all three modes and these additional FF would not be required. They are still listed in parentheses in Table 6.8 for completeness to represent the current design. The increase in combinatorial logic and Flip-Flops is moderate around 6-7%. The more costly resource - the on-chip SRAM - has only increased by less than 2%. This shows that the idea of E-SSB is a feasible solution that actually can be implemented in hardware.

Chapter 7

E-SSB CASE STUDY: WAVEFRONT COMPUTATION

This chapter details the wavefront microbenchmark, as introduced in section 4.1, implementation for the Cyclops-64 many-core architecture using the following synchronization constructs: barrier, signal-wait, and fine-grain in-memory synchronization. The different implementations are described in the following subsections.

7.1 Wavefront Computation with Barriers

A well-known coarse-grain synchronization construct is the barrier. A barrier enforces order on the issuing of memory operations and thread execution. Barriers, even if implemented or partially supported in hardware, can incur additional overhead, which needs to be considered when parallelizing an application. One way to reduce the synchronization overhead is to use a blocking approach. The 2D array is divided into blocks and each row of blocks is processed by one thread. Threads are statically assigned in a round-robin fashion to rows. By increasing the block size, the overhead of the barrier can be mitigated, but it also reduces parallelism. The barrier synchronization is considered a coarse-grain synchronization construct because it synchronizes all threads. The work allocated to each thread is equal (except for the corner cases), but other unpredictable side-effects, like crossbar congestion, can produce a variation in execution time for each thread. This means that if a single thread falls behind, all threads must wait for this one thread to complete, even though the wait for certain threads may be wasteful (i.e. there are no dependencies with the slowest thread). This unnecessarily harsh synchronization takes its toll and the problem is even further aggravated with increasing number of cores.

On the Cyclops-64 architecture this parallelization strategy does not scale very well with the number of cores, although there is assisting hardware support for barriers. Even an increase in the problem size, which helps to mitigate the overhead of ramping up and down the wavefront, does not provide significant performance gains either. Figure 7.1 shows the speedup for the barrier version of the benchmark with a maximum speedup of 24x.

7.2 Wavefront Computation with Signal-Wait

Another well known synchronization construct is signal-wait. Signal-wait can be seen as a fine-grain synchronization construct when compared to barriers. Instead of synchronizing a set of threads, it allows a finer control akin to point-to-point synchronization methods. In this parallelization strategy, the producer can signal the consumer when it has finished the write. The consumer will wait until the signal arrives and then read the data it was waiting for. Depending on the architecture the Signal and Wait functions might have different implementations and special hardware support might be required. On an out-of-order architecture, a special operation called a fence instruction is required to make sure that the signal from the producer is not sent before the write operation and the read operation from the consumer is not issued before the wait. The overhead of signal-wait can be reduced by unrolling the loop, therefore reducing the number of synchronization operations, much like a smaller scale of the blocked barrier approach.

Experiments on the Cyclops-64 architecture show that this fine-grain parallelization strategy is more successful than the barrier approach. Figure 7.2 shows the speedup for the signal-wait version of the benchmark with a maximum speedup of 72x.

7.3 Wavefront with Fine-Grain In-Memory Synchronization

The last synchronization construct used in this experimental study is a dataflow-like fine-grain in-memory synchronization method. In fact, three different versions of this fine-grain synchronization construct are used, which are described in detail in

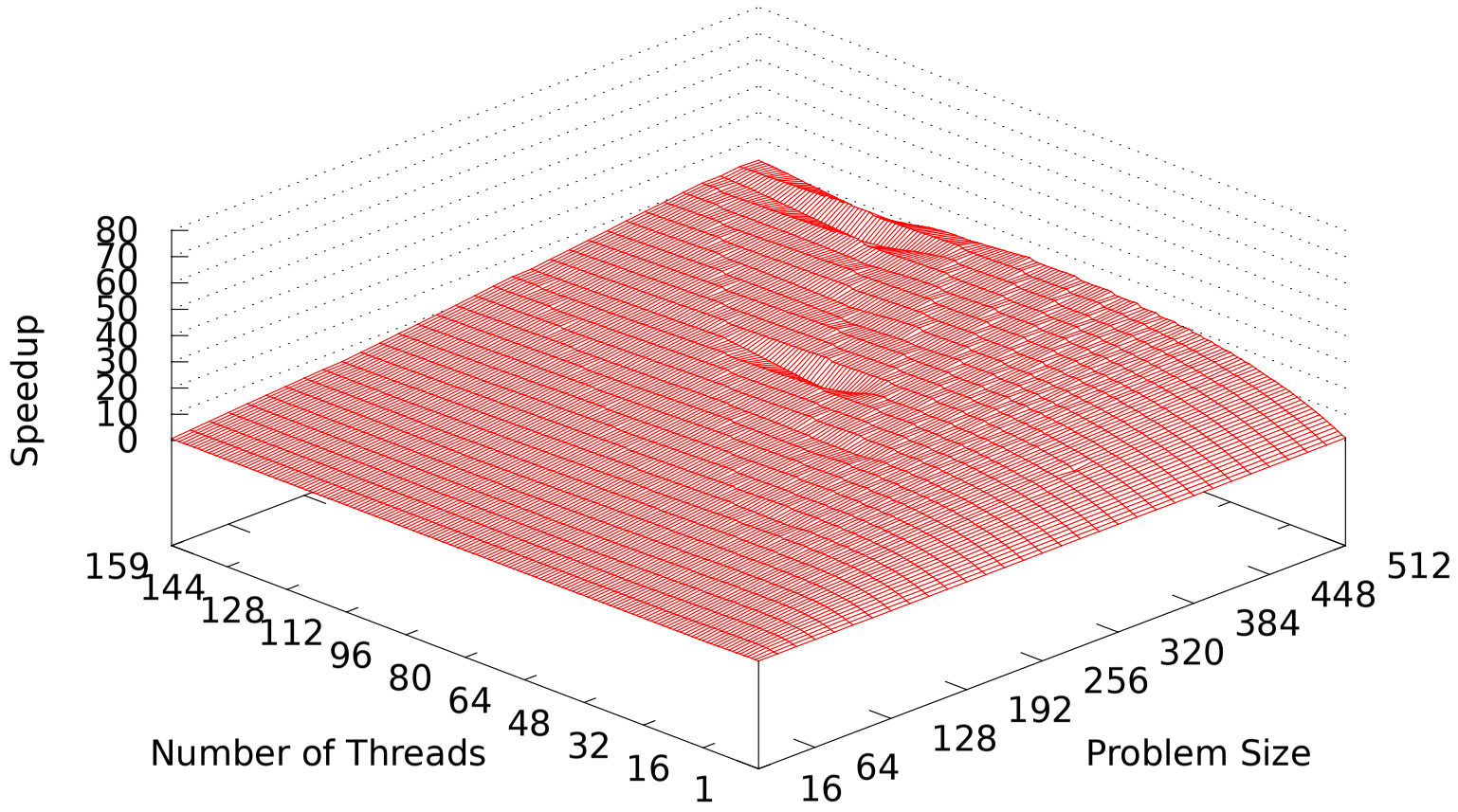


Figure 7.1: Wavefront Speedup (Barrier)

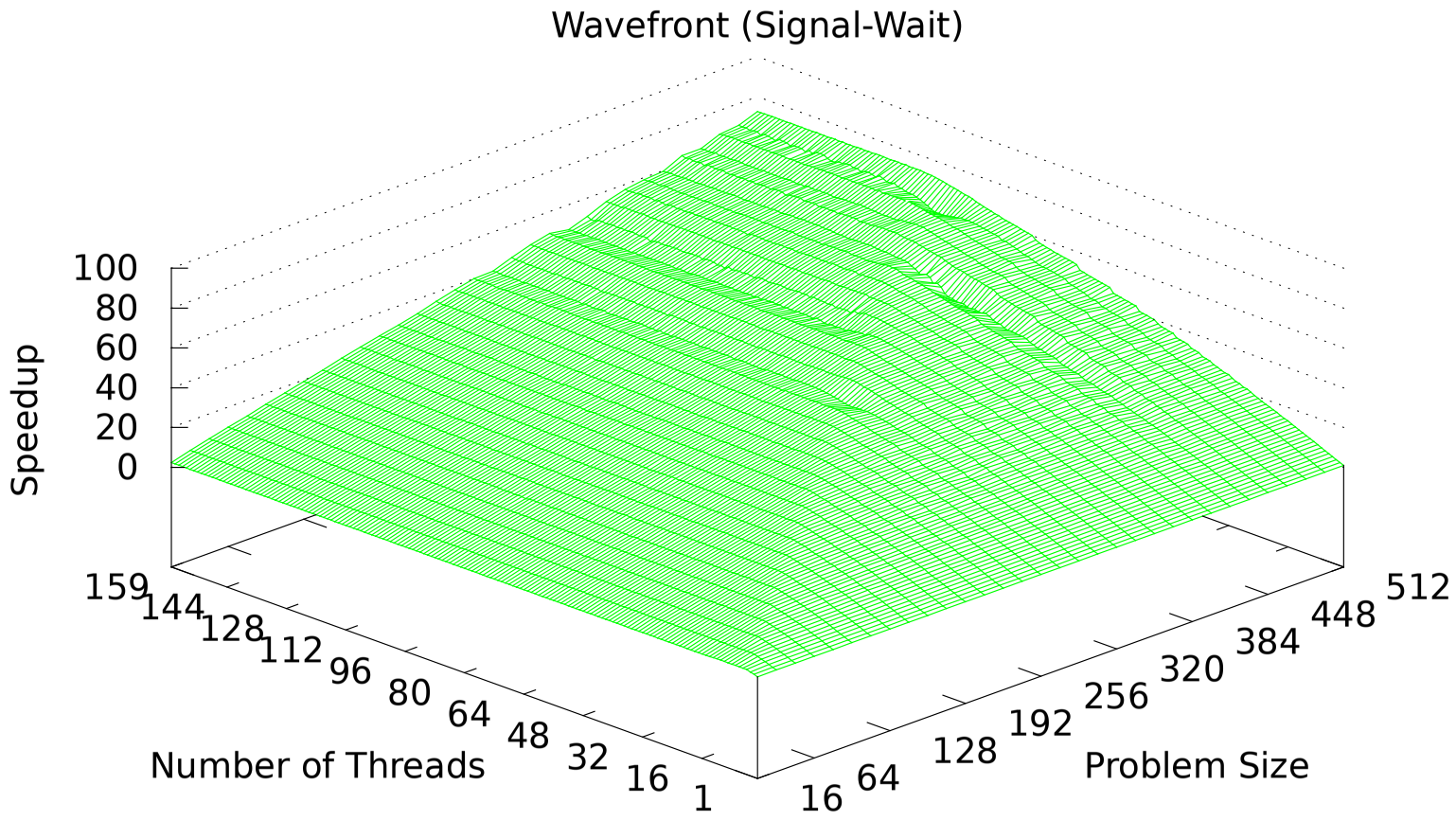


Figure 7.2: Wavefront Speedup (Signal-Wait)

Section 4.4 and Chapter 5. The important difference between the three versions is that the first two versions are blocking, while the last one is non-blocking. They use the same approach as signal-wait, but replaces the Signal and Wait functions with synchronizing load and store instructions, which are supported in hardware. Much better results are expected from these synchronization constructs, because it only synchronizes the load and store and not any unrelated memory operations.

The first two synchronization constructs prove to be faster than the barrier approach, but are still slower than the signal-wait implementation. This is due to the blocking behavior of the first two synchronization constructs, which is fatal for in-order-issue processors. Signal-wait is blocking on the receiver side, but not on the sender side. The first two fine-grain synchronization constructs are blocking on both the sender and receiver side. To solve this dilemma, the third fine-grain synchronization construct is non-blocking on both sides - sender and receiver. This change achieves promising results. The third implementation beats all other implementations in every case. It receives maximum speedup for any problem size and scales much better with the number of threads. Even small problem sizes achieve better speedup with this implementation than with any of the previous synchronization constructs. Figures 7.3, 7.4, and 7.5 show the speedups for the different fine-grain synchronization versions of the benchmark with a maximum speedup of 60x, 50x and 94x, respectively.

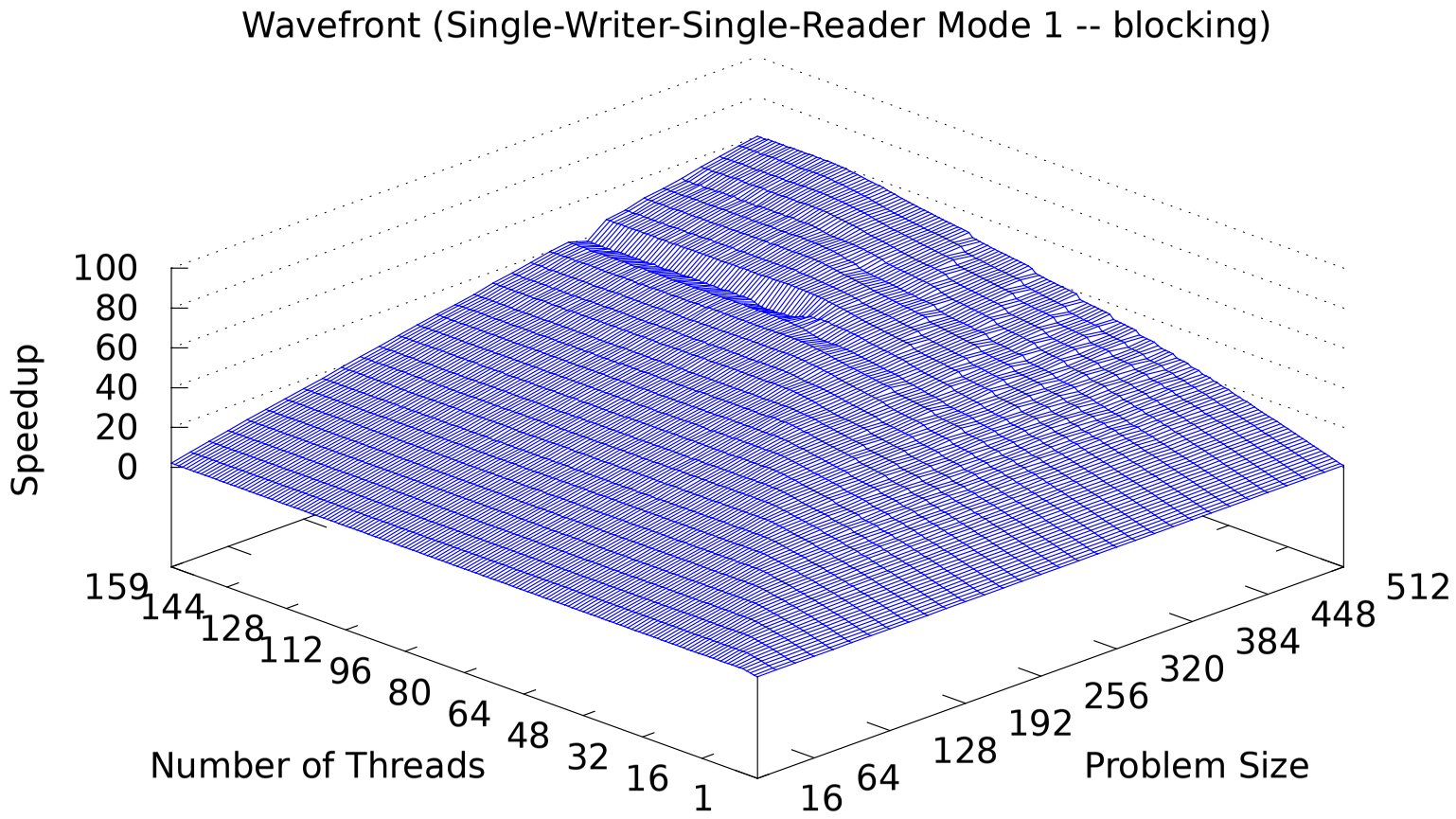


Figure 7.3: Wavefront Speedup (SWSR 1)

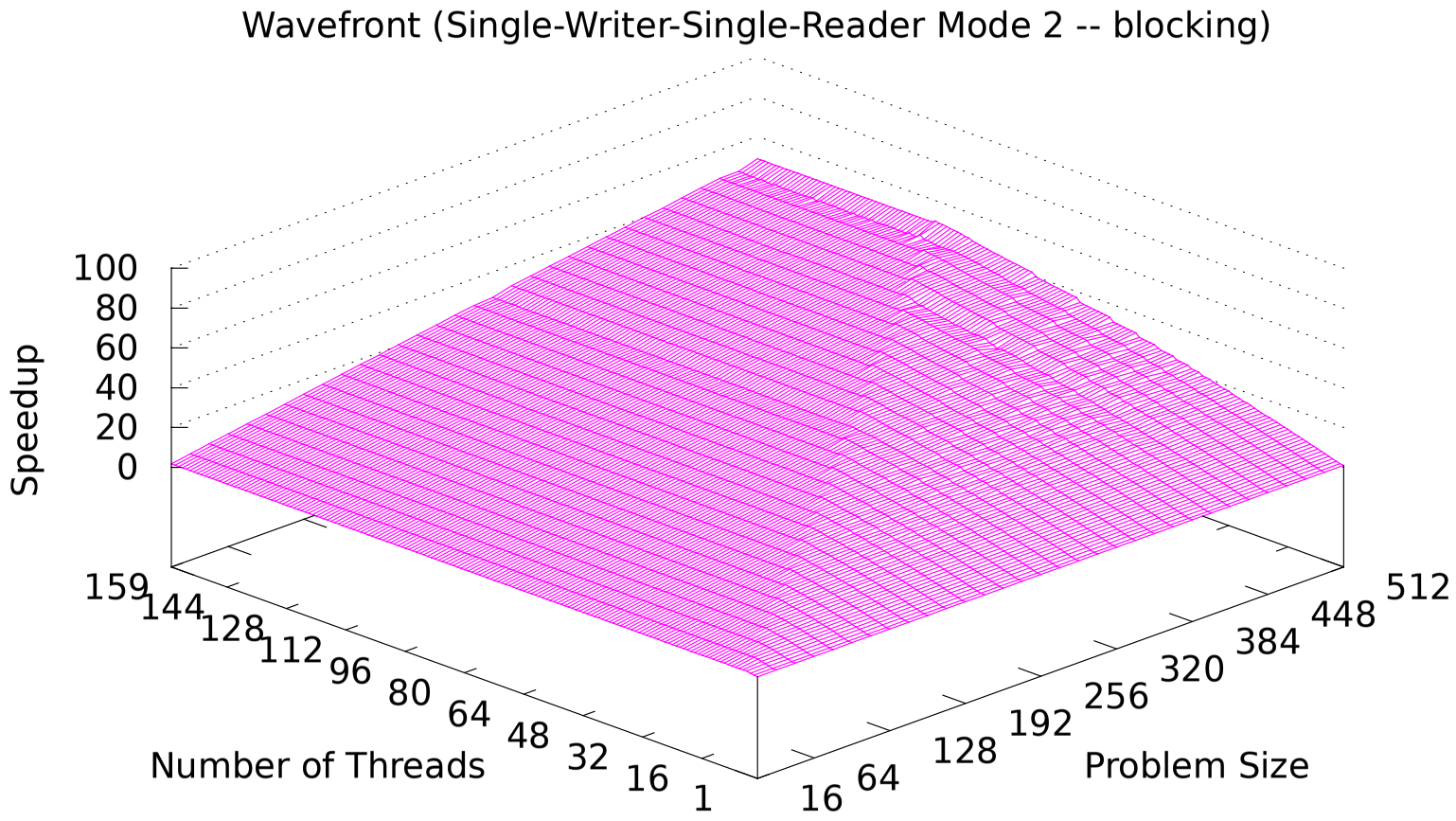


Figure 7.4: Wavefront Speedup (SWSR 2)

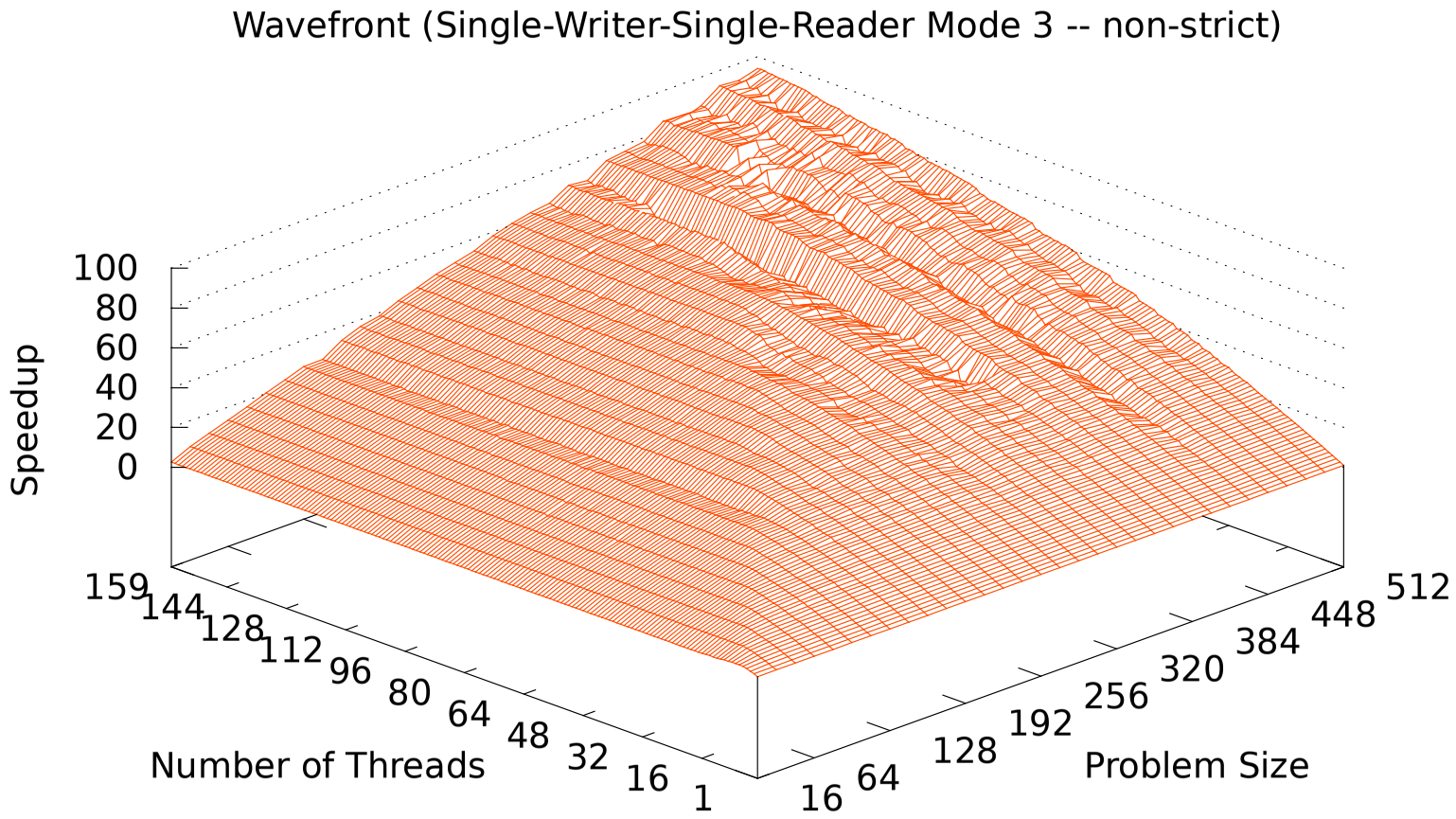


Figure 7.5: Wavefront Speedup (SWSR 3)

Chapter 8

THE ADVANTAGES AND DISADVANTAGES OF NON-STRICT SYNCHRONIZATION

The different E-SSB modes provide all the feature of fine-grain synchronization, but conceptually they are quite different. This affects the way how they have to be used programmatically and therefore also affect performance. Figure 8.1 shows the assembly code for the wavefront kernel using E-SSB 1 for synchronization. The loop has been unrolled so that four iterations are processed at one. This reduces not only the usual loop overhead such as pointer increments and branching, but also reduces the number of synchronizations operations needed. Line 9-11 show the loop construct that is need for E-SSB 1 to read a synchronized variable. The loop will keep spinning until the memory location has been marked as full by the producer. This requirement to check the return code to make sure that the data is available makes the overall construct blocking. The code actually takes advantage of this blocking behavior. The following load instructions in line 12-15 would all requires synchronization too, but by making sure that the synchronization is successful first before the execution continues, it is guaranteed that the following load instruction are reading the correct values. This is because the producer that creates these values writers first all the unsynchronized values and then the synchronized value last (see line 34 - 36). The characteristics of this first mode are that both, the write and the read operation, are blocking. That means the processor cannot continue operation until the result comes back, which is a full round-trip through the crossbar for this code example. The impact for code size is minimal, since only one additional branch instruction is required to check for the result of the operation. There is more crossbar traffic for two reasons. First there might be more traffic if the read fails, because it has to be repeated. Second, the synchronized

write also receives a return code back. A normal store instruction would not receive any data, so there is also one more crossbar return package.

The blocking behavior is not a direct effect of the E-SSB 1 instruction itself, but how it is being used. The code in Figure 8.2 uses optimistic speculation. Instead of checking the return code right away, the code first issues all the required load instructions (line 11-14) and then checks the return code (line 15). If the return indicates success, then all the data returned from the load instructions contain the correct data. If not, then all the operations have to be repeated. The code size in this particular example does not increase, but more sophisticated compiler generate code might schedule also other instructions that could update loop induction variables before the return code is checked. In this case an alternative code path must be taken to undo the changes already made or do not repeat them. This allows even further speculation and overlap of instructions, but also negatively affects code size and makes code generation and scheduling more difficult. Another more important issue to consider is the potential increase in memory traffic. Every time the synchronization fails all of the memory operation have to be repeated in this example. This puts additional burden on the crossbar network and the memory controllers. It could lead to congestion in the network or exacerbate it. If power is a concern then network traffic should be limited or avoided, since most of the power is used by memory operations [44].

Figure 8.3 displays the code for the same kernel, but using E-SSB 2 instructions for synchronization. The approach is very similar to E-SSB 1 instructions, but additional return code handling is required to send the processor to sleep (line 15) or wake up another processor (lines 46 - 49). The additional instruction overhead is no longer neglectable and can negatively affect instruction caches and the additional branching can lead to more branch mispredictions, since the outcome is not predictable.

The code in Figure 8.4 looks almost identical to a serial implementation of the code. All the load operations that require synchronization are replaced by E-SSB 3 load instructions. The same applies to all store operations. This approach does not require any additional instructions, so it does not affect the instruction cache. It also does

```
1  N <- problem size
2  OOT <- 1/3
3  r11 <- 1
4  r13 <- a[i-1][j  ]
5  r14 <- a[i  ][j  ]
6
7  loop_head:
8  ldd     r15,-8(r13)       # a[i-1][j-1]
9  loop_swsr1_rd:
10 swsr1_rd r16,r13           # a[i-1][j  ]
11 bne     r16,loop_swsr1_rd
12 ldd     r18, 8(r13)       # a[i-1][j+1]
13 ldd     r19,16(r13)       # a[i-1][j+2]
14 ldd     r20,24(r13)       # a[i-1][j+3]
15 ldd     r21,-8(r14)       # a[i  ][j-1]
16 addi    r13,r13,32
17 addi    r11,r11,4         # j += 4
18 cmplt   r26,r11,N         # j < N
19 faddd   r22,r15,r17
20 faddd   r23,r17,r18
21 faddd   r24,r18,r19
22 faddd   r25,r19,r20
23 faddd   r22,r22,r21
24 fmuld   r22,r22,OOT
25 faddd   r23,r23,r22
26 fmuld   r23,r23,OOT
27 faddd   r24,r24,r23
28 std     r23,  8(r14)
29 fmuld   r24,r24,OOT
30 faddd   r25,r25,r24
31 std     r24, 16(r14)
32 fmuld   r25,r25,OOT
33 std     r25, 24(r14)
34 loop_swsr1_wd:
35 swsr1_wd r16,r14,r22
36 bne     r16,loop_swsr1_wd
37 addi    r14,r14,32
38 bt      r26,loop_head
```

Figure 8.1: Assembly Code of the Wavefront Kernel using E-SSB 1

```
1  N <- problem size
2  OOT <- 1/3
3  r11 <- 1
4  r13 <- a[i-1][j  ]
5  r14 <- a[i  ][j  ]
6
7  loop_head:
8  ldd      r15,-8(r13)        # a[i-1][j-1]
9  loop_swsr1_rd:
10 swsr1_rd r16,r13            # a[i-1][j  ]
11 ldd      r18, 8(r13)        # a[i-1][j+1]
12 ldd      r19,16(r13)        # a[i-1][j+2]
13 ldd      r20,24(r13)        # a[i-1][j+3]
14 ldd      r21,-8(r14)        # a[i  ][j-1]
15 bne      r16,loop_swsr1_rd
16 addi     r13,r13,32
17 addi     r11,r11,4          # j += 4
18 cmplt    r26,r11,N          # j < N
19 faddd    r22,r15,r17
20 faddd    r23,r17,r18
21 faddd    r24,r18,r19
22 faddd    r25,r19,r20
23 faddd    r22,r22,r21
24 fmuld    r22,r22,OOT
25 faddd    r23,r23,r22
26 fmuld    r23,r23,OOT
27 faddd    r24,r24,r23
28 std      r23,  8(r14)
29 fmuld    r24,r24,OOT
30 faddd    r25,r25,r24
31 std      r24, 16(r14)
32 fmuld    r25,r25,OOT
33 std      r25, 24(r14)
34 loop_swsr1_wd:
35 swsr1_wd r16,r14,r22
36 bne      r16,loop_swsr1_wd
37 addi     r14,r14,32
38 bt       r26,loop_head
```

Figure 8.2: Assembly Code of the Wavefront Kernel using E-SSB 1 and Optimistic Speculation

```
1  N <- problem size
2  OOT <- 1/3
3  r11 <- 1
4  r13 <- a[i-1][j  ]
5  r14 <- a[i  ][j  ]
6
7  loop_head:
8  ldd     r15,-8(r13)       # a[i-1][j-1]
9  loop_swsr2_rd:
10 swsr2_rd r16,r13
11 cmpieq  r18,r16,-1
12 bt      r18,loop_swsr2_rd
13 cmpine  r18,r16,-2
14 bt      r18,swsr2_rd_done
15 sleep   r0
16 b       loop_swsr2_rd
17 swsr2_rd_done:
18 ldd     r18, 8(r13)       # a[i-1][j+1]
19 ldd     r19,16(r13)       # a[i-1][j+2]
20 ldd     r20,24(r13)       # a[i-1][j+3]
21 ldd     r21,-8(r14)       # a[i  ][j-1]
22 addi    r13,r13,32
23 addi    r11,r11,4         # j += 4
24 cmplt   r26,r11,N         # j < N
25 faddd   r22,r15,r17
26 faddd   r23,r17,r18
27 faddd   r24,r18,r19
28 faddd   r25,r19,r20
29 faddd   r22,r22,r21
30 fmuld   r22,r22,OOT
31 faddd   r23,r23,r22
32 fmuld   r23,r23,OOT
33 faddd   r24,r24,r23
34 std     r23,  8(r14)
35 fmuld   r24,r24,OOT
36 faddd   r25,r25,r24
37 std     r24, 16(r14)
38 fmuld   r25,r25,OOT
39 std     r25, 24(r14)
```

Figure 8.3: Assembly Code of the Wavefront Kernel using E-SSB 2

```
40 loop_swsr2_wd:
41 swsr2_wd  r16,r14,r22
42 cmpieq   r17,r16,-1
43 bt       r17,loop_swsr2_wd
44 cmpieq   r17,r16,-2
45 bt       r17,swsr2_wd_done
46 shli     r17,r16,4
47 ori      r17,r17,0x4002
48 shli     r17,r17,16
49 std      r0,0(r17)
50 swsr2_wd_done:
51 addi     r14,r14,32
52 bt       r26,loop_head
```

Figure 8.3: Assembly Code of the Wavefront Kernel using E-SSB 2 (continued)

not introduce any additional branches that could lead to flushes of the pipeline due to branch misprediction. The instructions are also not blocking and there is no additional overhead for the crossbar either. But this does not come for free. The instructions are no longer blocking, therefore all instructions need now to be synchronized, which increasing the pressure on the E-SSB.

Figure 8.5 illustrates the advantages of the non-strict behavior of an E-SSB 3 load operation. Even if the load operation is outstanding, other operations can still be issued in-order until a register dependency is met. The register dependencies are enforced by the scoreboard. An E-SSB 3 load or store operation produces the exact same number of crossbar packages as their regular memory load and store counterparts. This means no additional crossbar overhead and possible congestion is introduced by using these operations.

```
1  N <- problem size
2  OOT <- 1/3
3  r11 <- 1
4  r13 <- a[i-1][j   ]
5  r14 <- a[i   ][j   ]
6
7  loop_head:
8  ldd      r15,-8(r13)        # a[i-1][j-1]
9  ldds     r17, 0(r13)        # a[i-1][j   ]
10 ldds     r18, 8(r13)        # a[i-1][j+1]
11 ldds     r19,16(r13)        # a[i-1][j+2]
12 ldds     r20,24(r13)        # a[i-1][j+3]
13 ldd      r21,-8(r14)        # a[i   ][j-1]
14 addi     r13,r13,32
15 addi     r14,r14,32
16 addi     r11,r11,4          # j += 4
17 cmplt    r26,r11,N          # j < N
18 faddd    r22,r15,r17
19 faddd    r23,r17,r18
20 faddd    r24,r18,r19
21 faddd    r25,r19,r20
22 faddd    r22,r22,r21
23 fmuld    r22,r22,OOT
24 faddd    r23,r23,r22
25 stds     r22,-32(r14)
26 fmuld    r23,r23,OOT
27 faddd    r24,r24,r23
28 stds     r23,-24(r14)
29 fmuld    r24,r24,OOT
30 faddd    r25,r25,r24
31 stds     r24,-16(r14)
32 fmuld    r25,r25,OOT
33 stds     r25, -8(r14)
34 bt       r26,loop_head
```

Figure 8.4: Assembly Code of the Wavefront Kernel using E-SSB 3

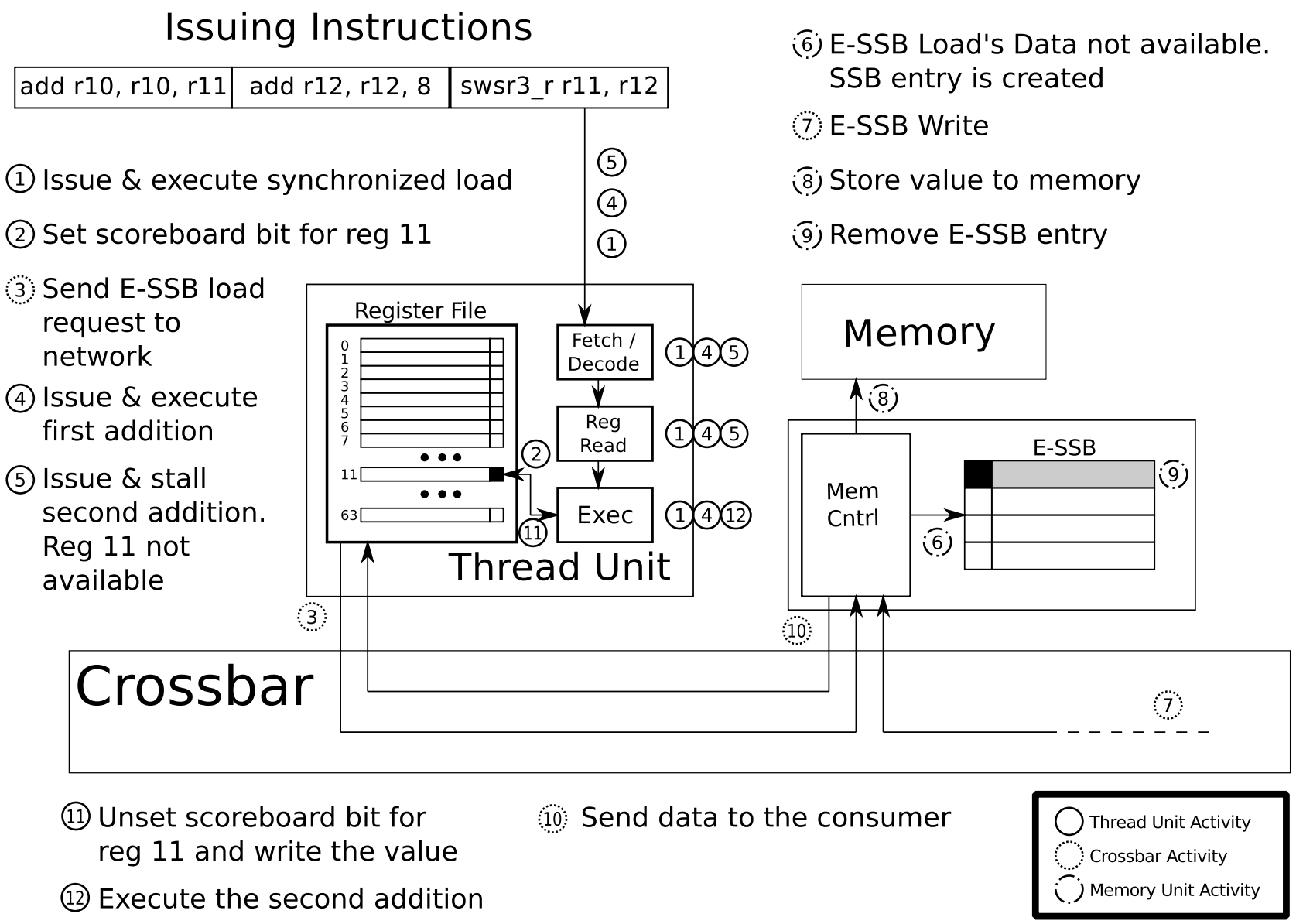


Figure 8.5: E-SSB 3 Example

Chapter 9

E-SSB EXPERIMENTAL TESTBED

For experimental performance evaluation the proposed Extended Synchronization State Buffer (E-SSB) is implemented at the Hardware Description Language (HDL) level of the Cyclops-64 (C64) architecture. Moreover, the Delaware Enhanced Emulation Platform (DEEP) [45] is used to emulate this many-core architecture. This emulation platform is fast and gate-level accurate compared to available software based methods. It is capable of emulating the whole many-core design with a relatively small number of FPGAs (32 Altera Stratix II) thanks to the Delaware Iterative Multiprocessor Emulation System (DIMES) mode.

9.1 DEEP: FPGA-based Emulation System

In this section we describe the hardware platform, the emulation methodology and the debugging support of the system. DEEP has been developed in order to both validate the Cyclops-64 architecture features and test its software stack. In the context of this thesis its use has been extended for benchmarking and performance evaluation. Figure 9.1 shows a frontal-view of the hardware platform. The challenge and cost in testing new hardware designs lies in the difficulty of verifying whether a circuit will work under real-world conditions. Software-based simulators can get close to the behavior of the real circuit, but take much longer to execute than the actual hardware. Therefore, it is unrealistic to run a great variety or larger benchmarks for a whole chip on existing software-based logic simulators to verify hardware design and/or test its software stack. Many concurrency related bugs are discovered by running real-world applications using full system emulation. While high emulation speed is required, it is very important to quickly respond to logic design changes, especially, at the early stage of logic design.

Although the amount of bugs at the early stages of development are usually high, they are easily found using simple synthetic test cases or small kernel benchmark. That means turn-around time regarding logic changes is more important than emulation speed - at least during early stages of development. The major objectives of DEEP are to support all design and test stages, to realize good turn-around time for the early stages and high emulation speed for the later stages, to do the whole chip emulation as well as to provide an efficient debugging environment.

9.2 DEEP Hardware Platform

The hardware platform of DEEP is comprised of a host system and a custom made system with a series of highly connected FPGAs. The host system can be any off-the-shelf PC with an Ethernet connection to run the control- and debugging software. Figure 9.2 shows a block diagram of DEEP. Inside the cabinet is a big backplane with power supply circuitry. 16 FPGA boards are plugged into the backplane which provides not only power, but also the global clock (100MHz) and interconnection to all FPGA boards. There are three different type of FPGA boards. The top and bottom row of FPGA boards are the processing boards. The most left one is the root board and the remaining boards in the middle row are the switching boards. The root board has two Altera Stratix II 2S90 FPGAs and a daughter board for the Ethernet connection to the host system. The Ethernet daughter board has additional logic, which allows the remote programming of all FPGAs in the system via Ethernet. The FPGAs on the root board are used to implement the root node of a tree. The remaining FPGAs in the system are connected in a tree like fashion to the root board. This allows the host system to communicate with all FPGAs. The processing board has five FPGAs. One Cyclone 1C4 FPGA for the tree node, two Stratix II 2S90 FPGAs for the emulation logic, and two Cyclone II 2C35 FPGAs for interfacing logic to the DIMMs. These additional FPGAs for the memory interface are required to refresh the memory while reprogramming the emulation logic in the other FPGA. The switching board has two Stratix II 2S90 FPGAs which are used to implement the

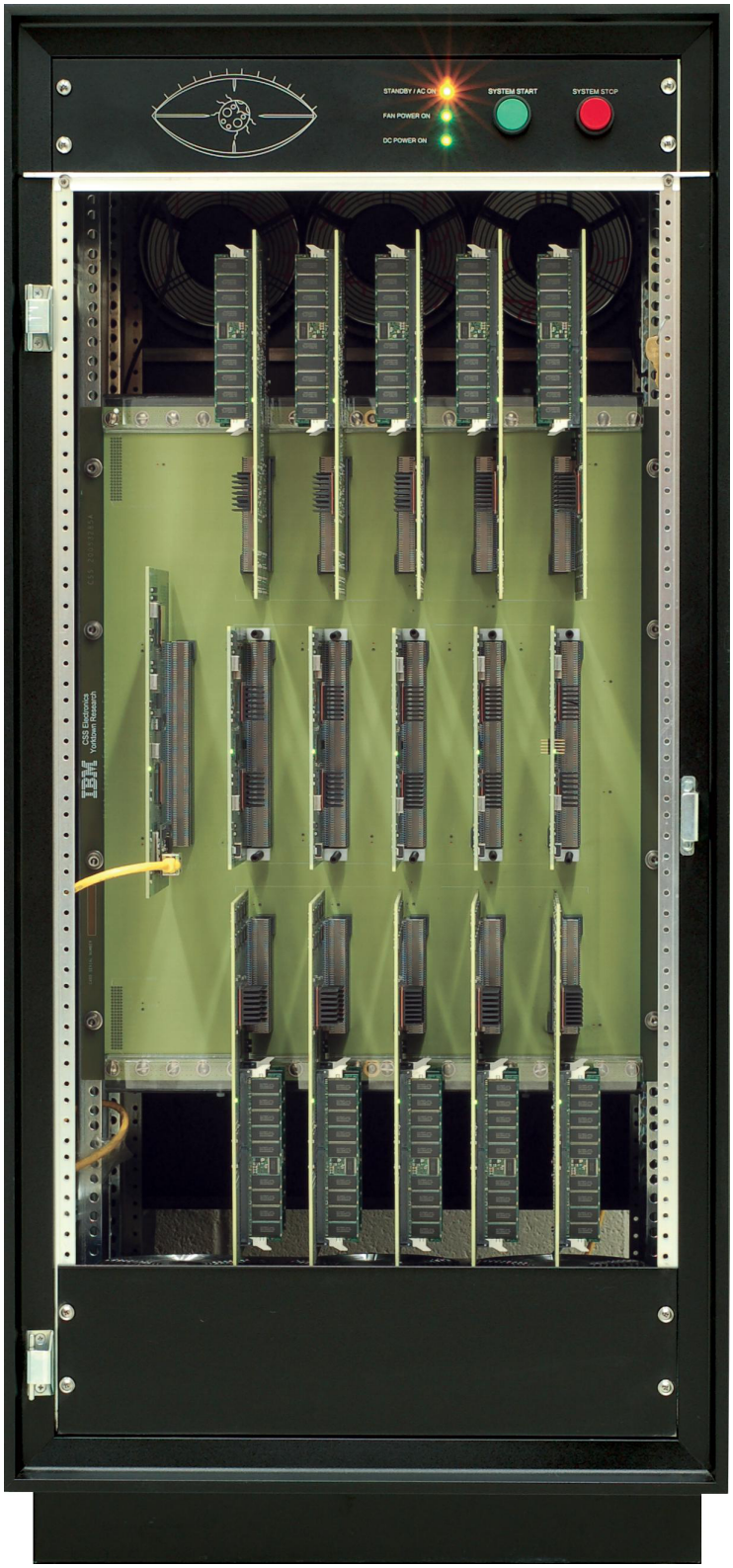


Figure 9.1: DEEP: The emulation platform consists of 32 Altera Stratix II FPGAs; 20 for processing units, 10 for switches, and 2 for host communication.

switching logic for the emulation system. The processing boards are connected to the switching boards. These connections are used during emulation to pass data between the different processing boards. The tree is only used by the host for communication with the FPGAs. Overall, only 20 Stratix II 2S90 FPGAs can be used for emulating user logic.

9.3 DEEP Emulation Methodology

In order to achieve its main objective DEEP supports two different modes: simulation mode and emulation mode. The simulation mode is a logic processor based logic simulation methodology. In this mode, the original logic design is translated into logic programs, and then these logic programs are run on a large number of logic processors. Usually, a logic design consists of a netlist of gates and memory cells including Flip-Flops (FF). It can always be mapped to a series of primitive logic operations such as AND, OR, etc.. Figure 9.3 shows the translation of a logic design into a logic program.

DEEP can quickly generate logic programs from an original logic design, due to simple translation. For instance, the Cyclops-64 combinatorial logic design (around 43 million gates) can be translated into logic programs within two minutes. Logic programs generated from an original design are executed on a huge number of logic processors, which are implemented on the processing FPGAs. Each processing FPGA has 20 logic processors, and one instruction queue is shared by all processors in one FPGA. Therefore, at most 20 different submodules in a design can be simulated in this system. If one submodule has more than 20 instances, multiple processing FPGAs are utilized for it. The simulation mode is available on a general workstation as well, so logic simulation can be done anywhere without the hardware, although the simulation speed is much slower.

On the other hand, the emulation mode design is based on an iterative emulation methodology [46]. Since the whole many-core architecture design cannot fit into a single FPGA of DEEP, or any current available FPGA on the market, the architectural design

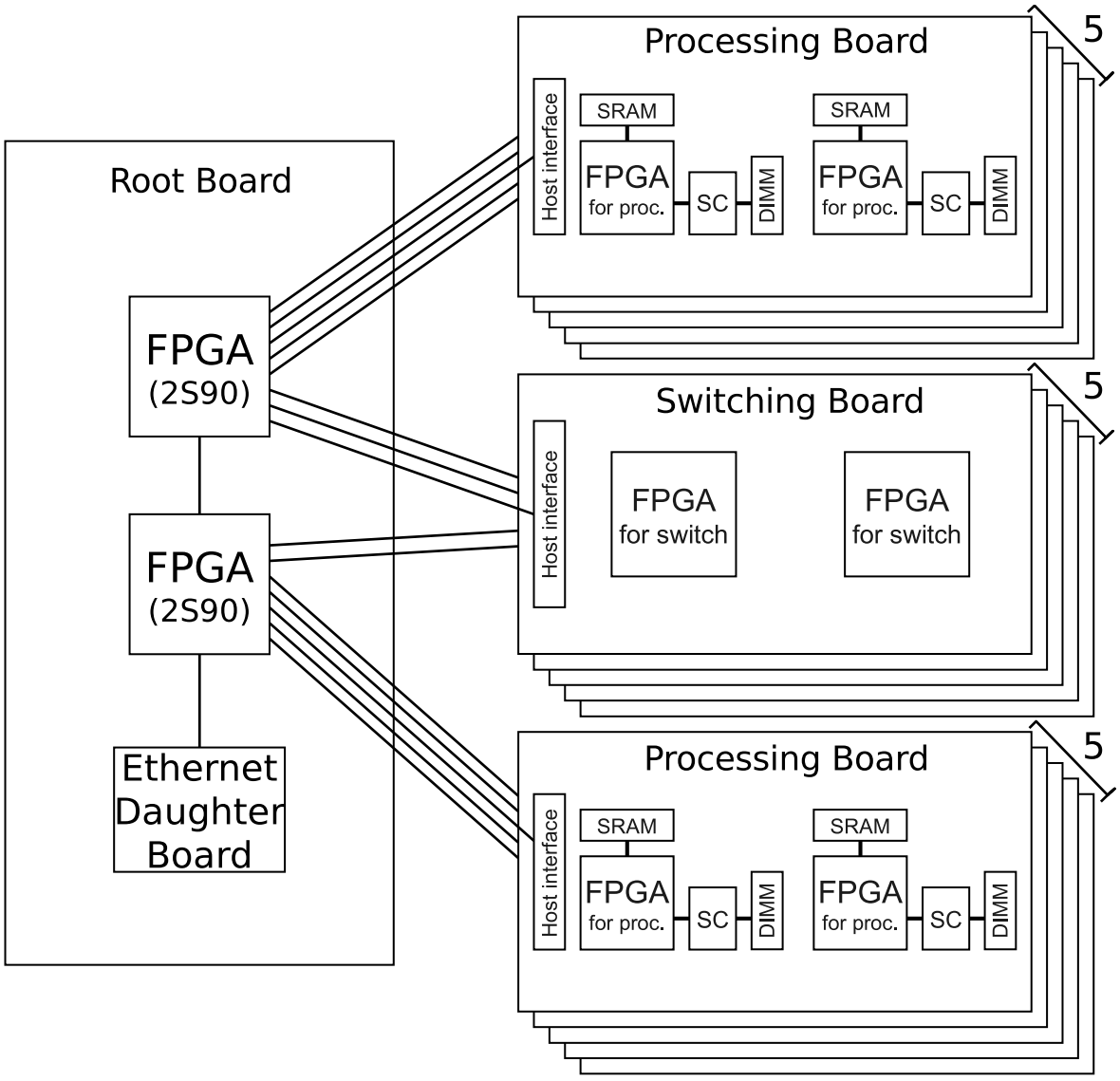


Figure 9.2: Block Diagram of DEEP: The figure shows the tree-like connections between the DEEP FPGAs and the different board types - root board, processing boards, and switching boards.

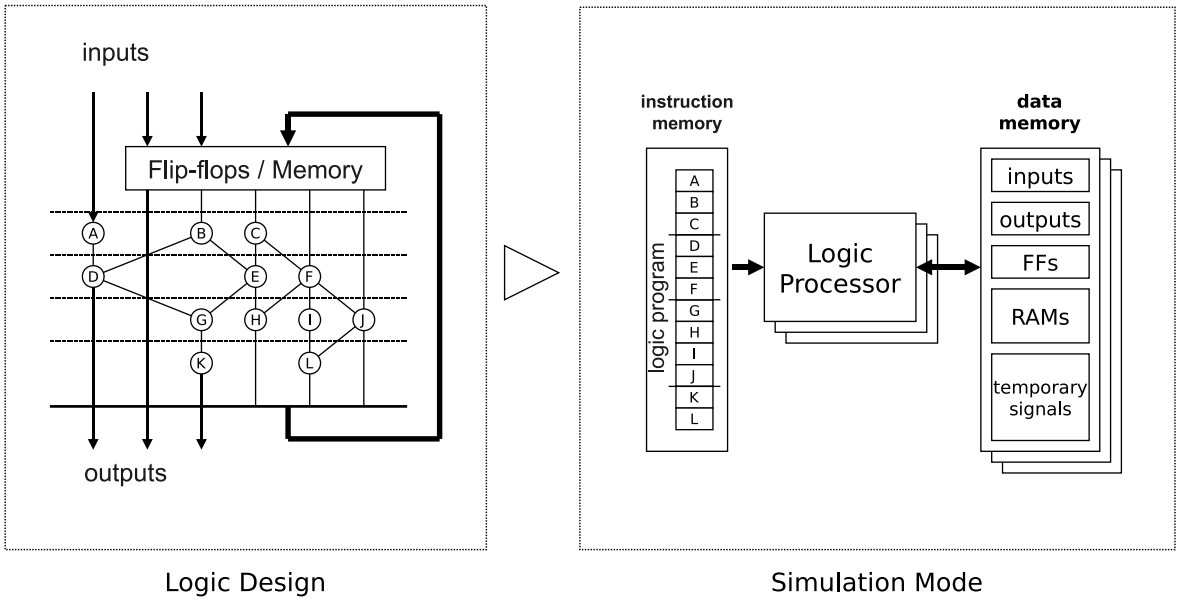


Figure 9.3: DEEP Simulation Mode: User Logic to Logic Program mapping. The logic primitives (A-L) shown in the original logic design on the left are translated into instructions for the logic processor shown on the right.

is separated into submodules, which can fit into a FPGA. Even though each submodule fits into one FPGA, a lot of FPGAs would be still required to implement the entire chip in the emulation system. Furthermore, many hardware resources would be required for communication between submodules in different FPGAs. Instead of mapping each submodule to a different FPGA, the emulation system adopts an iterative emulation approach (see Figure 9.4).

Combinatorial logic equivalent submodules are implemented on only one (or a few FPGAs), and then iteratively utilized to emulate all instances of the submodule. This emulation method drastically reduces the necessary number of FPGAs. Each submodule's FFs and internal RAM blocks are isolated from the original logic design. The content of the FFs and RAMs are independent of each submodule's instance, so they must be stored separately. The emulation system utilizes internal memories for FFs and external memories for RAM blocks, and only the combinatorial logic is implemented in the FPGA. The flow described above is done by the DEEP software

automatically. By adopting the iterative emulation methodology, huge logic designs, which cannot fit into an existing single FPGA, can be emulated in a few FPGAs. Because a target logic design needs to be synthesized and mapped into a FPGA, it takes much more preparation time than the simulation mode until the logic design is ready to be emulated. However, after the logic design is mapped into the FPGA, it runs at native combinatorial logic speed of the FPGA even though it is required to emulate the logic iteratively. In case of the Cyclops-64 design (with E-SSB extension) the average emulation speed of the whole chip is around 20k cycles/sec.

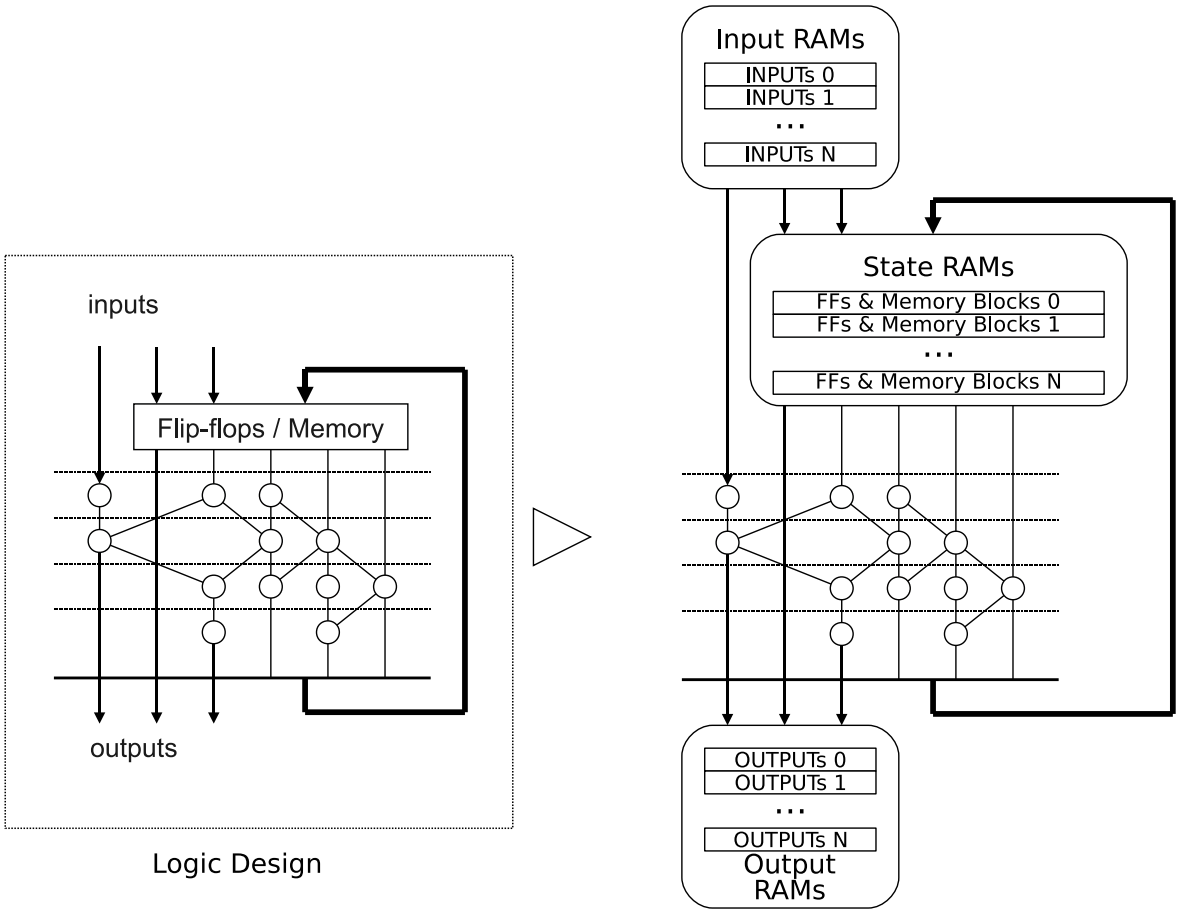


Figure 9.4: DEEP Emulation Mode: User Logic to Iterative Emulation mapping. FFs and RAMs are extracted from the original logic design on the left and mapped to instance addressable memory blocks in the FPGA. The remaining combinatorial logic is used iteratively in the FPGA.

9.4 DEEP Debugging Support

In simulation mode, it is easy to check any signals since all logic is executed as logic programs. If inputs, outputs and contents of FFs/RAM blocks need to be observed, the DEEP host directly accesses an external memory where the data is stored. For other signals, additional processing is required, because all intermediate signals are overwritten in the local temporary memory of a logic processor which is unreachable from the host. First, the host sets a breakpoint in the debugging special-purpose register of the logic processor. Second, the logic processor starts execution until the PC reaches the breakpoint. Third, the debugging control unit issues several logic instructions to move the debugged signal to the external memory of the FPGA. Finally, the host loads the data. For this debugging feature, there are 16 special purpose registers reserved. If more than 16 signals in one submodule are necessary to be observed at the same time, the host needs to repeat this process for every set of 16 signals. Moreover, not only simple signal tracing is possible, but also program tracing is supported when a processor is simulated. Using program tracing, correctness of a target benchmarks can be easily confirmed. If an error is discovered, the system can switch to signal tracing or use both tracing strategies although simulation speed slows down considerably. The key feature of this simulation mode is fast translation into logic programs and good debugging support.

Next, the debugging support is also necessary in the emulation mode because such support is very helpful to locate a bug for long running benchmarks. The software simulator is used in conjunction with the hardware emulator to obtain signals inside combinatorial logic. In this mode, all combinatorial logic is mapped into a FPGA, so it cannot be observed directly. Fortunately, the content of FFs and memory blocks is reachable because they are stored in memories of the FPGAs. Being able to read this data from the emulation hardware, all signals can be observed by simulating combinatorial logic on the DEEP host workstation.

Chapter 10

E-SSB EXPERIMENTAL EVALUATION

This section presents the results obtained from the experimental testbed, using the wavefront computation kernel and selected SPEC OpenMP kernel loops.

10.1 Wavefront Computation

The wavefront computation kernel is implemented in six different versions. The different versions are serial, barrier, signal-wait, and E-SSB Modes 1 to 3. All kernels are hand-coded in assembly. In all versions the inner loop is unrolled four times to reduce the overhead of the synchronization and to allow for a better overlapping of memory operations and arithmetic computation. The benchmark are run on the emulation system for problem sizes starting at 16x16 elements at increments of 16 up to the maximum supported problem size of 512x512 elements. For each problem size, the wavefront benchmark is run with different number of threads, starting with one thread and going up by increments of one to 159 threads. The architecture supports up to 160 hardware threads, but only 159 can be used, because the OS kernel is running on the first thread unit. In the real system this number is even further reduced, because additional thread units are dedicated by the OS for inter-chip communication. Furthermore, with such large chips it is possible that not all thread units are working properly due to manufacturing faults and are therefore disabled.

The runtime is calculated only for the kernel and the speedup is calculated based on the results of the serial version. Figure 10.1 shows the speedups of the different parallel versions.

10.1.1 Barrier

Even though the hardware-enabled barrier is very efficient, the speedup of the application is limited. The weakest link is the slowest thread. All other threads have to wait for it before they can continue doing useful work. Using barriers for these kinds of workloads is not necessarily a good choice, and dynamic scheduling approaches have achieved better results. Nevertheless, the barrier implementation is still considered for two important reasons. First, the barrier is supported in hardware and this study is supposed to compare different hardware supported synchronization constructs. Second, from a programming point of view the barrier seems to be an easy and efficient construct, because the work for each thread is the same. This show that this thinking cannot be applied anymore to many-core architectures and that congestion, bank conflicts, etc., can have unpredictable impacts on a thread's execution. The barrier version of the benchmark achieves a maximum speedup of 24x.

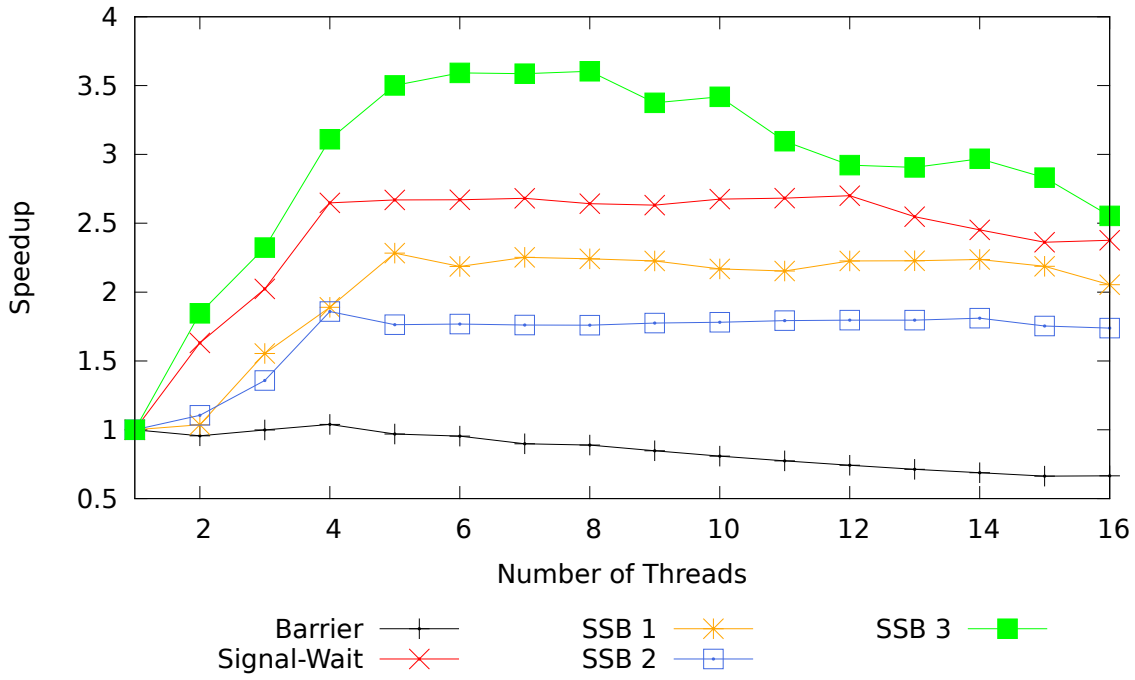
10.1.2 Signal-Wait

The signal-wait version can be implemented very efficiently on the Cyclops-64 architecture by taking advantage of the extensive atomic memory operation support and the local, low-latency scratch pad memory resulting in a speedup of 72x. Figure 3.29 illustrates the synchronization delay of the different benchmark versions. For all examples in this illustration, Thread 1 (consumer) always tries to read the shared data, whereas Thread 2 (producer) is producing this shared data. The first example shows the synchronization delay for signal-wait. The dashed arrows represent accesses to scratch pad memory via the back-door each thread unit has to its own scratch pad memory. This access is much faster, because it does not have to go through the crossbar. Solid arrows represent memory operations that go through the crossbar and therefore take more time. Since the consumer spins on its own local synchronization variable, changes to this variable are observed with little delay. Once the signal from the producer arrives, the consumer can continue execution without any further synchronization related stalls. This allows the overlap of computation and memory operations

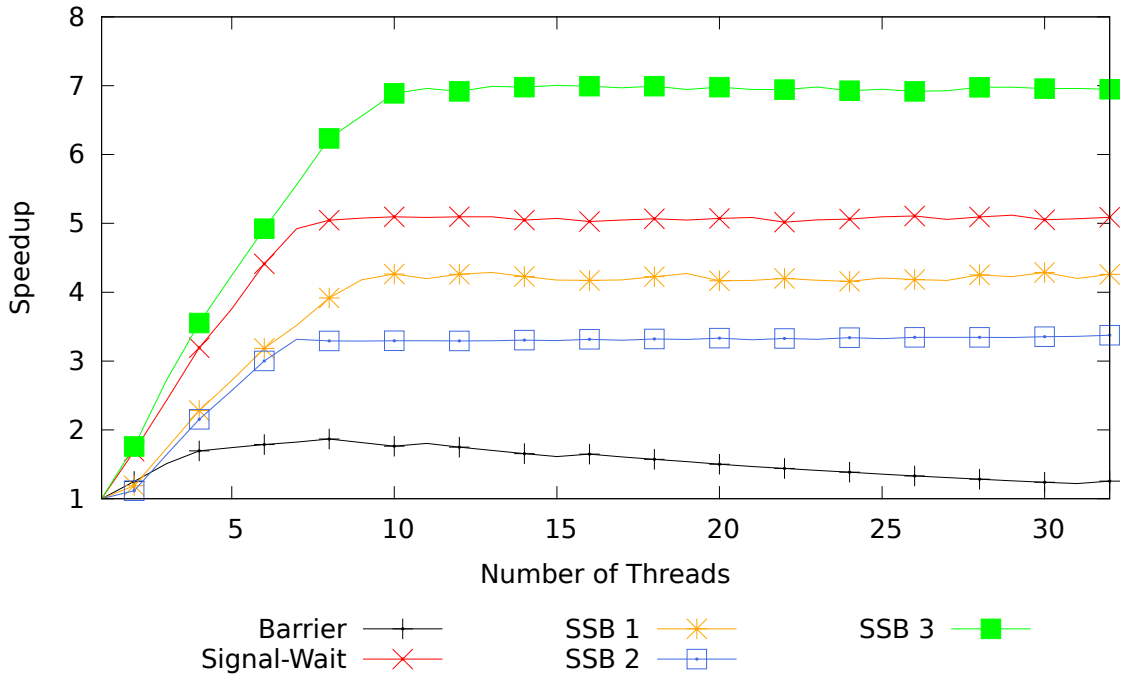
after the wait. The producer does not need to stall at all. This makes signal-wait a very efficient synchronization construct on Cyclops-64.

10.1.3 Fine-grain In-Memory Synchronization

The different SSB versions of the benchmark achieve speedups of 60x, 50x and 94x respectively. A closer look at the benchmarks by using performance counters shows that they are not memory bound. The SSB 1 (busy-wait) version has a synchronization failure rate of 150%. That means every synchronizing load operation has to be repeated 1.5 times on average, because the data has not been written yet by the producer. The SSB 2 (sleep-wakeup) version on the other hand has a failure rate of only 1-2%. Nevertheless, the SSB 1 (busy-wait) approach still achieves better speedups. The second approach generates fewer memory operations and also saves power, but the price is a longer synchronization delay, which hinders parallelism and therefore performance. The SSB 3 (non-strict) version has a failure rate of 25%, but that only means that the load arrived before the store. No additional overhead or memory transactions were required to correct this, because the memory controller has already taken care of it. The second illustration in Figure 10.2 shows that SSB 1 employs a similar busy-waiting approach as signal-wait, but it has to go through the crossbar every time. Furthermore, the producer and the consumer have to stall and cannot overlap any other computation or memory operations until the memory operation on their side has successfully completed. The SSB 2 sleep-wakeup approach in the next example even further aggravates this problem, because now the producer has to wake up the consumer and the synchronization delay increases further. The last SSB mode solves all the problems of the previous versions by performing the synchronization completely in the memory controller. No further action is required from the producer or the consumer. In this mode synchronizing memory operations act like normal memory operations for the thread unit and the synchronization is transparent to them. This allows aggressive scheduling of synchronizing and non-synchronizing memory operations and arithmetic instructions.

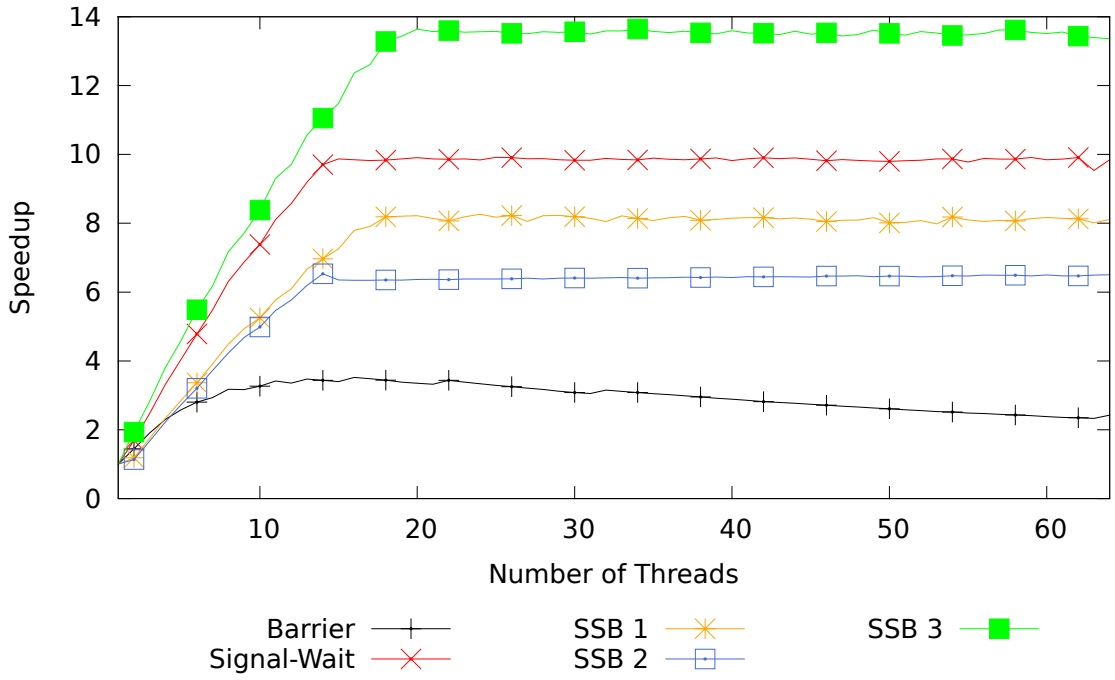


(a) Wavefront (16x16)

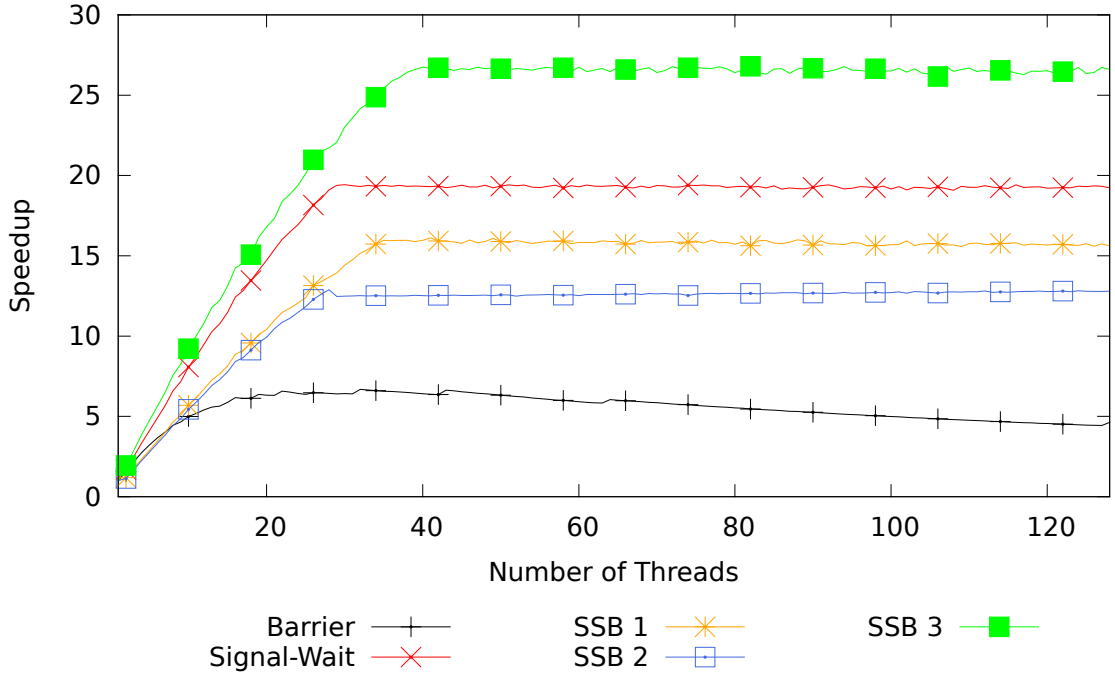


(b) Wavefront (32x32)

Figure 10.1: Wavefront Speedup

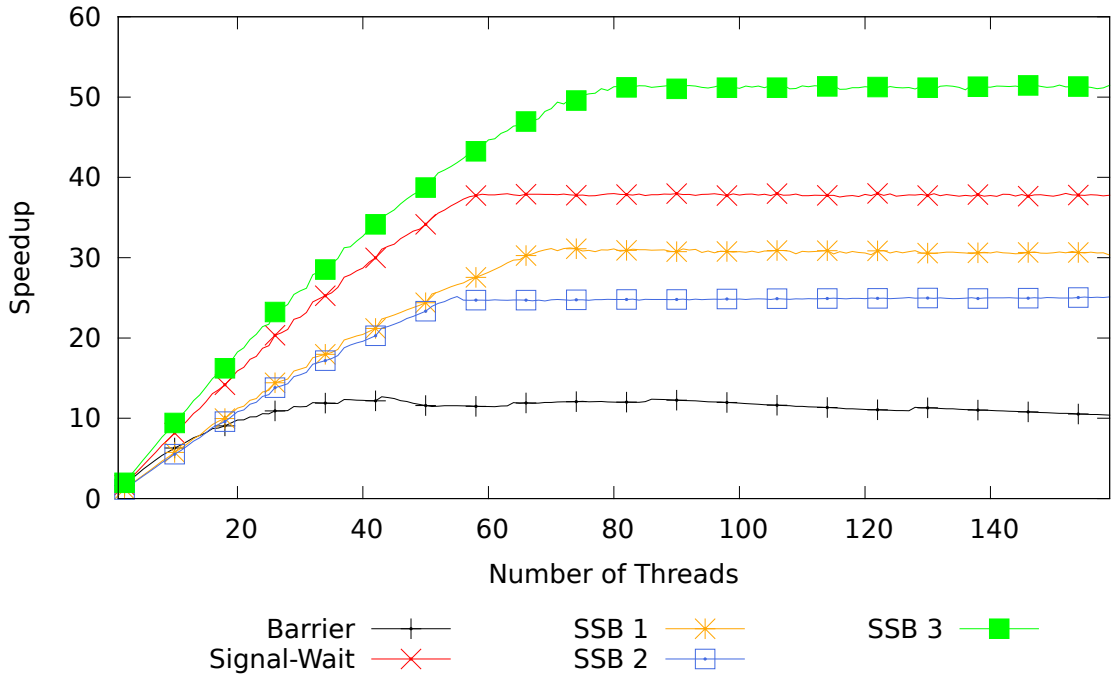


(a) Wavefront (64x64)

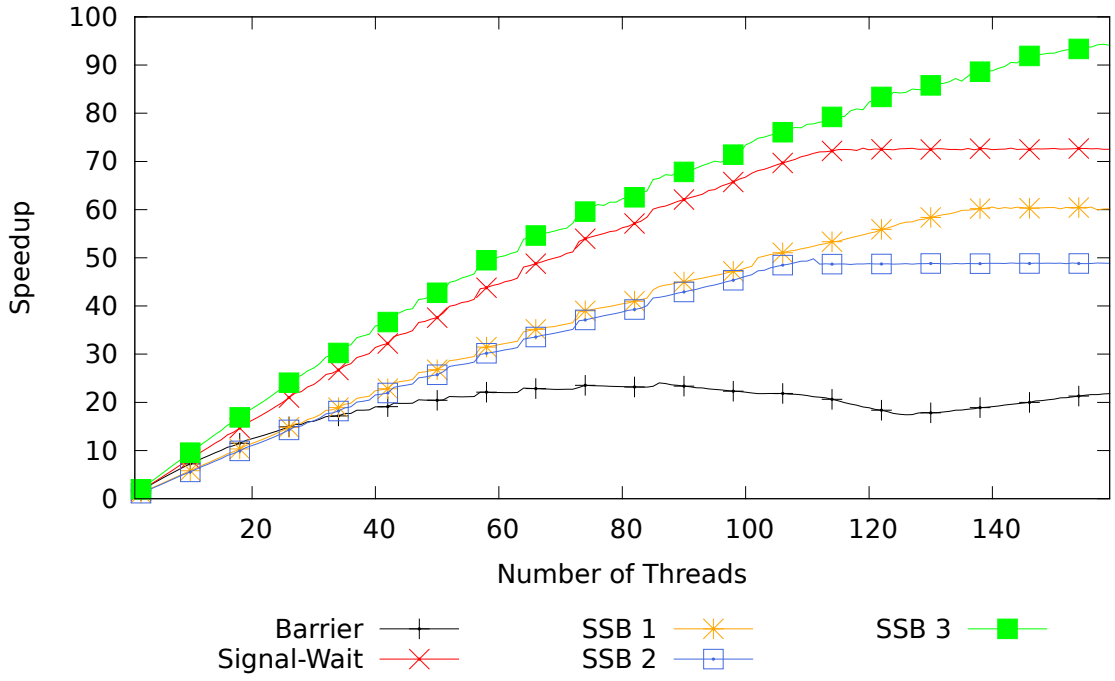


(b) Wavefront (128x128)

Figure 10.1: Wavefront Speedup (continued)



(c) Wavefront (256x256)



(d) Wavefront (512x512)

Figure 10.1: Wavefront Speedup (continued)

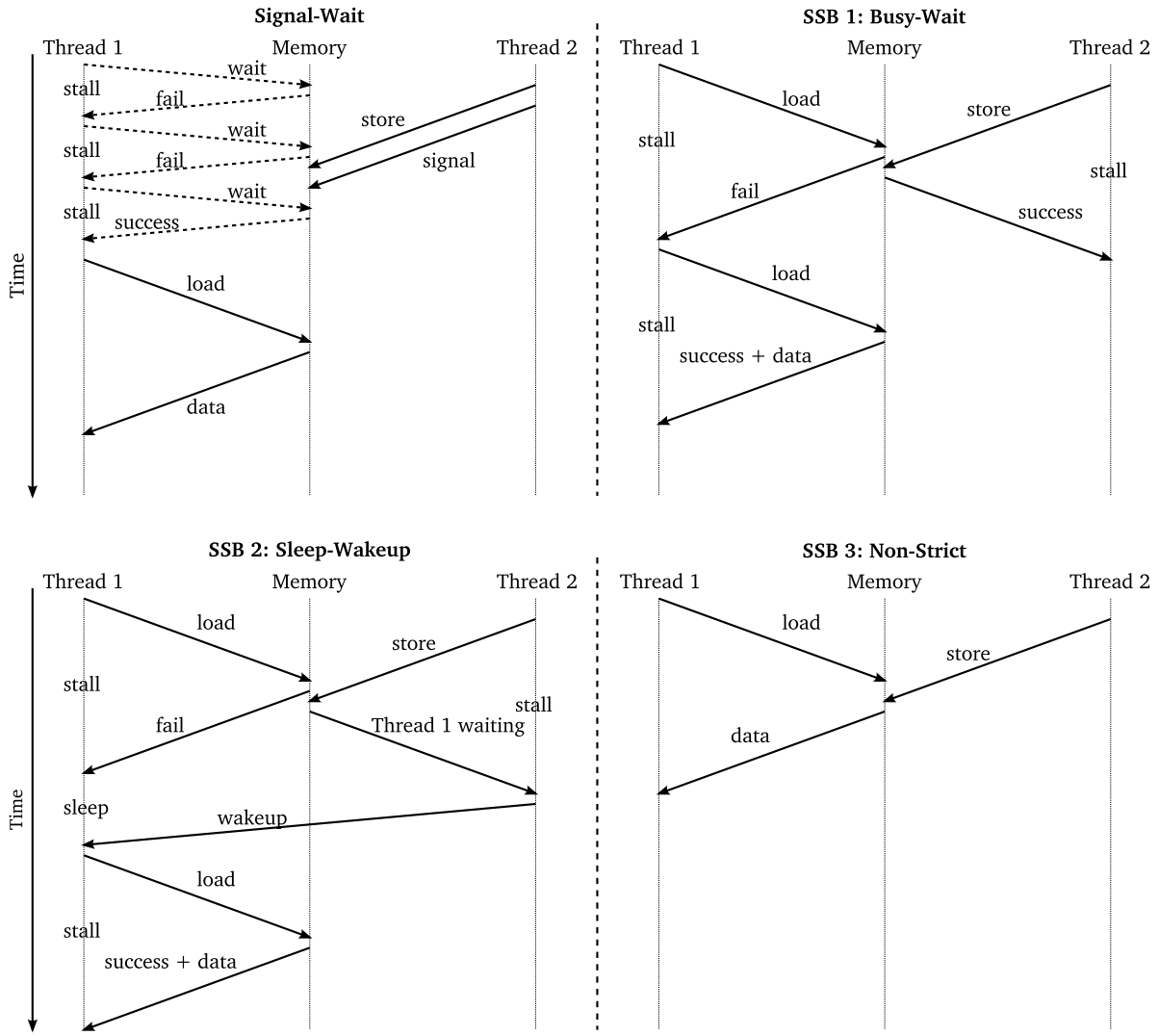


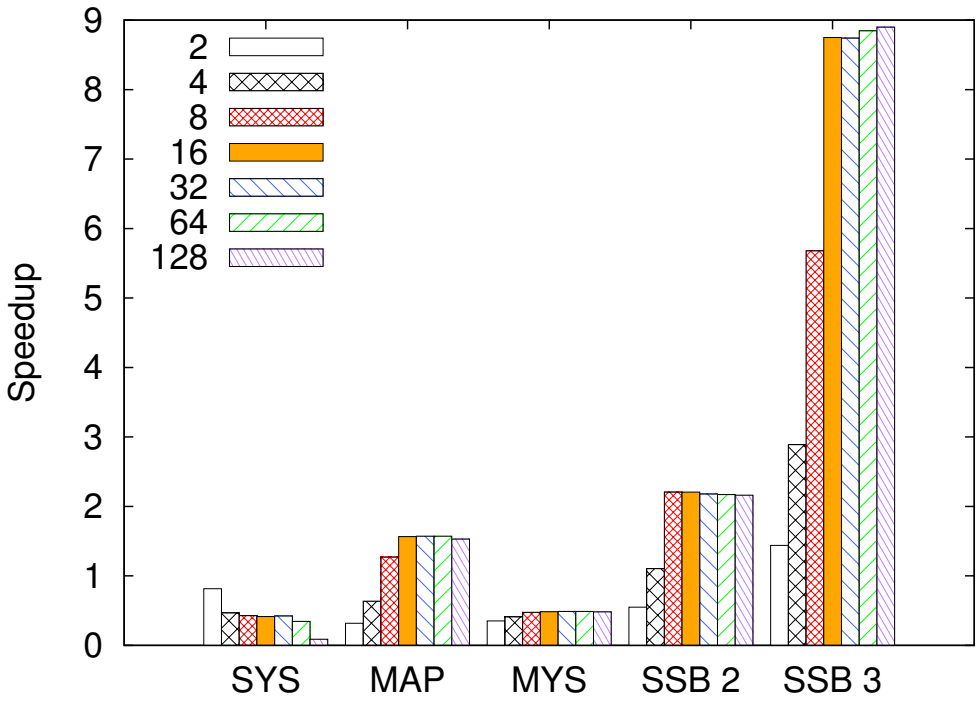
Figure 10.2: Synchronization Delay Illustration

10.2 SPEC OpenMP Kernel Loops

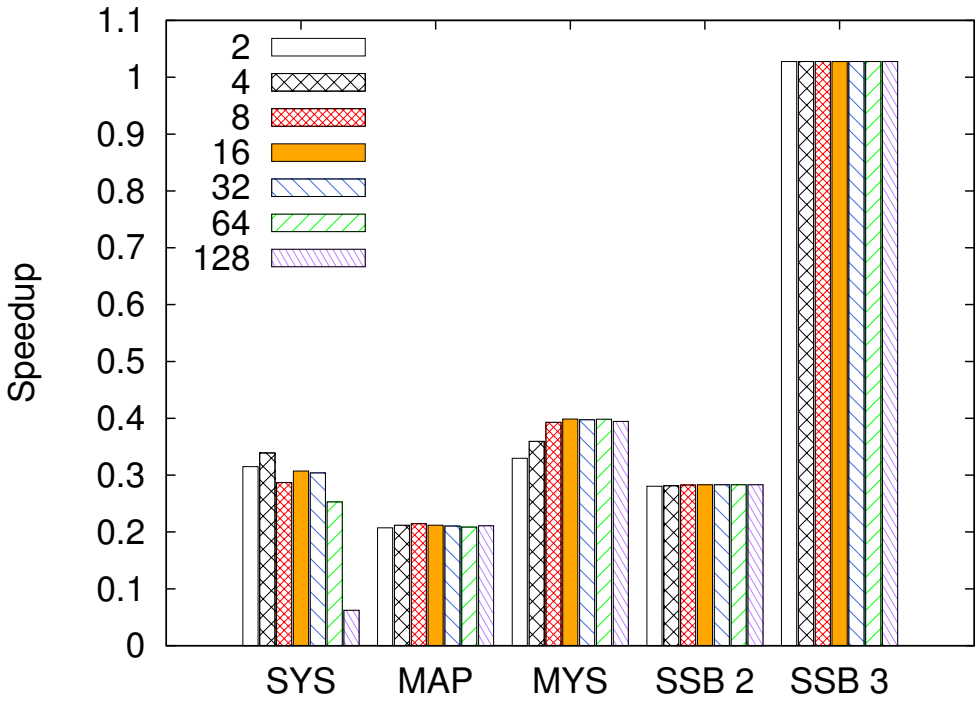
The kernel loops are extracted from SPEC OpenMP benchmarks, such as 314.mgrid and 318.galgel. As in the original SSB paper, this work compares the E-SSB versions against the software-based approaches proposed by Kejariwal et al. [47]. All loops exhibit the same characteristics, namely, that dependencies between loop iterations are positive and constant. They also fulfill the requirement of Single-Writer-Single-Reader, such that the E-SSB synchronization constructs can be applied. Figure 10.3 shows the speedup of the different parallel versions against the sequential version. E-SSB 3 clearly outperforms all other versions, both software and hardware based. Another interesting aspect is that there is no performance loss when the number of threads increases. K1's and K2's speedups are severely limited, but this is understandable and expected. K1 only performs a single arithmetic operation in the loop and therefore the speedup is clearly limited by it and the only form of parallelism can be obtained from the number of iterations that can be performed in parallel without dependence. K2's story is even worse, because the iteration dependence is 1. That means none of the iterations can be performed in parallel. Nevertheless, E-SSB 3 is still able to obtain instruction level parallelism between iterations through its fine-grain non-strict behavior and does not suffer any performance degradation as the other approaches. K3, K4 and K5 do not only provide sufficient iteration level parallelism due to a larger dependence distance of 8, but also a larger kernel that provides a great source of cross-iteration instruction level parallelism that can only be leveraged by E-SSB 3.

10.3 Analysis Breakdown

This section takes an in-depth look at the different versions of the tested wavefront benchmarks. This in-depth look consists of breaking down the collected information into different important activities and overheads such as cycles spent on useful work, synchronization overhead, loop overhead, arithmetic stalls, and stalls due to synchronized and unrelated memory operations, among others. The program tracing feature on the emulation engine is enabled to obtain detailed traces of all 160 thread

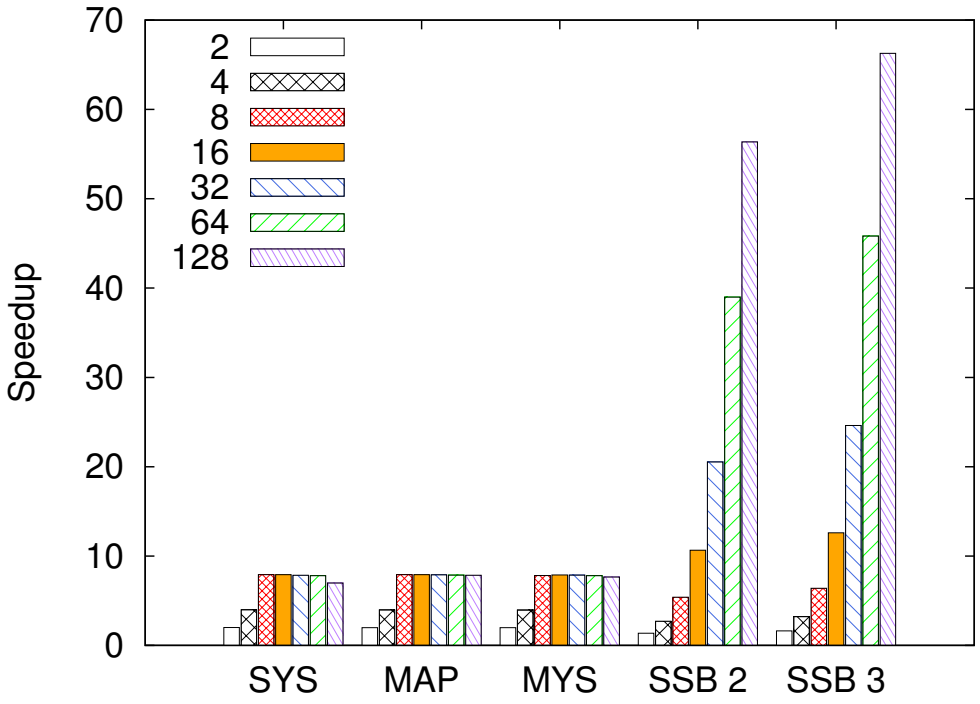


(a) K1

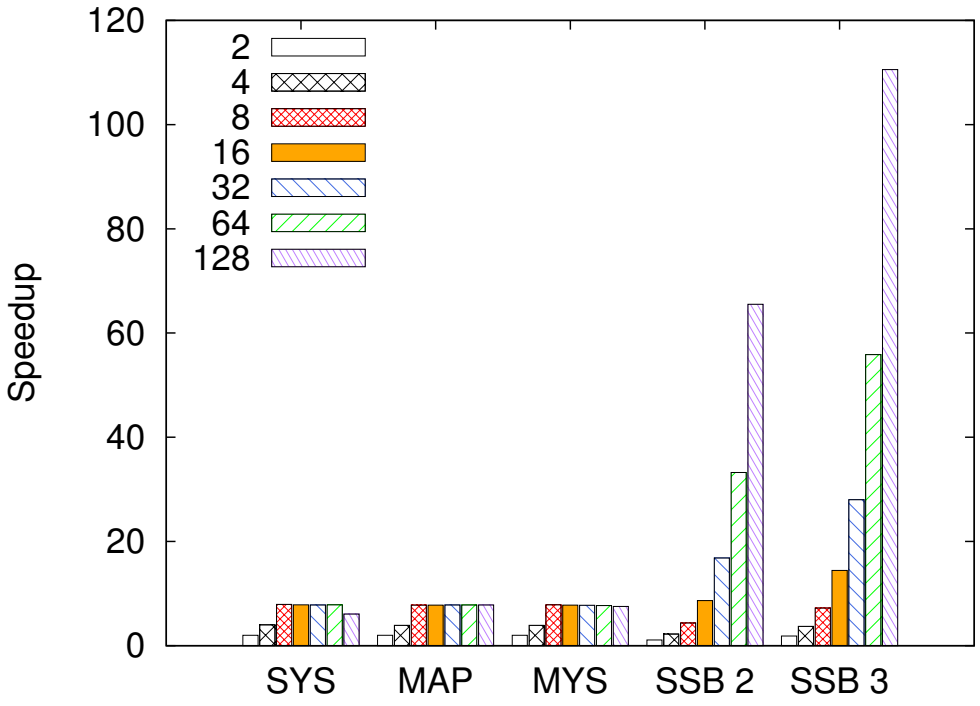


(b) K2

Figure 10.3: SPEC OpenMP Loops Speedup

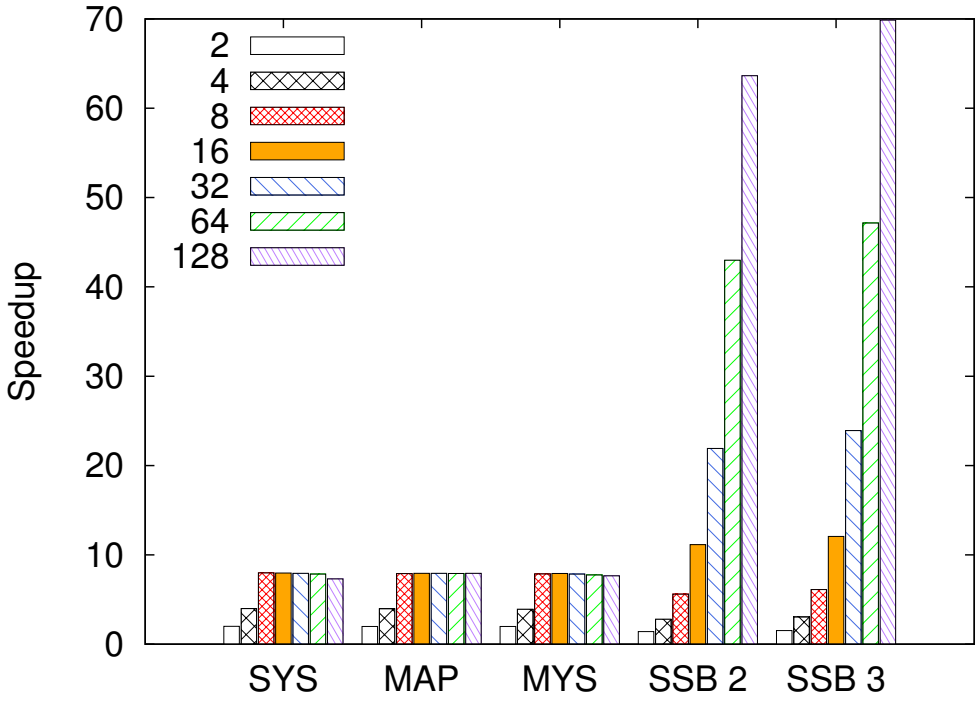


(a) K3



(b) K4

Figure 10.3: SPEC OpenMP Loops Speedup (continued)



(c) K5

Figure 10.3: SPEC OpenMP Loops Speedup (continued)

units instruction mix. Figure 10.4 shows a break down of the instruction mix of the different benchmark versions. The maximum problem size (512x512) for the benchmark is used to fully utilize the whole system. The histogram shows the accumulated cycles spent by all thread units to complete the work. The serial version uses of course only one thread unit, whereas all the other versions use 159 thread units. To obtain the actual execution time, each version has to be divided by the number of threads used. "Work" contains all instructions necessary to perform the actual required computation. This includes the arithmetic instructions and the memory operations to obtain the data. "Stall Arithmetic" contains all the stall cycles in the kernel due to dependence on an unfinished arithmetic instruction. "Stall Arithmetic" can be seen as part of "Work", because it depends on the given schedule and the arithmetic instruction latency of the architecture. "Loop Overhead" contains pointer increments, loop exit checks, branches and branch delays. "Stall Memory" contains all the stall cycles inside the kernel due to instructions waiting on data to return from memory. "Synchronization Overhead" contains all the additional instructions, stalls and branch delays which are required to perform the actual synchronization. "Function Overhead" contains all the instructions, stall cycles, etc, which are not part of the kernel code, but the surrounding setup code of the function. "Stall Misc" contains stalls due to the fact that the crossbar port and the floating-point unit are shared between two thread units and other architecture related stall cycles. "Function Overhead" and "Stall Misc" are so small (less than 1%) that they are omitted in the figure.

The instruction mix break-down, overall, is predictable: "Work" and "Stall Arithmetic" are uniform across all benchmarks. "Loop Overhead" should also be rather constant across all benchmarks, but signal-wait and barrier have a much higher loop overhead than the other versions. A detailed analysis shows that this increase is solely due to branch delays for the signal-wait version, because the loop does not fit completely in the instruction prefetch buffer. The barrier version suffers from the same problem, but it also has more complex loop exit checks on top of that. The barrier version also suffers a lot under a very large amount of memory related stall cycles. This is

due to the pathologic nature of programs that use barriers, which have normally three distinct phases. During phase one, all threads try to obtain the data at the same time, followed by the computation phase with almost no memory operations. Finally, in the last phase the results are written back to memory. This behavior makes the first phase a memory bound problem, which is responsible for the increase in memory related stall cycles.

Due to the factors described above, the main components that determine performance are the "Stall Memory" and the "Synchronization Overhead". The E-SSB 3 method shows no "Synchronization Overhead", because these cycles are hidden in the memory stall cycles. Thus, for a fair comparison, both of these components must be used together to evaluate our synchronization methods. Under these conditions, E-SSB 3 still clearly outperforms all other methods, even the highly optimized signal-wait version.

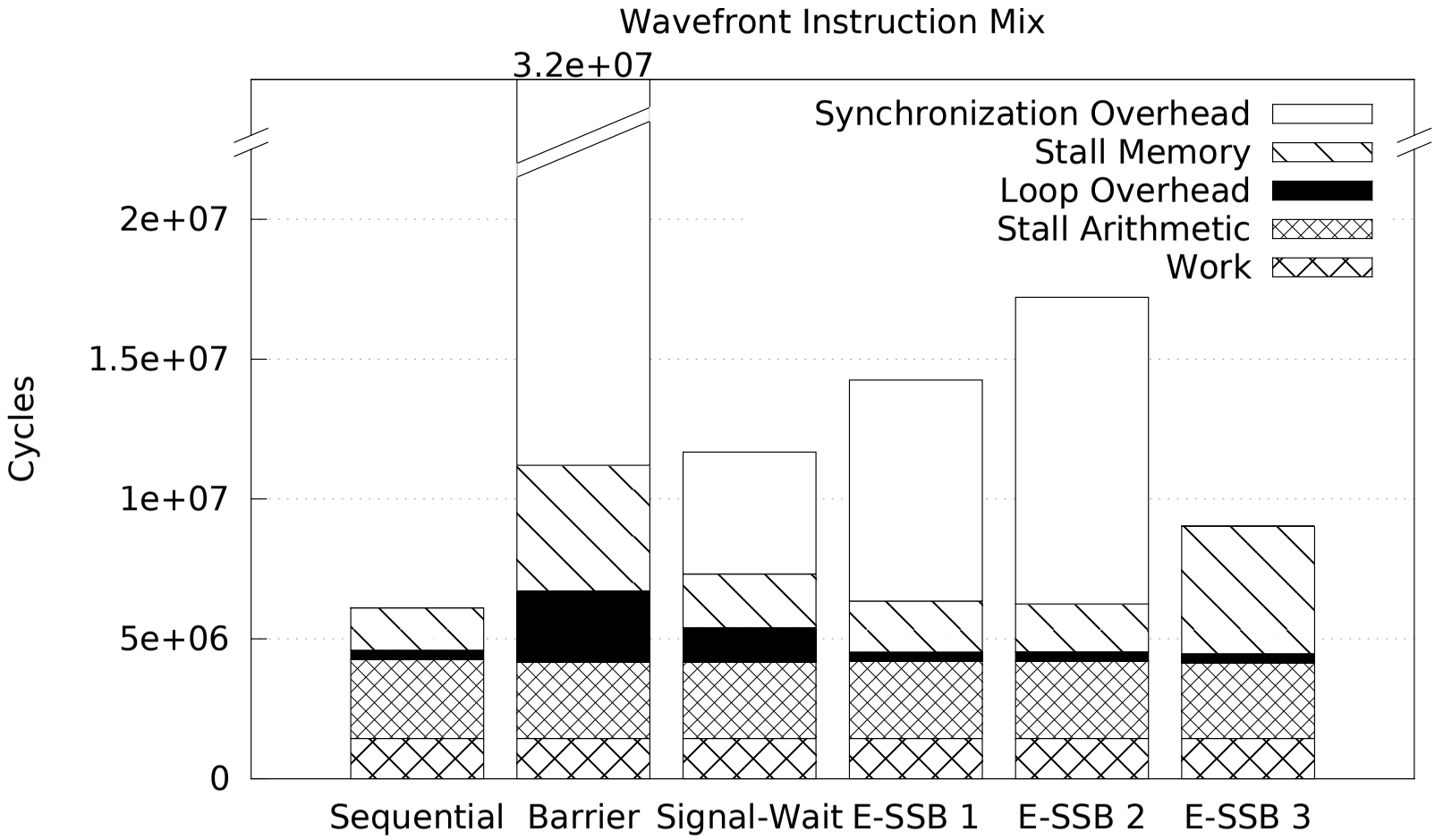


Figure 10.4: Wavefront Execution Runtime Breakdown: This histogram shows a breakdown of cycles spent on certain important aspects of the program. The Sequential version shows cycles spent by a single thread, whereas the other versions show the accumulated cycles spent by all 159 threads.

Chapter 11

INTEL'S CONCURRENT COLLECTIONS (CNC)

Intel's Concurrent Collections (CnC) is a programming model that extends existing (sequential) programming languages to turn them into inherently parallel programming languages. This greatly simplifies the programming for the user, because they do not have to add parallelism to their programs or design for parallelism. CnC adds new constructs and certain restrictions to a programming language to achieve this. As long as the programmer follows the rules and writes valid CnC programs, parallelism is inherent and data-race free, deterministic programs are guaranteed. Furthermore, CnC strictly separates the specification of the parallel program, called CnC Specification Graph, from the tuning. This greatly simplifies the process of writing applications for several architectures or retargeting applications to a new architecture. Ideally domain experts only concentrate on optimizing their algorithms without a particular architecture in mind, while tuning experts do not require any knowledge of the algorithm to perform their optimization task. CnC provides three new constructs to a programming language: Step Collections, Item Collections and Tag Collections.

- Step Collections: A step collection is a set of step instances of a particular step. A CnC program can consist of one or more steps and each step has its own step collection. A step is like a blueprint - it has all the information how a step works, but it cannot be executed or run. It requires a tag collection - the factory - that uses the blueprint to create a step instance with a specific tag - serial number - assigned to it. A step instance does not have any visible state and its outputs only depend on its inputs. This is also known as a pure function. Since the output only depends on the inputs and the inputs are always the same for a given tag, the whole execution of a step is deterministic. This requires that all "get" operations (reading data from an item collection) have to be executed before the first "put" operation(writing data to an item collection).

- Item Collections: In an item collection each tag has a unique immutable value assigned to it once it has been "put" into the item collection. Subsequent "put"s to the same tag are not allowed and every "get" will always return the same value for the same tag. Item collections guarantee that only values can be obtained that have been put there before. If a step instance tries to obtain a value that has not been "put" yet, then the step instance is suspended until the data element becomes available. Item collections enforce the order between steps - the when, but not the if.

- Tag Collections: Tag collections are the control part of every CnC graph/program. For each tag that is placed into the tag collection a step instance for each prescribed step is created. A tag collection can prescribe more than one step collection. A tag collection defines if, but not when a step instance is executed.

The combination of only these three constructs allows the creation of correct and valid CnC programs. The CnC runtime knows everything about the static connection of components of a CnC graph, but it does not know anything about its dynamic behavior. As a result, the performance might vary with each run of the program. The runtime needs further information to better utilize the resources available to it and to guide the execution of the whole application. This information is provided in the form of a tuning specification. Each of the three basic building blocks of the CnC language can be optimized by this tuning specification without changing the original CnC graph of the program. There can be several different tuning specifications for many architectures, using all the same CnC program graph specification. The tuning specification can, for instance, specify how to partition the tag space, where and when to schedule the step instances, which data storage to use, etc. The current tuning language still requires a human to create it, but it should be already possible with today's compiler technology or other tools to generate this tuning specification automatically. The tuning language is currently rather simple, but already very powerful in its effects. Future architectures will require an even more powerful tuning language. This chapter focuses on the tuning feature for item collections. The new proposed approach for meta-data tracking can be enabled by using the new dense data item collection type.

Chapter 12

DATA AVAILABILITY TRACKING IN SOFTWARE

With the advent of current many-core architectures new possibilities and great challenges are on the horizon. These new challenges have awakened new interest for fine-grain, asynchronous execution models. One such challenge is the need to efficiently keep track of the availability of data. The heterogeneity and dynamics of new many-core architectures exacerbate this problem for a number of reasons: (1) reassignment of cores to other programs, (2) change of priority, (3) power, (4) heat, and (5) hardware failures, to just list a few. As a result, future systems will contain an ever changing architecture even during the execution of a single program. This requires a more flexible software stack from the programming language all the way down to the operating system, which is capable to adapt during execution. A fine-grain, asynchronous execution model provides a base framework for dealing with such challenges dynamically, but there are still fundamental open questions that need to be solved. Among them is data book keeping (meta-data which is used to keep track of the availability, location and other intrinsic properties of the data block like access pattern, footprint, etc).

The seemingly simple task of keeping this metadata up-to-date to ensure a semantically correct program execution is still an ongoing research problem even today. Keeping track of the availability of data seems to be one of the hardest tasks that the bookkeeping system must take care of. There exist hardware solutions that stem from the dataflow world which solve this task with very fine grain-level data. However, the associated hardware costs prevented wide adoption in general purpose computing. Some of these hardware solutions are the I-structure [20], the full/empty bit of the Tera MTA/Cray XMT machines [43], and the Extended-Synchronization State Buffer (E-SSB) [48] as described in the previous chapters. Current software solutions are

adequate (but far from optimal) or not suitable for general use without hardware support (e.g. Software Transactional Memory). They have been originally designed for smaller systems which could be easily statically scheduled and partitioned. These solutions become ill adapted or right out unusable when scaling out from hundreds to thousands of cores because of the dynamic behavior that these systems exhibit.

Thus, static approaches will not work well with such dynamic systems and a more flexible approach is required that allows for example hierarchical tiling and dynamic partitioning and therefore variable tile sizes during runtime.

This chapter propose a new dependency tracking method and describes its implementation as a proof-of-concept in the Concurrent Collections (CnC) programming language [36] from Intel. Although this method is investigated under the CnC model, this method should be applicable to other fine-grain parallel programming and execution models.

12.1 Problem Formulation

Many real world problems consist of a large amount of dense data. In such problems, keeping track of the data's different states becomes cumbersome when a large number of fine grained actors are involved. Moreover, the efficient management of such states is especially important for dynamic runtime systems in which optimization parameters are updated during the execution of the target application. Such runtimes require an efficient method to keep track of the new/available data as the computation progresses. Furthermore, a fast query method is needed so that this information can be collected and processed by the runtime internal optimization engine to steer the computation for a given set of constraints (i.e. power, performance, memory footprint, etc). An example of this is to efficiently query and update n-dimensional tiles of arbitrary size in applications and architectures in which locality can greatly affect the total execution time. In summary, how to implement an efficient data tracking and querying system is a major concern among dynamic runtime system writers and their users.

12.2 CnC Item Collections

The original item collection provided by CnC is a hash map based container. That means every insertion or retrieval of a data element requires the usual access procedures and overhead of a concurrent hash map. As long as the work performed in a step instance and the size of the data stored for every data element is large enough, the overhead of an access to the item collection is negligible. If there is a dense distribution of tags it is possible to use a vector based item collection[1]. This eliminates the cost of the concurrent hash map to access an element, but it does not reduce the meta-data associated with each data element.

One way to alleviate this problem is to tile the dense data and work on tiles instead of single data elements. This is in most cases the desired way of working on dense data arrays anyways. Unfortunately, this requires the programmers to perform this task by themselves and it also locks them into a fixed tile size. Moreover, since the tile size is now part of the CnC program specification and not the tuning specification, it violates the idea of separation of concerns and it makes the tuning for different architectures more difficult.

Another aspect is the requirements of different steps working with the same item collection. The tile size requirements might be different, as we will show in the gaussian blur filter example in Section 15.2, and a fixed tile size does not fit all. In the following chapter I propose a new data structure that meets the needs of arbitrary tile sizes with low storage overhead for meta-data.

[1] new construct as of CnC version 0.7

Chapter 13

RD-TREE

In this chapter a tree-based data structure - the RD-Tree - is proposed to keep track of the availability of data and also to store the data itself. The data structure has to provide the following properties and capabilities:

- Single Assignment: Each data element can only be assigned once and the value is immutable thereafter. Multiple assignments to the same data element have to be recognized by the data structure and signaled back to the user as errors.
- Availability: Provide information if a data element has already been written and is available.
- Serve Data: Provide the requested data if and only if all elements are available
- Dependency Tracking: Keep a list of suspended steps that are waiting for data to become available and resume them once all their requested data becomes available.
- Memory Management: Keep a get counter of the data elements and deallocate the data if no longer needed.

13.1 Data Structures

The basic data structure the RD-Tree is working with are called tiles. There are two type of tiles - data tiles and waiting tiles. Data tiles contain the actual data, a n-dimensional bounding box, and a get counter. The bounding box describes the location and shape of the data in n-dimensional space. The get counter indicates how often the tile will be read. The waiting tiles does not contain any data, because they serve a different purpose. They describe on which data a particular step instance is waiting on. They also use the same bounding box to describe the requested data and a counter indicating how many outstanding elements are left.

The tree itself uses two data structures for its nodes - internal nodes and leaf nodes. Internal nodes have two child pointers that can point to either inner nodes or leaf nodes. Furthermore there is a selector that indicates the dimension this node is splitting, a upper left bound, and a lower right bound. Leaf nodes are just fixed size containers that hold tiles.

13.2 Splitting Strategy

Initially the root of the RD-Tree is just a leaf node. Once that node has been filed up with tiles the node has to be split. The partition strategy only looks at a single dimension to split the node. This allows the fanout to be independent of the number of dimensions. Although the RD-Tree only allows single assignment of data elements, the bounding boxes of waiting tiles can overlap with other tiles. This makes it sometimes impossible to find a single point to split the tiles into two distinct regions. Instead an upper left bound and a lower right bound are used to describe two regions that can overlap. The split algorithm searches all dimensions first and selects the dimension with the smallest overlap.

13.3 Insertion Algorithms

The insertion algorithm starts at the root of the tree. If the root is a inner node it checks if the tile fit either in the left or right subtree. If it fits in neither, then the tile is added to the subtree that creates the smallest increase in overlap. Although the tile is inserted only in one branch of the tree, the algorithm still might also visit the other branch of the tree. This only happens if both sub-regions overlap and the tile intersects with the other sub-region. This done to inform potential waiting tile that this data is available now. Therefore it is important to minimize overlaps as much as possible when splitting nodes and inserting new tiles. This algorithm is invoked recursively until a leaf node is reached. First all waiting tiles in the leaf node are checked if they overlap with the new data tile and updated if required. This update is very efficient in the RD-Tree due to its single assignment property. The algorithm has

only to check the intersection of the data tile's and waiting tile's bounding box and substract the number of intersection elements from the counter. The counter has been originally initialized to the number of requested elements. Should all data of a waiting tile have become available (that means the counter has reached zero), the suspended step instance is rescheduled and the waiting tile is removed from the leaf node. After all the wait tile have been processed the data tile is added to the leaf node. If the leaf node is full the splitting algorithm is invoked to split the leaf node if necessary.

13.4 Query Algorithm

The query algorithm initializes a counter with the number of requested elements and starts at the root node of the tree. If the root node is a inner node it has to check if the requested bounding box intersects with the left and the right sub-tree and traverse each subtree in turn. Once a leaf node is reached all data tiles that intersect with the requested bounding box marked in a work list and the number of intersecting elements is subtracted from the counter. After the algorithm has walked the tree and the counter has reached zero it knows that all the data is available. In this case it will create a new data tile and copy all the partial elements from the work list into that new data tile and return it. If not all elements are available a waiting tile is initialized with the counter value, the step instance and inserted into the tree.

13.5 Memory Management

This algorithm traverses the same way as the query algorithm the tree. In every leaf node it checks if a data tile intersect with the provided bounding box and decrements the get counter for that tile. If a tiles get counter reaches zero the data tile is deallocated and removed from the tree. If a leaf node goes below the minimum utilization threshold, then its tiles are inserted into the neighboring sub-tree and the leaf node and inner node are deleted. This could also lead to a split if the neighboring sub-tree's leaf node overflows.

Chapter 14

RD-TREE IMPLEMENTATION

To test the concept of the RD-Tree the CnC item collection is extended to allow the storage and retrieval of tiles, otherwise known as blocked ranges in CnC. For this purpose a new item collection tuner that is intended for dense data arrays and that uses the RD-Tree is added. A blocked range is a description of a dense linear n-dimensional tuple space in the terms of a start point and an end point. This implementation of blocked ranges is based on the idea of TBB's blocked ranges, but is much more flexible. It uses variadic templates to implement an universal implementation for n-dimensional rectangles. The programmer can request or insert any size n-dimensional blocked ranges. The item collection is responsible for enforcing data dependencies and element storage. This means that no read request can be completed successful before all the requested elements have been written.

Two new application programming interfaces (APIs) are provided to the programmer to use this new item collection: "put_range" and "get_range". There are two different versions of each interface. The first version returns a copy of the requested data in a C++ Standard Template Library (STL) container like std::vector for "get_range" or takes a STL container as input for "put_range" and copies the data to the item collection. The second uses the new move semantics of the C++11 standard and performs a destructive put of the data. This means that the data copy into the item collection comes for free, since no actual copy operation is performed. But this also means that the vector that originally contained the user data is empty now. In most programs this is not a problem, because the data is not used afterwards anyways.

Chapter 15

RD-TREE EVALUATION

15.1 Testbed

All experiments were performed on a 48 core system. The quad socket system is equipped with 12 core AMD Opteron 6234 processors (Interlagos) running at 2.4 GHz and a total of 128 GB system memory.

15.2 Gaussian Blur Filter

This benchmark is a Gaussian blur filter that is applied iteratively on an image. In this algorithm the new value of each pixel in the image depends not only on its own value, but also on its surrounding pixels. Each of the participating pixels is weighted based on the Gaussian distribution and summed up to the new value. To take advantage of the cache it is beneficial to tile the computation. To calculate one new tile, the computation depends not only on the data of the previous tile, but also on the data of the surrounding tiles. A fixed tile size can be used, but is not optimal due to the different tile size requirements. It might also have negative impact on the cache usage, since more data is requested than actually used. Another interesting aspect is the runtime overhead. Instead of requesting just a single tile, each step has to request a total of nine tiles before it can start its computation. If no tuning is used at all this could result in the suspension of each step instance for nine times, before it can finish its execution successful. This is a big scheduling overhead and it can be avoided by using tuning. But the runtime overhead of requesting nine separate tiles cannot be avoided with the currently provided CnC item collections. Here it shows that this new framework has several benefits. Even if no tuning is used, a step instance has only to be suspended at most once. With tuning the same result will be achieved, but at

lower cost, because there is only one pending request that has to be maintained and not nine. This also means that the runtime overhead of each step has been reduced, because only one request has to be processed by the item collection instead of nine.

Figures 15.1 , 15.2 and 15.3 show the runtime and speedup results for seven different implementations of the Gaussian blur filter benchmark. The first two version are sequential and do not use CnC at all. The first sequential version uses blocking to improve cache efficiency, but uses one big single array. The second sequential version uses tiling. The tiled sequential version is used as baseline for the speedup calculations. The third version uses the hash map implementation of the item collection to store every data element (one pixel) separately. This is a very inefficient approach and only shown for educational purposes. The fourth version uses the vector implementation of the item collection to store every data element (one pixel) separately. This allows the use of completely dynamic tile sizes, but every element has to be requested separately. CnC programmers are not encouraged to ever do this, due to the storage overhead and runtime overhead. The fifth version uses the hash map implementation, but instead of storing each element separately a whole tile is used now. The sixth version uses the vector implementation and also applies manual tiling. The seventh version uses the RD-Tree.

The third and fourth benchmark version perform considerably worse than the sequential version if only one thread is used, but the forth performs on par for larger tile sizes. Nevertheless, its scalability is severely limited and does not perform well with increasing number of threads. This was an expected result and as we mentioned above, this is not how this item collection should be used and it was never intended so by its creators. Although, these are the only version that allows dynamic tile sizes using the existing features of CnC.

The version that use manual tiling show similar performance numbers for all benchmarks. This shows that there is no significant difference between the vector item collection and the hash map item collection for well balanced steps. They are also as expected the best performing versions, since the tiling has been implemented manually

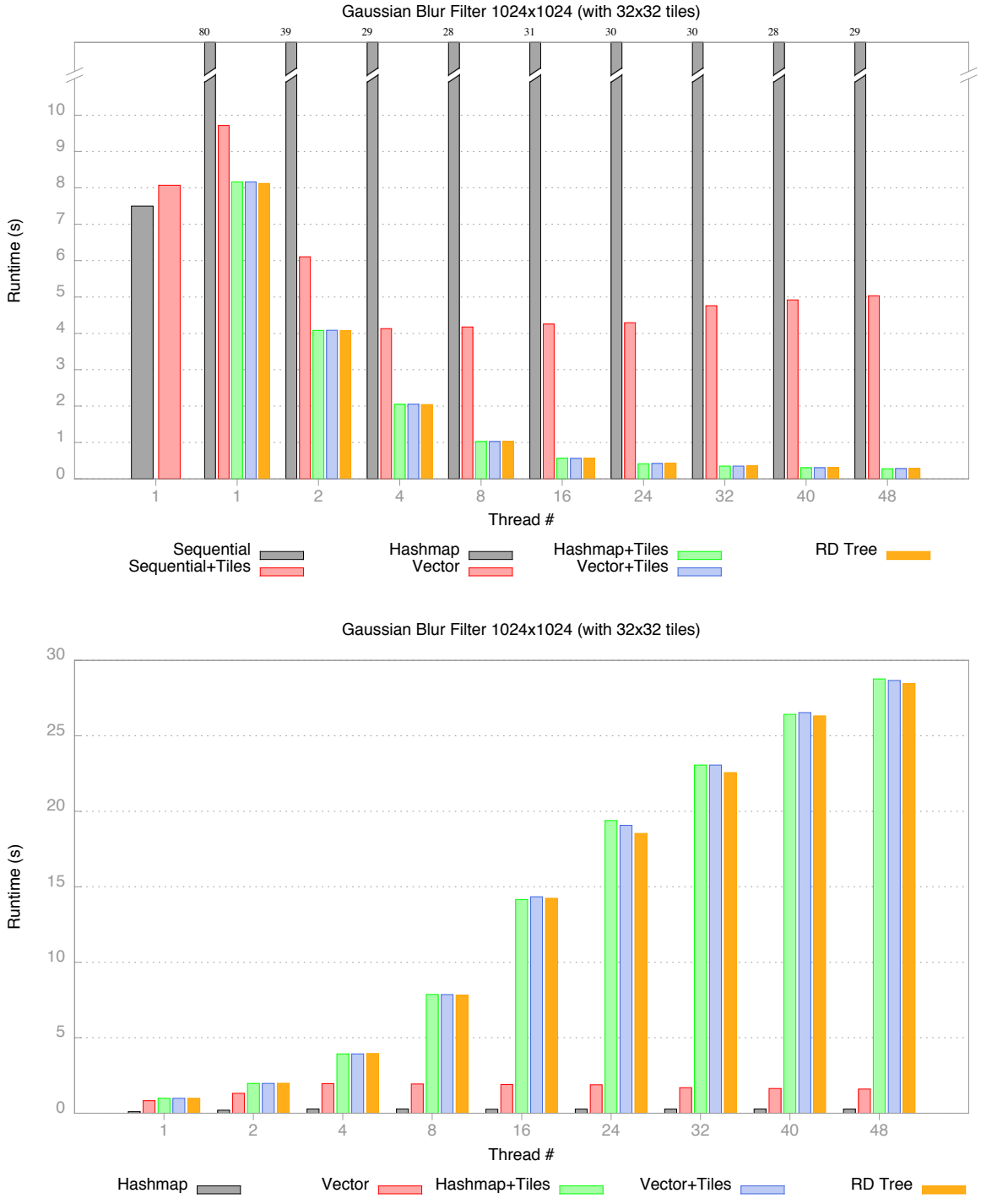


Figure 15.1: Gaussian Blur Filter Results for Problem Size 1024x1024

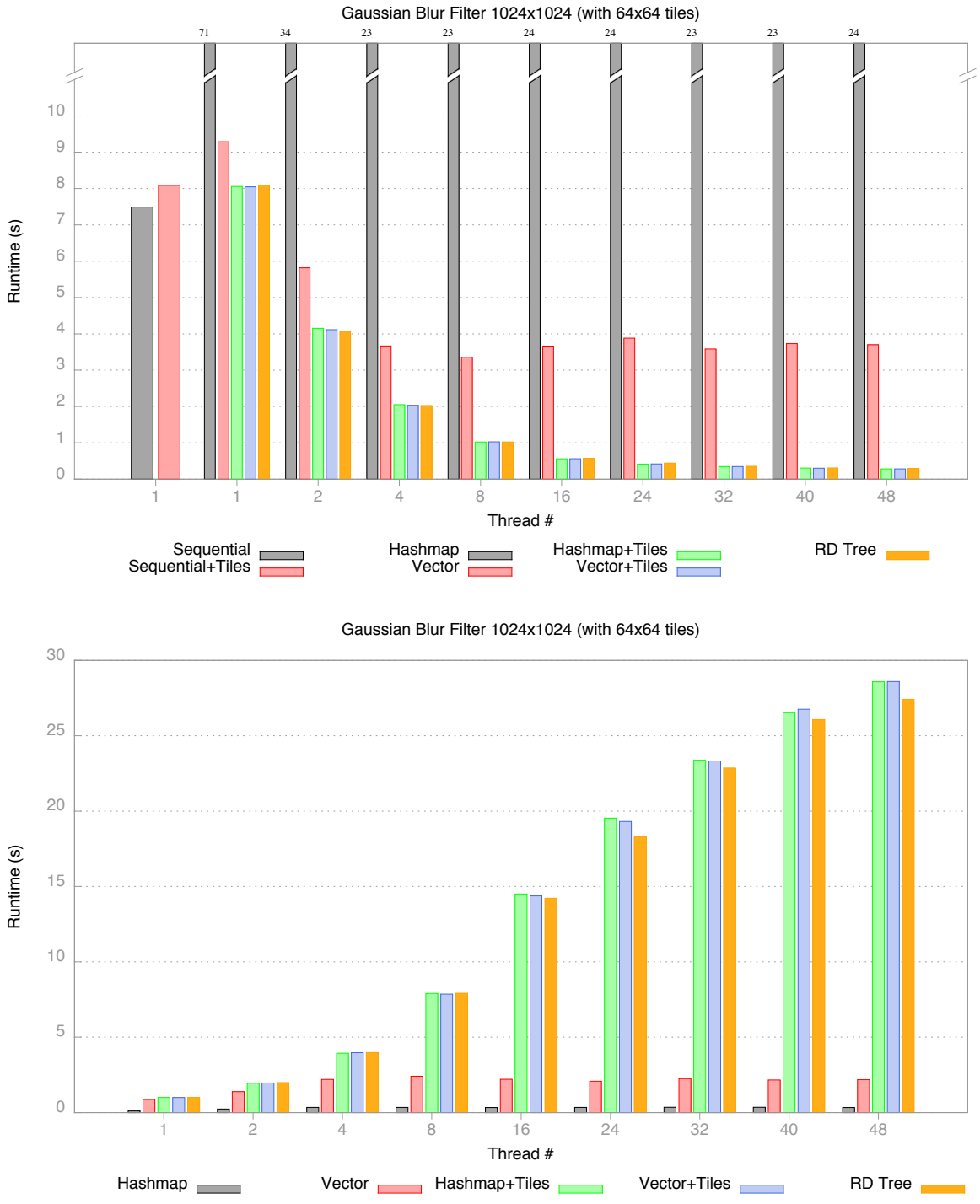


Figure 15.1: Gaussian Blur Filter Results for Problem Size 1024x1024 (continued)

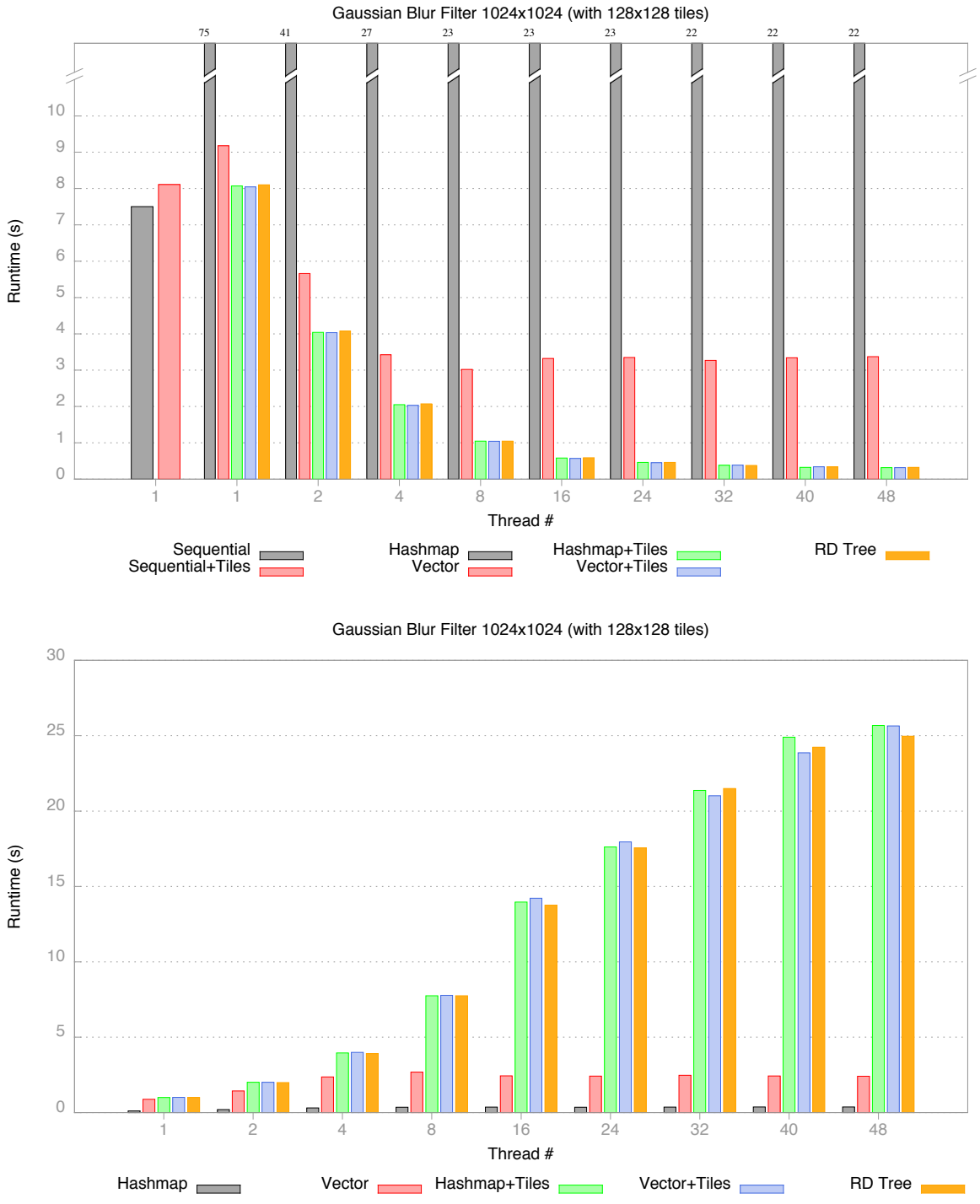


Figure 15.1: Gaussian Blur Filter Results for Problem Size 1024x1024 (continued)

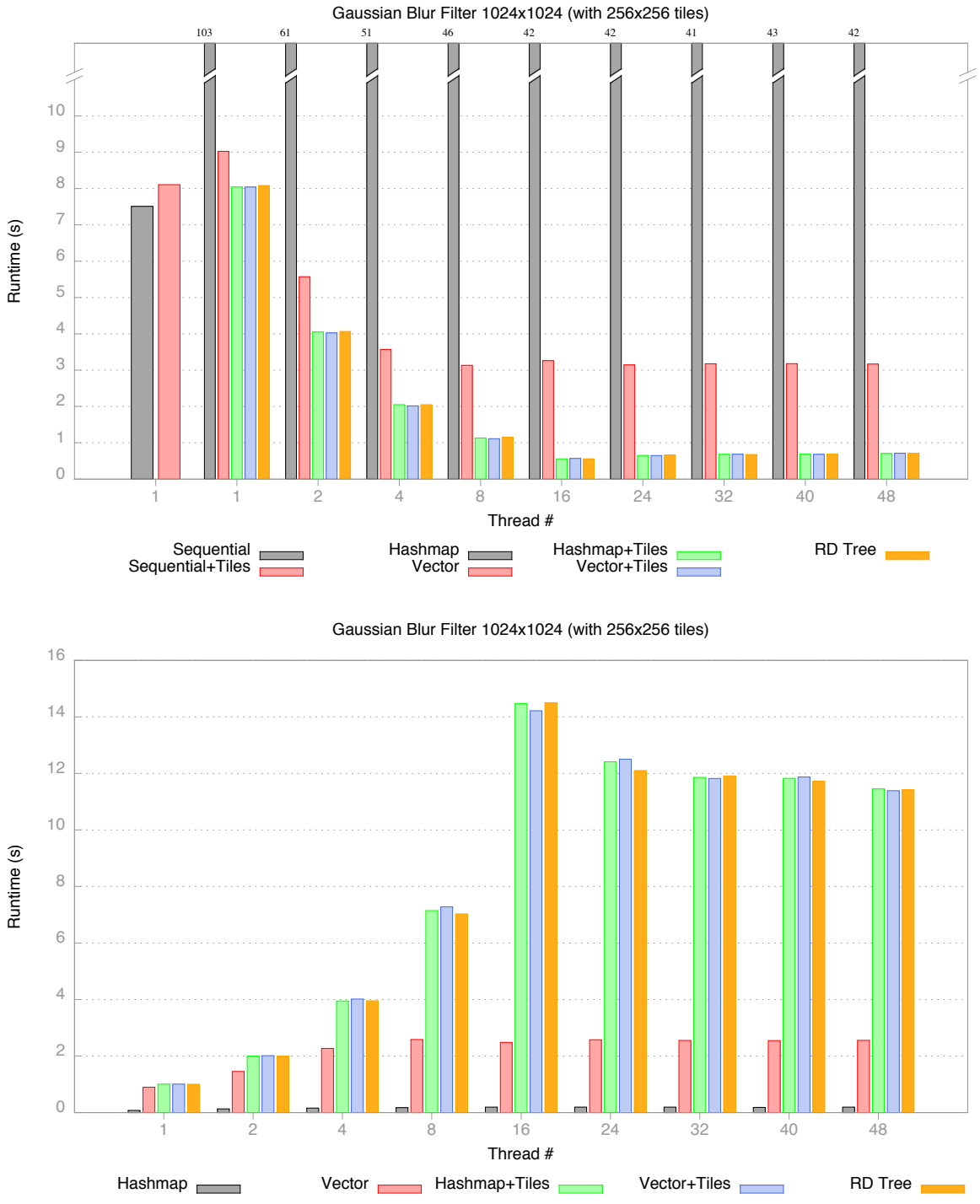


Figure 15.1: Gaussian Blur Filter Results for Problem Size 1024x1024 (continued)

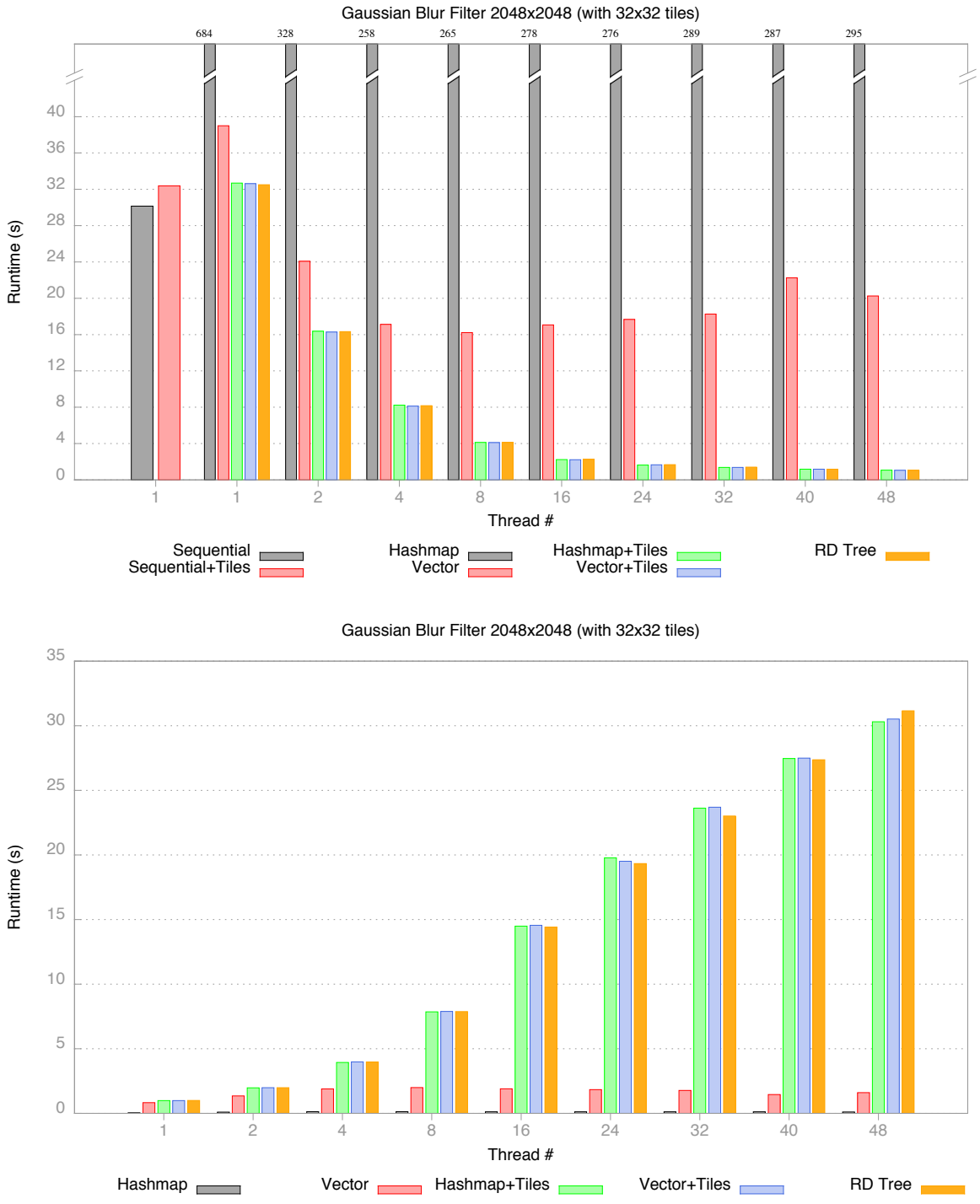


Figure 15.2: Gaussian Blur Filter Results for Problem Size 2048x2048

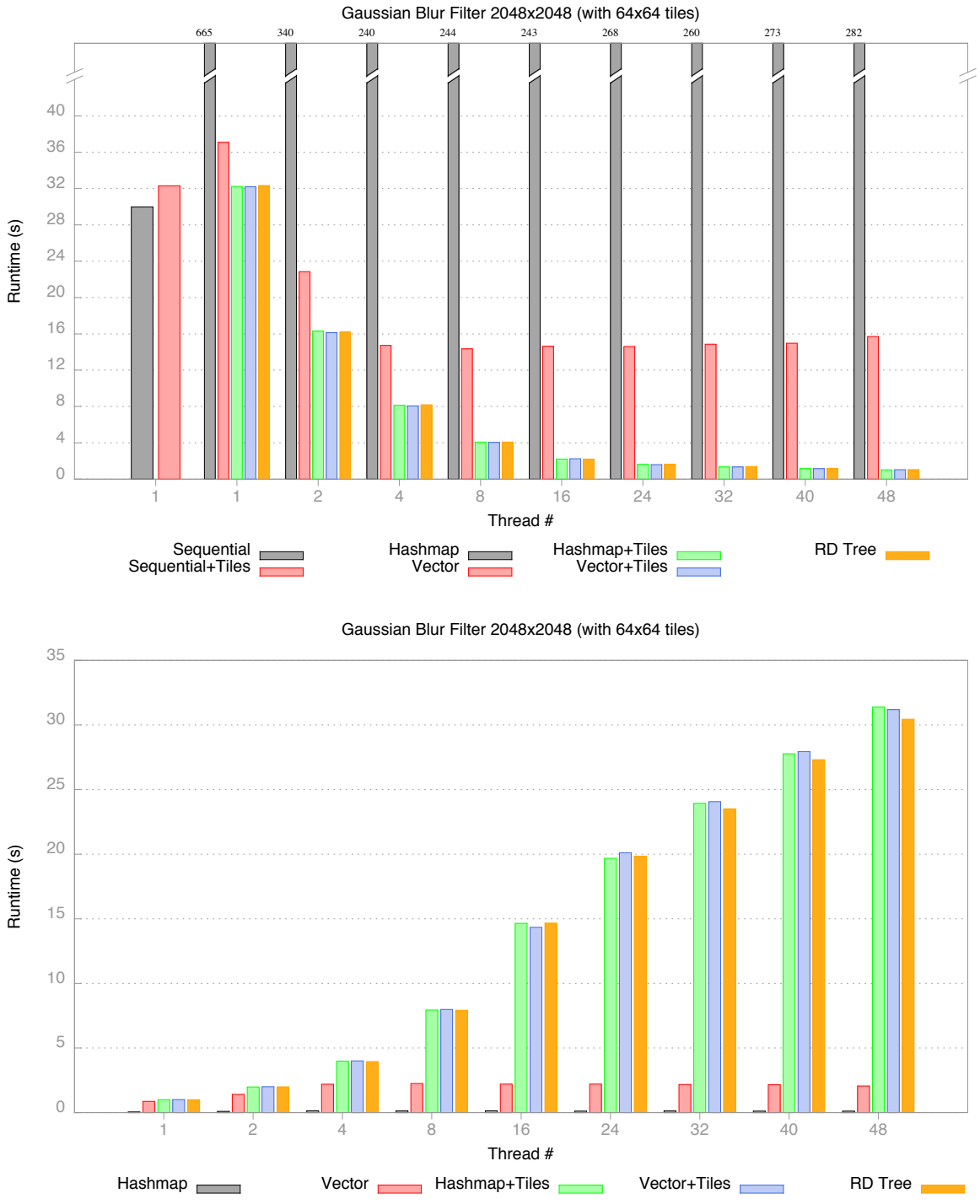


Figure 15.2: Gaussian Blur Filter Results for Problem Size 2048x2048 (continued)

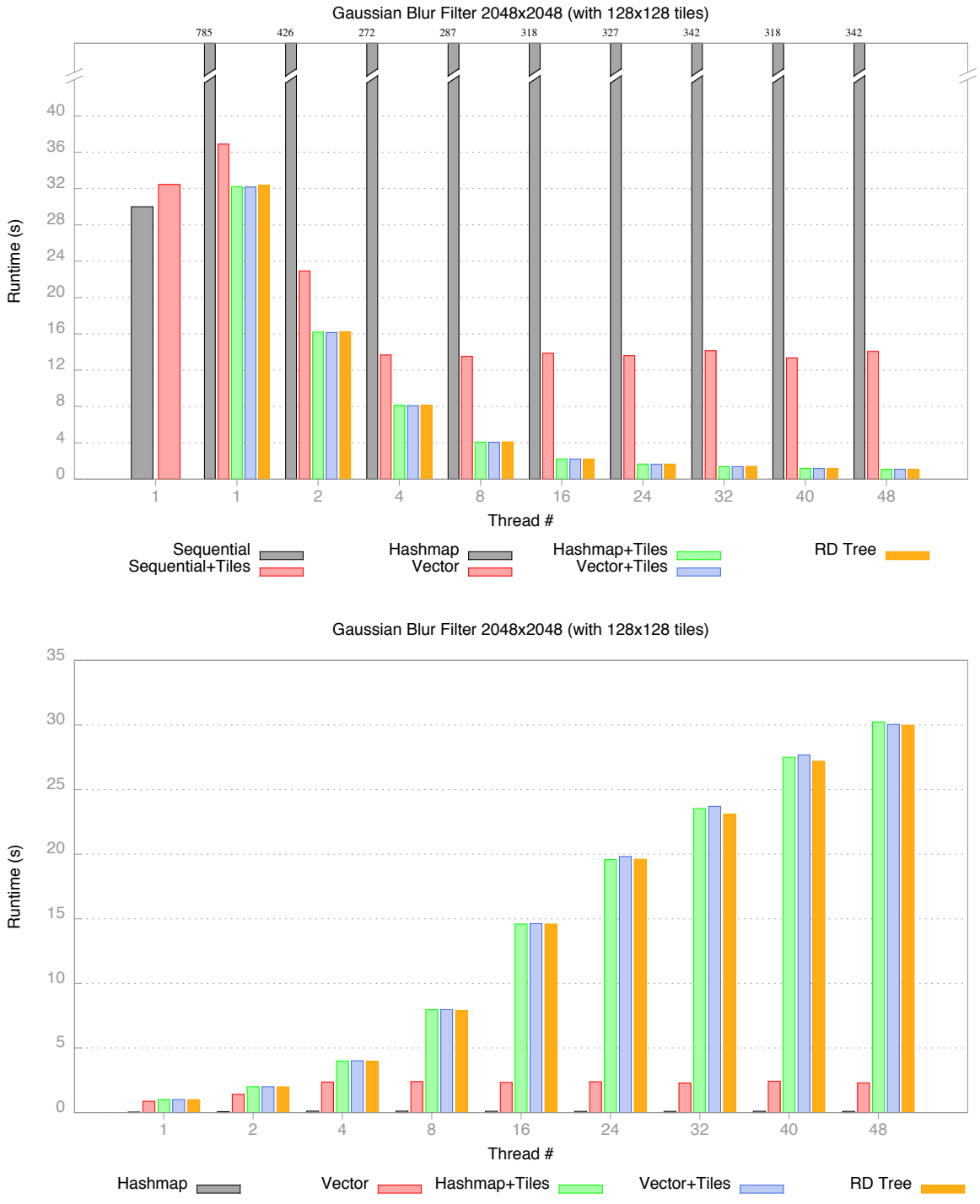


Figure 15.2: Gaussian Blur Filter Results for Problem Size 2048x2048 (continued)

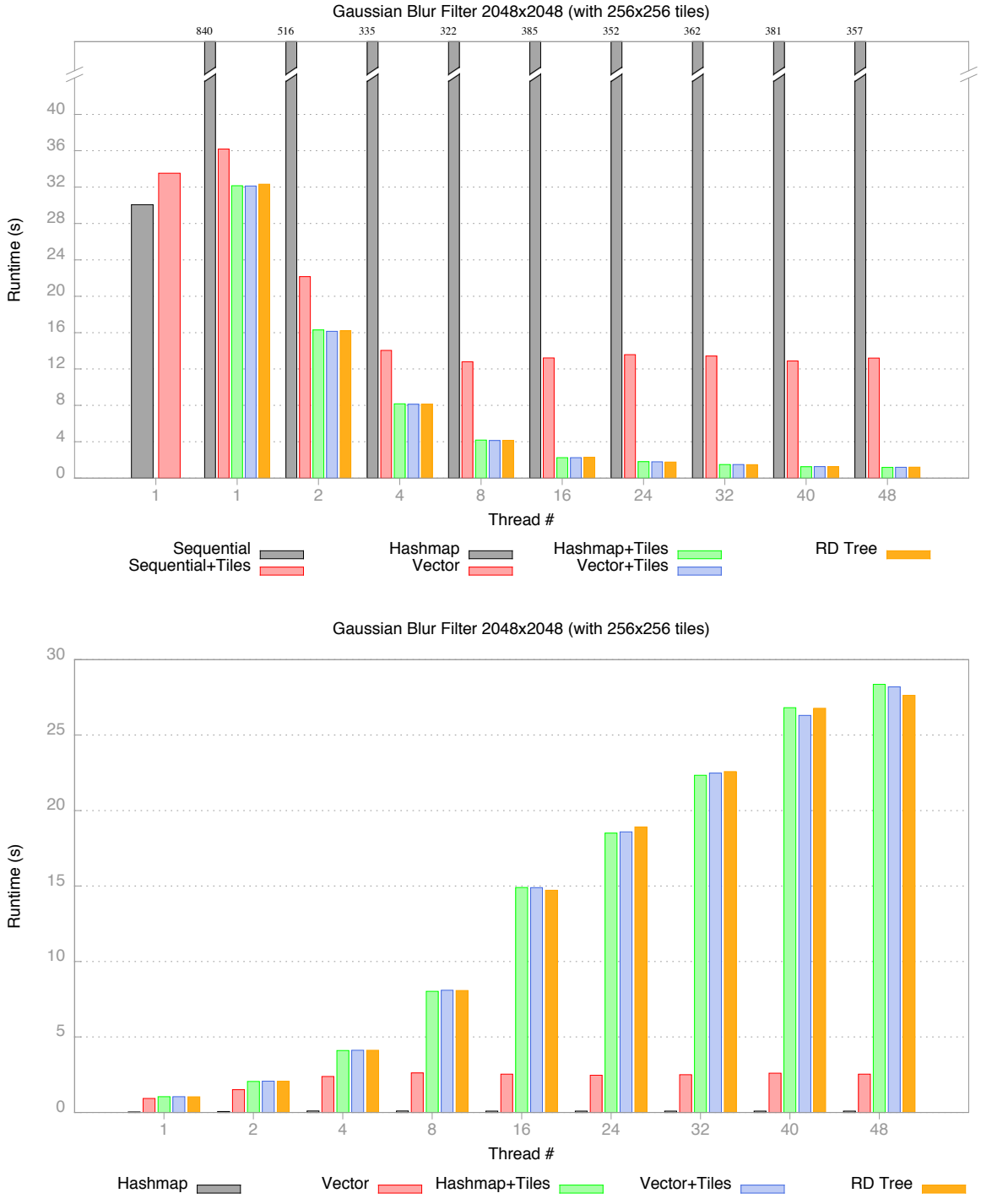


Figure 15.2: Gaussian Blur Filter Results for Problem Size 2048x2048 (continued)

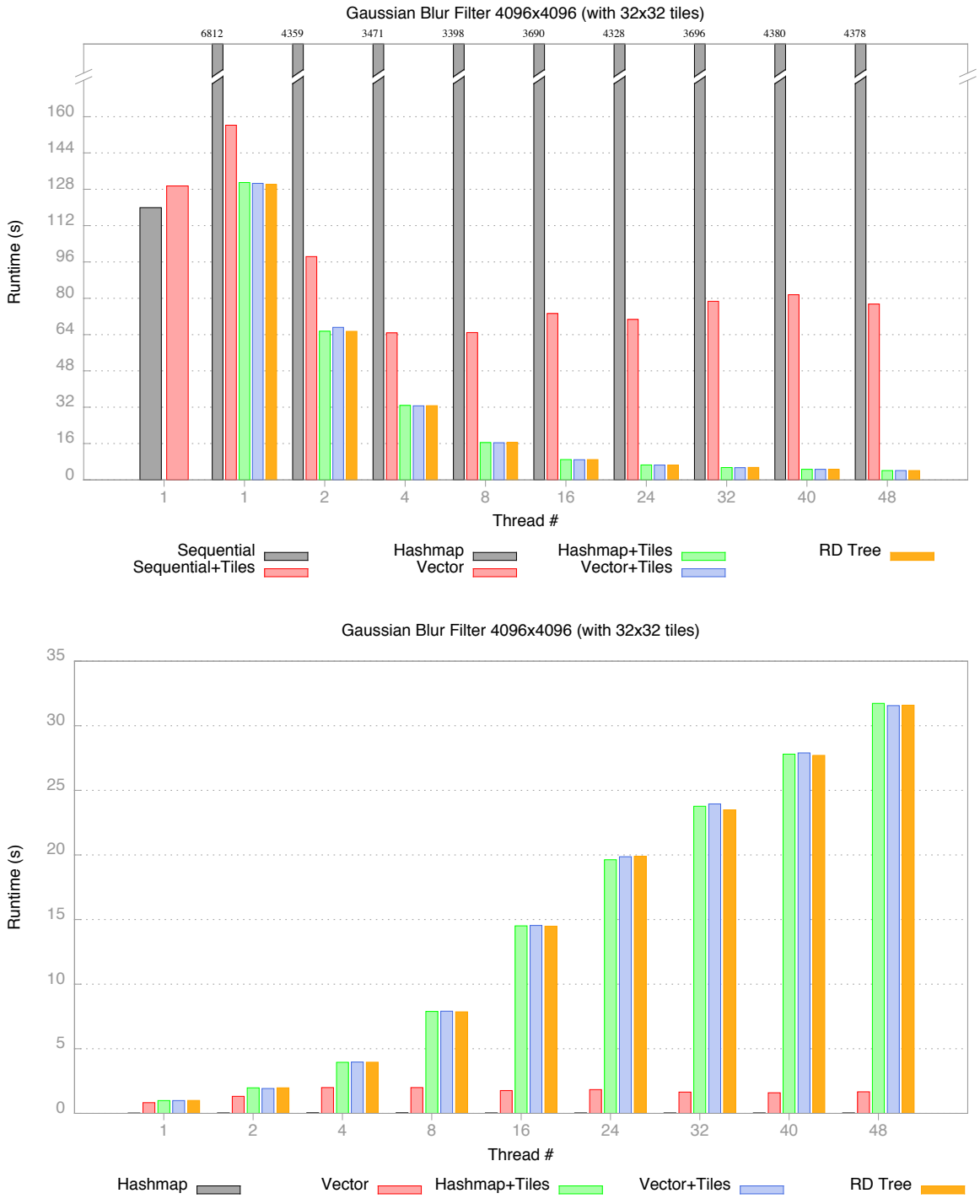


Figure 15.3: Gaussian Blur Filter Results for Problem Size 4096x4096

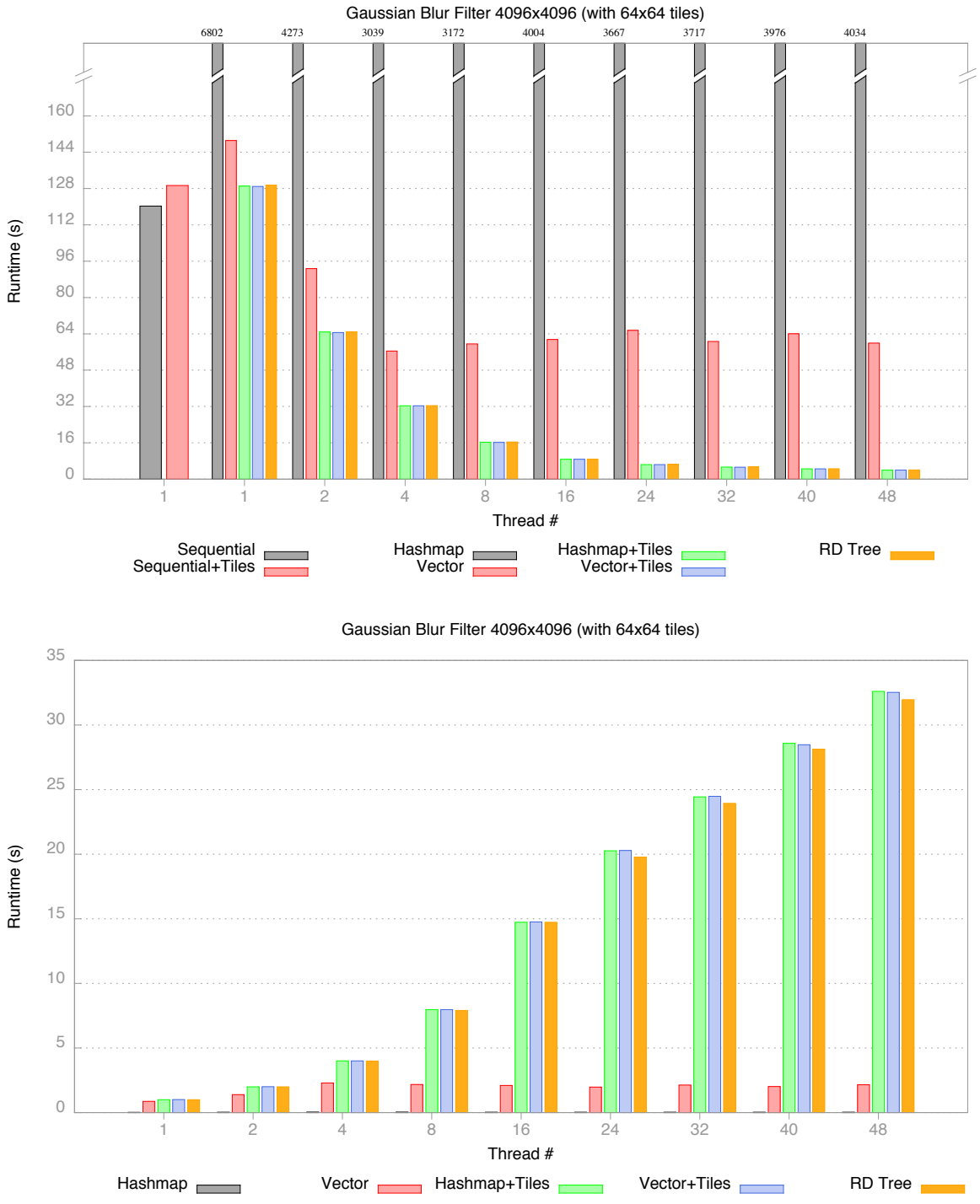


Figure 15.3: Gaussian Blur Filter Results for Problem Size 4096x4096 (continued)

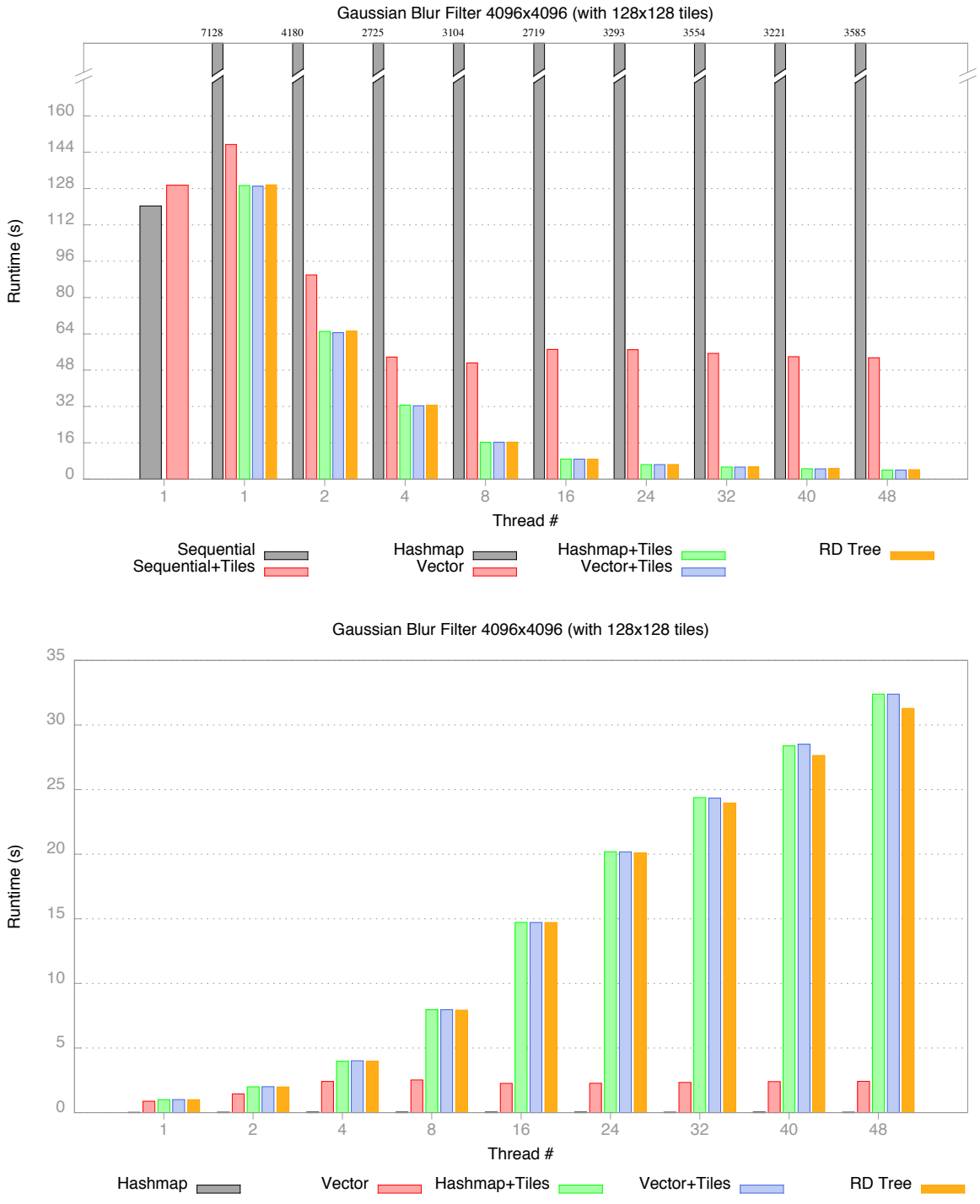


Figure 15.3: Gaussian Blur Filter Results for Problem Size 4096x4096 (continued)

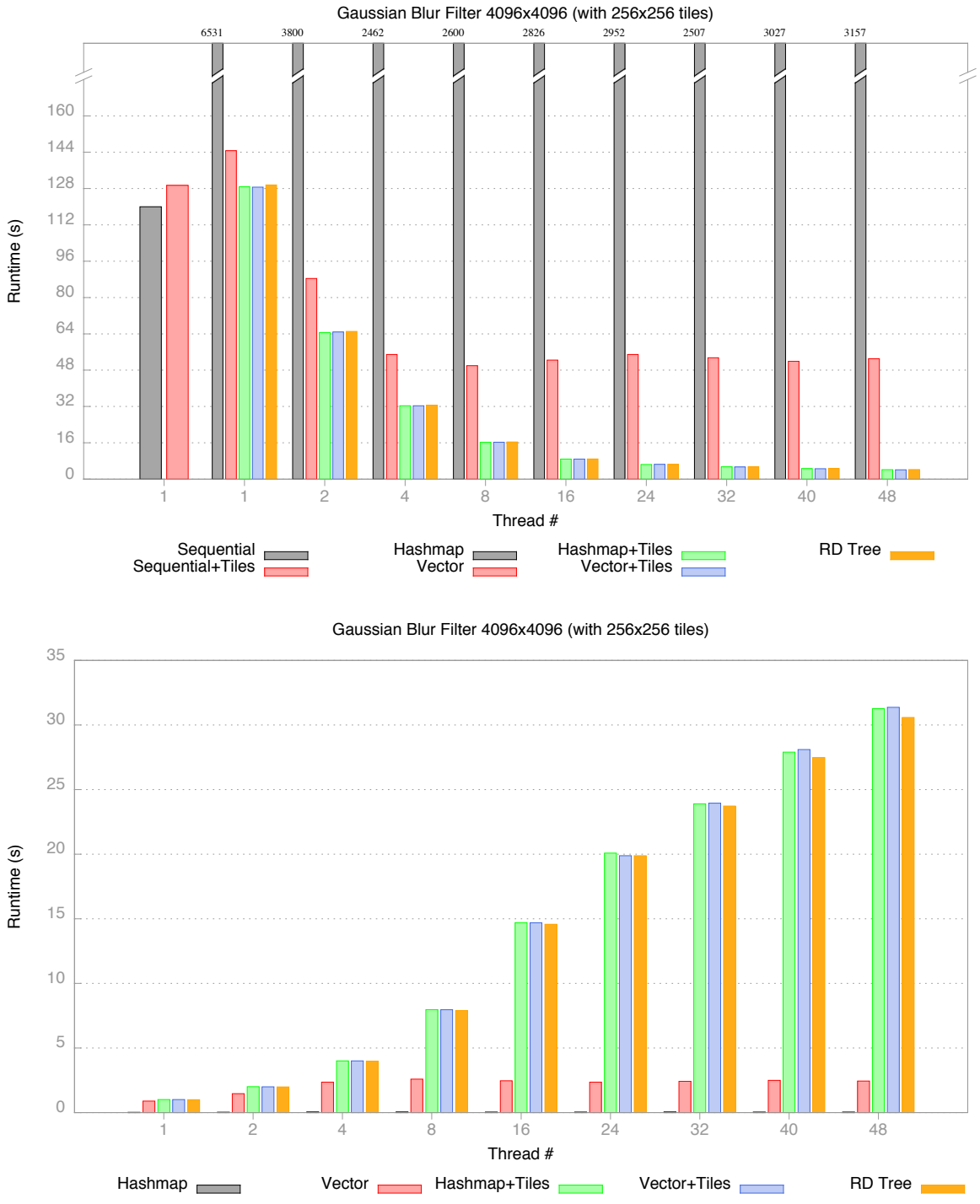


Figure 15.3: Gaussian Blur Filter Results for Problem Size 4096x4096 (continued)

and uses fixed tile sizes.

The performance of the RD-Tree version is comparable with the manual tiled versions and does not show any major regressions.

Chapter 16

RELATED WORK

This research for the hardware based approach was greatly influenced by previous work on fine-grain synchronization constructs by academia and industry. This includes research on dataflow constructs like the I-Structure [20], the Synchronization State Buffer (SSB) [42], and the Tera MTA/Cray XMT [25, 43]. The use of tagged memory, full/empty bits, and I-Structure have been explained in Chapter 2. E-SSB differs in the following aspect from previous work: It enables "virtual tagging" of the whole memory space like SSB; it also supports all data sizes of the architecture and it is not limited to double-word synchronization. Furthermore, it has been enhanced to support non-strict synchronization, which is the most crucial part in obtaining great and scalable performance. It has the benefits of SSB, which means using fewer hardware resources, and the non-strict behavior of I-Structures.

Another approach that gained momentum in recent years is Transactional Memory (TM) [49, 50], which also employs a non-blocking synchronization approach. The major difference with this approach is that in Transactional Memory, if a transaction fails, all changes done inside a transaction must be rolled back and the transaction has to be restarted. This results in unnecessary computation every time a transaction has to be restarted. The fine-grain in-memory synchronization approach does not require this. The Transactional Memory approach works very well if there is little to no conflicts and transactions don't have to be restarted. If it is used for heavily concurrently accessed memory regions this approach falls apart and transactions will have a high failure rate [51].

An extensive collection of papers have been publish in the area of tree-based data structures, but very little of them were designed for the purpose of data availability

tracking. There was early work in dataflow [52, 53] that uses the idea to represent arrays as trees. Unfortunately all these approaches assume an implementation in hardware or the overhead of storing every single elements as a leaf node would defeat the purpose of tiling and generate too much storage overhead. The Fresh Breeze memory model [54, 55], which also has its roots in dataflow, is more suited, because it work on chucks of data that encourage the use of tiling. There is also a myriad of spatial data structures that have strong similarities with the RD-Tree. The R-Tree family [56, 57, 58] of data structures can work with higher dimensional data, but all dimensions are considered during splitting and that can lead to a very low fan out. K-d tree [59] uses only one dimesion for splitting, but does not allow for overlap. The K-D-B Tree [60], the Spatial kd Tree[61], the Hybrid Tree [62], and the SH Tree [63] has the closest similarity with the RD-Tree in the aspect that they allow overlap of regions and split nodes on a single dimension. Most of these spatial data structures were designed for efficient queries of large data sets on disk, but non of them were designed with the properties that were required for this work in mind.

Chapter 17

CONCLUSIONS AND FUTURE WORK

This dissertation highlighted two separate approaches - one at the hardware level and one at the software level - to tackle the issue of synchronization in a massive parallel environment. Both approaches are not exclusive and could be combined to get the benefits from both worlds.

At the architectural level this dissertation presented a new design for a dataflow-like fine-grain synchronization, based on the Synchronization State Buffer (SSB) [42], and its implementation at the Hardware Description Level (HDL) of the Cyclops-64 many-core architecture. The experiments were performed on an emulation engine with gate-level accuracy. The results surpassed expectations and show very good scalability for even small problem sizes. Even for larger problem sizes, the non-strict synchronization approach surpasses all other synchronization constructs, such as barriers with hardware support and signal-wait. The most noticeable result is the scalability beyond the 100 core barrier all the way up to the maximum core count of 160 cores.

At the language level this dissertation proposes a new item collection, which facilitates the automatic tiling and synchronization between threads to enforce the correct order of memory operations to prevent data races. A prototype implementation - the RD-Tree - shows that this method could promise efficient data tracking and querying for dense data arrays. It also comes with several additional benefits. The tile size is no longer fixed and can be a tuning parameter, which in turn enables the separation of concerns model. It also greatly reduces the coding effort, because the item collection does all the tiling work automatically. On a broader scale, this means that runtime systems and execution model implementations that employ this method

will be able to support fine-grain task granularity, exhibit less data movement overall, and see increased performance.

Although the results are already very promising, there are still several open questions that haven't been answered in this dissertation and provide a great opportunity of future work to further improve the performance and applicability of this idea. One interesting idea that has been proposed is regarding the tree data structure itself. Currently the tree allows overlap of subregions and this requires query and insertion algorithms to traverse both branches of a tree node if the overlapping region is within the search space. Sometimes this might turn out as unnecessary, because the required data is only in one of the sub-trees. The proposed solution is to further subdivide the tree until there are only regions that are guarantied to overlap or not overlap at all. This could reduce unnecessary traversals of sub-trees and improve the performance of the algorithm.

The current prototype is a proof-of-concept implementation that is limited to dense n-dimensional data arrays with rectangular n-dimensional tiling. One possible extension would be to add support for different types of shapes to enable a wider variety of stencil-like applications. Another possible extension would be to support sparse data arrays. Graph-based applications on the other side wouldn't be a good match and would require a separate optimized item collection designed and tuned for graph creation, traversal, and update.

There are also certain limitations that cannot be addressed by this approach. The algorithm still has to be written with tiling in mind by the programmer. Furthermore the programmer has to ensure that the tiling that is created and guided by the tuning specification is a legal tiling for the algorithm. The item collection cannot verify or ensure this property, because it doesn't know anything about the access pattern within a tile or across tiles. It only maintains data provided to it and serves it when requested. A compiler that can analyze loop dependencies would be required to perform these checks and to create and ensure a legal tiling.

BIBLIOGRAPHY

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[2] Gordon E. Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.

[3] Tse-Yu Yeh and Yale N Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61. ACM, 1991.

[4] Hubert Kaeslin. *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, 2008.

[5] MS Hrishikesh, Doug Burger, Norman P Jouppi, Stephen W Keckler, Keith I Farkas, and Premkishore Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 fo4 inverter delays. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 14–24. IEEE Computer Society, 2002.

[6] Sheng Sun, Yi Han, Xinyu Guo, Kian Haur Chong, Larry McMurchie, and Carl Sechen. 409ps 4.7 fo4 64b adder based on output prediction logic in 0.18 um cmos. In *VLSI, 2005. Proceedings. IEEE Computer Society Annual Symposium on*, pages 52–58. IEEE, 2005.

[7] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[8] Susan J Eggers, Joel S Emer, Henry M Leby, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *Micro, IEEE*, 17(5):12–19, 1997.

[9] Wm A Wulf and Sally A McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[10] Sally A McKee. Reflections on the Memory Wall. In *Proceedings of the 1st Conference on Computing Frontiers*, pages 162–167. ACM, 2004.

[11] Nam Sung Kim, Todd Austin, D Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage Current: Moore's Law Meets Static Power. *Computer*, 36(12):68–75, 2003.

[12] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[13] Mark Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Newsletter, IEEE*, 12(1):11–13, 2007.

[14] Himanshu Kaul, Mark Anders, Steven Hsu, Amit Agarwal, Ram Krishnamurthy, and Shekhar Borkar. Near-Threshold Voltage (NTV) Design: Opportunities and Challenges. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1153–1158. ACM, 2012.

[15] Vikas Agarwal, MS Hrishikesh, Stephen W Keckler, and Doug Burger. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. *ACM SIGARCH Computer Architecture News*, 28(2):248–259, 2000.

[16] G. Almasi, C. Caşcaval, J.G. Castanos, M. Denneau, D. Lieber, J.E. Moreira, and H.S. Warren Jr. Dissecting Cyclops: A detailed analysis of a multithreaded architecture. *ACM SIGARCH Computer Architecture News*, 31(1):38, 2003.

[17] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, et al. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[18] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15–31, 2007.

[19] Michael B Taylor, Walter Lee, Jason E Miller, David Wentzlaff, Ian Bratt, Ben Greenwald, Henry Hoffmann, Paul R Johnson, Jason S Kim, James Psota, et al. Tiled multicore processors. In *Multicore Processors and Systems*, pages 1–33. Springer, 2009.

[20] R.S.N. Arvind, R.S. Nikhil, and K. Pingali. I-Structures: Data Structures for Parallel Computing. *TOPLAS*, 11(4):598–632, 1989.

[21] Arvind and Rishiyur Nikhil. Executing a program on the MIT Tagged-Token Dataflow architecture. In *PARLE Parallel Architectures and Languages Europe*, volume 259 of *Lecture Notes in Computer Science*, pages 1–29. 1987.

[22] P. Barth and R. Nikhil. M-Structures: Extending a Parallel, Non-strict, Functional Language with State. In *Functional Programming Languages and Computer Architecture*, pages 538–568. Springer.

[23] Burton J. Smith. A Pipelined, Shared Resource MIMD Computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, 1978.

[24] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[25] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6. ACM, 1990.

[26] Gail Alverson, Robert Alverson, David Callahan, Brian Koblenz, Allan Porterfield, and Burton Smith. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. In *Proceedings of the 6th International Conference on Supercomputing*, pages 188–197. ACM, 1992.

[27] James T. Kuehn and Burton J. Smith. The Horizon Supercomputing System: Architecture and Software. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, pages 28–34. IEEE Computer Society Press, 1988.

[28] Mark R. Thistle and Burton J. Smith. A Processor Architecture for Horizon. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, pages 35–41. IEEE Computer Society Press, 1988.

[29] Frank Pittelli and David Smitley. Analysis of a 3D Toroidal Network for a Shared Memory Architecture. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, pages 42–46. IEEE Computer Society Press, 1988.

[30] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. *Cilk: An Efficient Multithreaded Runtime System*, volume 30. ACM, 1995.

[31] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads*. PhD thesis, McGill University, 1999.

[32] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G.R. Gao. Position Paper: Using a "Codelet" Program Execution Model for Exascale Machines. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69. ACM, 2011.

[33] Guang R. Gao, Joshua Suetterlein, and Stephane Zuckerman. Toward an Execution Model for Extreme-Scale Systems-Runnemede and Beyond. Technical Report UD-CAPSL-TM 104, University of Delaware, April 2011.

[34] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.

[35] W. Kim and M. Voss. Multicore desktop programming with intel threading building blocks. *Software, IEEE*, 28(1):23–31, 2011.

[36] Michael Burke, Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The Concurrent Collections Programming Model. Technical Report RU-CS-TR 10-12, Rice University, December 2010.

[37] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[38] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, 2005.

[39] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011.

[40] G.R. Gao, T. Sterling, R. Stevens, M. Hereld, and W. Zhu. Parallex: A study of a new parallel computation model. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–6. IEEE, 2007.

[41] Frances Allen, G Almasi, Wanda Andreoni, D Beece, Bruce J. Berne, A Bright, Jose Brunheroto, Calin Cascaval, J Castanos, Paul Coteus, et al. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2):310–327, 2001.

[42] W. Zhu, V.C. Sreedhar, Z. Hu, and G.R. Gao. Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, page 45. ACM, 2007.

[43] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *Proceedings of the 2nd Conference on Computing Frontiers*, page 34. ACM, 2005.

[44] Elkin Garcia, Daniel Orozco, and Guang R. Gao. Energy efficient tiling on a Many-Core Architecture. In *Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2011)*, pages 53–66, 2011.

[45] J. Ributzka, Y. Hayashi, F. Chen, and G.R. Gao. DEEP: An Iterative FPGA-based Many-core Emulation System for Chip Verification and Architecture Research. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 115–118. ACM, 2011.

[46] H. Sakane, L. Yakay, V. Karna, C. Leung, and GR Gao. DIMES: An Iterative Emulation Platform for Multiprocessor-System-On-Chip Designs. In *2003 IEEE International Conference on Field-Programmable Technology (FPT), 2003. Proceedings*, pages 244–251, 2003.

[47] A. Kejariwal, H. Saito, X. Tian, M. Girkar, W. Li, U. Banerjee, A. Nicolau, and C.D. Polychronopoulos. Lightweight Lock-Free Synchronization Methods for Multithreading. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 361–371. ACM, 2006.

[48] J. Ributzka, Y. Hayashi, J.B. Manzano, and G.R. Gao. The Elephant and the Mice: The Role of Non-Strict Fine-Grain Synchronization for Modern Many-Core Architectures. In *Proceedings of the International Conference on Supercomputing*, pages 338–347. ACM, 2011.

[49] M. Herlihy and J.E.B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, page 300. ACM, 1993.

[50] B.D. Carlstrom, A. McDonald, H. Chafi, J.W. Chung, C.C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. *ACM SIGPLAN Notices*, 41(6):13, 2006.

[51] Vladimir Gajinov, Ferad Zyulkyarov, Osman S Unsal, Adrian Cristal, Eduard Ayguade, Tim Harris, and Mateo Valero. QuakeTM: Parallelizing a Complex Sequential Application using Transactional Memory. In *Proceedings of the 23rd International Conference on Supercomputing*, pages 126–135. ACM, 2009.

[52] Kim P Gostelow and Robert E Thomas. A view of dataflow. In *afips*, page 629. IEEE Computer Society, 1899.

[53] William B Ackerman. A structure memory for data flow computers. Technical report, DTIC Document, 1977.

[54] Jack B Dennis. Fresh breeze: a multiprocessor chip architecture guided by modular programming principles. *ACM SIGARCH Computer Architecture News*, 31(1):7–15, 2003.

[55] Jack B Dennis, Guang R Gao, and Xiao X Meng. Experiments with the fresh breeze tree-based memory model. *Computer Science-Research and Development*, 26(3-4):325–337, 2011.

[56] A. Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching*, volume 14. ACM, 1984.

[57] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*, volume 19. ACM, 1990.

[58] Stefan Berchtold, Daniel A Keim, and Hans-Peter Kriegel. The X-Tree: An Index Structure for High-Dimensional Data. *Readings in multimedia computing and networking*, page 451, 2001.

[59] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[60] John T Robinson. The kdb-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 10–18. ACM, 1981.

[61] Beng Chin Ooi, Ken J McDonell, and Ron Sacks-Davis. Spatial kd-tree: An indexing mechanism for spatial databases. In *Proc. IEEE COMPSAC Conf., Tokyo*, pages 433–438, 1987.

[62] Kaushik Chakrabarti and Sharad Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 440–447. IEEE, 1999.

[63] Beomseok Nam and Alan Sussman. A comparative study of spatial indexing techniques for multidimensional scientific datasets. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on*, pages 171–180. IEEE, 2004.

Appendix A

CYCLOPS-64

A.1 Instruction Format

Table A.1: X1: Fix-Point Instruction Format

Primary Opcode	Target Register	Source Register 1	Source Register 2	unused	Extended Opcode
4	6	6	6	1	9
OP	RT	RA	RB	0	XO

Table A.2: X2: Floating-Point Instruction Format

Primary Opcode	Target Register	Source Register 1	Source Register 2	Precision (single/double)	Extended Opcode
4	6	6	6	1	9
OP	RT	RA	RB	p	XO

Table A.3: X3: Logic and Compare Instruction Format

Primary Opcode	Target Register	Source Register 1	Source Register 2	Logic/Condition Code	Extended Opcode
4	6	6	6	6	4
OP	RT	RA	RB	UU/CC	XO

Table A.4: X4: Bit Field Instruction Format

Primary Opcode	Target/Source Register	Source Register	Immediate 1	Immediate 2	Extended Opcode
4	6	6	6	6	4
OP	RT	RA	M	N	XO

Table A.5: X5: Move Special Purpose Register Instruction Format

Primary Opcode	Target Register	Special Purpose Register	Source Register	unused	Extended Opcode
4	6	6	6	1	9
OP	RT	SPR	RB	0	XO

Table A.6: EX: Extended Synchronization State Buffer Instruction Format

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
OP	RT	RA	RB	EE	Sz	S	0

Table A.7: C: Compare and Trap Immediate Instruction Format

Primary Opcode	Target Register	Source Register	Integer Condition Codes	Immediate
4	6	6	4	12
OP	RT	RA	VV	IM

Table A.8: D: Memory Instruction Format

Primary Opcode	Target/Source Register	Address Register	Size	Offset
4	6	6	2	14
OP	RT/RS	RA	Sz	D

Table A.9: I: Fix-Point Immediate Instruction Format

Primary Opcode	Target Register	Source Register	Immediate
4	6	6	16
OP	RT	RA	IMM

Table A.10: BC: Conditional Branch Instruction Format

Primary Opcode	Condition Code	Source Register	Prediction	Displacement
4	6	6	1	15
OP	CC	RA	P	DISP

Table A.11: B: Branch and Link Instruction Format

Primary Opcode	Target Register	Displacement
4	6	22
OP	RT	DISP

A.2 E-SSB Instructions

A.2.1 Read Lock

```
rlockb  RT,RA   Read Lock Byte              Sz=0, S=1
rlockh  RT,RA   Read Lock Half              Sz=1, S=1
rlockw  RT,RA   Read Lock Word              Sz=2, S=1

rlockbu RT,RA   Read Lock Byte Unsigned     Sz=0, S=0
rlockhu RT,RA   Read Lock Half Unsigned     Sz=1, S=0
rlockwu RT,RA   Read Lock Word Unsigned     Sz=2, S=0
rlockd  RT,RA   Read Lock Double Unsigned   Sz=3, S=0
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x00	Sz	S	0

The instruction tries to obtain a read lock on the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in register RT means **SUCCESS** and a value of -1 means **FAIL**. If the read lock was successfully acquired, then the value of the memory location is available in register RT+1.

A.2.2 Write Lock

```
wlockb  RT,RA   Write Lock Byte             Sz=0, S=1
wlockh  RT,RA   Write Lock Half             Sz=1, S=1
wlockw  RT,RA   Write Lock Word             Sz=2, S=1

wlockbu RT,RA   Write Lock Byte Unsigned    Sz=0, S=0
wlockhu RT,RA   Write Lock Half Unsigned    Sz=1, S=0
```

```
wlockwu RT,RA   Write Lock Word Unsigned     Sz=2, S=0
wlockd  RT,RA   Write Lock Double Unsigned   Sz=3, S=0
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x01	Sz	S	0

The instruction tries to obtain a write lock on the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in register RT means **SUCCESS** and a value of -1 means **FAIL**. If the read lock was successfully acquired, then the value of the memory location is available in register RT+1.

A.2.3 Unlock

```
unlockb  RT,RA   Write Lock Byte             Sz=0
unlockh  RT,RA   Write Lock Half             Sz=1
unlockw  RT,RA   Write Lock Word             Sz=2
unlockd  RT,RA   Write Lock Double           Sz=3
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x02	Sz	0	0

The instruction tries to unlock a previously acquired read or write lock for the memory location defined in register RA. The return code is written to register RT. A value of 0 in register RT means **SUCCESS** and a value of -1 means **FAIL**.

A.2.4 Single-Writer-Single-Reader Mode 1 Read

```
swsr1_rb  RT,RA   SWSR1 Read Byte              Sz=0, S=1
swsr1_rh  RT,RA   SWSR1 Read Half              Sz=1, S=1
swsr1_rw  RT,RA   SWSR1 Read Word              Sz=2, S=1

swsr1_rbu RT,RA   SWSR1 Read Byte Unsigned     Sz=0, S=0
swsr1_rhu RT,RA   SWSR1 Read Half Unsigned     Sz=1, S=0
swsr1_rwu RT,RA   SWSR1 Read Word Unsigned     Sz=2, S=0
swsr1_rd  RT,RA   SWSR1 Read Double Unsigned   Sz=3, S=0
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x03	Sz	S	0

This instructions tries to read a value from the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in RT means **SUCCESS** and a value of -1 means **FAIL**. If the read was successful, then the value of the memory location is available in RT+1.

A.2.5 Single-Writer-Single-Reader Mode 1 Write

```
swsr1_wb  RT,RA,RB   SWSR1 Write Byte          Sz=0
swsr1_wh  RT,RA,RB   SWSR1 Write Half          Sz=1
swsr1_ww  RT,RA,RB   SWSR1 Write Word          Sz=2
swsr1_wd  RT,RA,RB   SWSR1 Write Double        Sz=3
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x04	Sz	0	0

This instructions tries to write the value specified in register RB to the memory location defined in register RA. The return code is written to register RT. A value of 0 in register RT means **SUCCESS** and a value of -1 means **FAIL**.

A.2.6 Single-Writer-Single-Reader Mode 2 Read

```
swsr2_rb  RT,RA   SWSR2 Read Byte              Sz=0, S=1
swsr2_rh  RT,RA   SWSR2 Read Half              Sz=1, S=1
swsr2_rw  RT,RA   SWSR2 Read Word              Sz=2, S=1

swsr2_rbu RT,RA   SWSR2 Read Byte Unsigned     Sz=0, S=0
swsr2_rhu RT,RA   SWSR2 Read Half Unsigned     Sz=1, S=0
swsr2_rwu RT,RA   SWSR2 Read Word Unsigned     Sz=2, S=0
swsr2_rd  RT,RA   SWSR2 Read Double Unsigned   Sz=3, S=0
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x05	Sz	S	0

This instructions tries to read a value from the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the return value is written to register RT+1. A value of 0 in register RT means **SUCCESS** and a value of -2 means **WAIT**. If the read was successful, then the value of the memory location is available in register RT+1.

A.2.7 Single-Writer-Single-Reader Mode 2 Write

```
swsr2_wb  RT,RA,RB   SWSR2 Write Byte              Sz=0
swsr2_wh  RT,RA,RB   SWSR2 Write Half              Sz=1
swsr2_ww  RT,RA,RB   SWSR2 Write Word              Sz=2
swsr2_wd  RT,RA,RB   SWSR2 Write Double            Sz=3
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x06	Sz	0	0

This instructions tries to write the value specified in register RB to the memory location defined in register RA. This instruction implicitly writes to two registers. The return code is written to register RT and the thread id (TID) to register RT+1. A value of 0 in register RT means **SUCCESS**, a value of -1 means **FAIL**, and a value of -2 means **NO WAITING THREAD**.

A.2.8 Single-Writer-Single-Reader Mode 3 Read

```
swsr3_rb  RT,RA   SWSR3 Read Byte              Sz=0, S=1
swsr3_rh  RT,RA   SWSR3 Read Half              Sz=1, S=1
swsr3_rw  RT,RA   SWSR3 Read Word              Sz=2, S=1

swsr3_rbu RT,RA   SWSR3 Read Byte Unsigned     Sz=0, S=0
swsr3_rhu RT,RA   SWSR3 Read Half Unsigned     Sz=1, S=0
swsr3_rwu RT,RA   SWSR3 Read Word Unsigned     Sz=2, S=0
swsr3_rd  RT,RA   SWSR3 Read Double Unsigned   Sz=3, S=0
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	RT	RA	0x00	0x07	Sz	S	0

```
ldbs  RT,D(RA)   SWSR3 Read Byte              Sz=0, OP=0x6
ldhs  RT,D(RA)   SWSR3 Read Half              Sz=1, OP=0x6
ldws  RT,D(RA)   SWSR3 Read Word              Sz=2, OP=0x6

ldbus RT,D(RA)   SWSR3 Read Byte Unsigned     Sz=0, OP=0x8
ldhus RT,D(RA)   SWSR3 Read Half Unsigned     Sz=1, OP=0x8
ldwus RT,D(RA)   SWSR3 Read Word Unsigned     Sz=2, OP=0x8
ldds  RT,D(RA)   SWSR3 Read Double Unsigned   Sz=3, OP=0x8
```

Primary Opcode	Target/Source Register	Address Register	Size	Offset
4	6	6	2	14
OP	RT	RA	Sz	D

The Single-Writer-Single-Reader Mode 3 instructions allow to be encoded in two separate formats, but they still provide the same memory semantics. The second format is the same format that is used by regular load and store instructions. This way SWSR3 load/store instructions can be used as drop-in replacement for regular load/store instructions without any changes. This instructions tries to read a value from the memory location defined in register RA. The load operation will wait in the memory controller if the location is empty until it is set to full. Once the location has been set to full the value is returned to register RT and the location is set empty again.

A.2.9 Single-Writer-Single-Reader Mode 3 Write

```
swsr3_wb  RA,RB   SWSR3 Write Byte              Sz=0
swsr3_wh  RA,RB   SWSR3 Write Half              Sz=1
swsr3_ww  RA,RB   SWSR3 Write Word              Sz=2
swsr3_wd  RA,RB   SWSR3 Write Double            Sz=3
```

Primary Opcode	Target Register	Source Register 1	Source Register 2	E-SSB Code	Size	Signed	unused
4	6	6	6	6	2	1	1
0x0	0x00	RA	RB	0x08	Sz	0	0

```
stbs  RS,D(RA)   SWSR3 Write Byte              Sz=0
sths  RS,D(RA)   SWSR3 Write Half              Sz=1
stws  RS,D(RA)   SWSR3 Write Word              Sz=2
stds  RS,D(RA)   SWSR3 Write Double            Sz=3
```

Primary Opcode	Target/Source Register	Address Register	Size	Offset
4	6	6	2	14
0x7	RS	RA	Sz	D

This instructions writes the value specified in register RB to the memory location defined in register RA and sets the memory location to full.

Appendix B

COPYRIGHT INFORMATION

This dissertation contains figures, tables and text that also have been published in conference proceedings or are available in the public domain.

B.1 Wikipedia

Figure 1.2 has been obtained from Wikipedia. The figure has been released by its author into the public domain and the author also granted the right to use it for any purpose without any restrictions.

B.2 ACM License Agreement

The papers about fine-grain synchronization [48] and the emulation system [45] have been used in this dissertation and the required permissions have been obtained from ACM under license agreements 3203440929960 and 3203441115225.

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

Aug 07, 2013

This is a License Agreement between Juergen Ributzka ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Association for Computing Machinery, Inc., and the payment terms and conditions.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3203440929960
License date	Aug 07, 2013
Licensed content publisher	Association for Computing Machinery, Inc.
Licensed content publication	Proceedings of the international conference on Supercomputing
Licensed content title	The elephant and the mice: the role of non-strict fine-grain synchronization for modern many-core architectures
Licensed content author	Juergen Ributzka, et al
Licensed content date	May 31, 2011
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	Yes
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	Concurrency and Synchronization in the Modern Many-Core Era: Challenges and Opportunities
Expected completion date	Aug 2013
Estimated size (pages)	150
Billing Type	Invoice
Billing address	20800 Homestead Rd Apt 7B Cupertino, DE 95014 United States
Total	8.00 USD

Rightslink Terms and Conditions for ACM Material

1. The publisher of this copyrighted material is Association for Computing Machinery, Inc. (ACM). By clicking "accept" in connection with completing this licensing transaction, you agree that the following terms and conditions apply to this transaction (along with the Billing and Payment terms and conditions established by Copyright Clearance Center, Inc. ("CCC"), at the time that you opened your Rightslink account and that are available at any time at).

2. ACM reserves all rights not specifically granted in the combination of (i) the license details provided by you and accepted in the course of this licensing transaction, (ii) these terms and conditions and (iii) CCC's Billing and Payment terms and conditions.

3. ACM hereby grants to licensee a non-exclusive license to use or republish this ACM-copyrighted material* in secondary works (especially for commercial distribution) with the stipulation that consent of the lead author has been obtained independently. Unless otherwise stipulated in a license, grants are for one-time use in a single edition of the work, only with a maximum distribution equal to the number that you identified in the licensing process. Any additional form of republication must be specified according to the terms included at the time of licensing.

*Please note that ACM cannot grant republication or distribution licenses for embedded third-party material. You must confirm the ownership of figures, drawings and artwork prior to use.

4. Any form of republication or redistribution must be used within 180 days from the date stated on the license and any electronic posting is limited to a period of six months unless an extended term is selected during the licensing process. Separate subsidiary and subsequent republication licenses must be purchased to redistribute copyrighted material on an extranet. These licenses may be exercised anywhere in the world.

5. Licensee may not alter or modify the material in any manner (except that you may use, within the scope of the license granted, one or more excerpts from the copyrighted material, provided that the process of excerpting does not alter the meaning of the material or in any way reflect negatively on the publisher or any writer of the material).

6. Licensee must include the following copyright and permission notice in connection with any reproduction of the licensed material: "[Citation] © YEAR Association for Computing Machinery, Inc. Reprinted by permission." Include the article DOI as a link to the definitive version in the ACM Digital Library. Example: Charles, L. "How to Improve Digital Rights Management," Communications of the ACM, Vol. 51:12, © 2008 ACM, Inc. http://doi.acm.org/10.1145/nnnnnn.nnnnnn (where nnnnnn.nnnnnn is replaced by the actual number).

7. Translation of the material in any language requires an explicit license identified during the licensing process. Due to the error-prone nature of language translations, Licensee must include the following copyright and permission notice and disclaimer in connection with any reproduction of the licensed material in translation: "This translation is a derivative of ACM-copyrighted material. ACM did not prepare this translation and does not guarantee that it is an accurate copy of the originally published work. The original intellectual property contained in this work remains the property of ACM."

8. You may exercise the rights licensed immediately upon issuance of the license at the end of the licensing transaction, provided that you have disclosed complete and accurate details of your proposed use. No license is finally effective unless and until full payment is received from you (either by CCC or ACM) as provided in CCC's Billing and Payment terms and conditions.

9. If full payment is not received within 90 days from the grant of license transaction, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted.

10. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and publisher reserves the right to take any and all action to protect its copyright in the materials.

11. ACM makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

12. You hereby indemnify and agree to hold harmless ACM and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

13. This license is personal to the requestor and may not be sublicensed, assigned, or transferred by you to any other person without publisher's written permission.

14. This license may not be amended except in a writing signed by both parties (or, in the case of ACM, by CCC on its behalf).

15. ACM hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and ACM (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

16. This license transaction shall be governed by and construed in accordance with the laws of New York State. You hereby agree to submit to the jurisdiction of the federal and state courts located in New York for purposes of resolving any disputes that may arise in connection with this licensing transaction.

17. There are additional terms and conditions, established by Copyright Clearance Center, Inc. ("CCC") as the administrator of this licensing service that relate to billing and payment for licenses provided through this service. Those terms and conditions apply to each transaction as if they were restated here. As a user of this service, you agreed to those terms and conditions at the time that you established your account, and you may see them again at any time at http://myaccount.copyright.com

18. Thesis/Dissertation: This type of use requires only the minimum administrative fee. It is not

a fee for permission. Further reuse of ACM content, by ProQuest/UMI or other document delivery providers, or in republication requires a separate permission license and fee. Commercial resellers of your dissertation containing this article must acquire a separate license.

Special Terms:

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501084859.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

Aug 07, 2013

This is a License Agreement between Juergen Ributzka ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by Association for Computing Machinery, Inc., and the payment terms and conditions.

License Number	3203441115225
License date	Aug 07, 2013
Licensed content publisher	Association for Computing Machinery, Inc.
Licensed content publication	Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays
Licensed content title	DEEP: an iterative fpga-based many-core emulation system for chip verification and architecture research
Licensed content author	Juergen Ributzka, et al
Licensed content date	Feb 27, 2011
Type of Use	Thesis/Dissertation
Requestor type	Author of this ACM article
Is reuse in the author's own new work?	Yes
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	Concurrency and Synchronization in the Modern Many-Core Era: Challenges and Opportunities
Expected completion date	Aug 2013
Estimated size (pages)	150
Billing Type	Credit Card
Credit card info	Master Card ending in 7624
Credit card expiration	06/2015
Total	8.00 USD
Terms and Conditions	

Rightslink Terms and Conditions for ACM Material

1. The publisher of this copyrighted material is Association for Computing Machinery, Inc.

9. If full payment is not received within 90 days from the grant of license transaction, then any license preliminarily granted shall be deemed automatically revoked and shall be void as if never granted. Further, in the event that you breach any of these terms and conditions or any of CCC's Billing and Payment terms and conditions, the license is automatically revoked and shall be void as if never granted.

10. Use of materials as described in a revoked license, as well as any use of the materials beyond the scope of an unrevoked license, may constitute copyright infringement and publisher reserves the right to take any and all action to protect its copyright in the materials.

11. ACM makes no representations or warranties with respect to the licensed material and adopts on its own behalf the limitations and disclaimers established by CCC on its behalf in its Billing and Payment terms and conditions for this licensing transaction.

12. You hereby indemnify and agree to hold harmless ACM and CCC, and their respective officers, directors, employees and agents, from and against any and all claims arising out of your use of the licensed material other than as specifically authorized pursuant to this license.

13. This license is personal to the requestor and may not be sublicensed, assigned, or transferred by you to any other person without publisher's written permission.

14. This license may not be amended except in a writing signed by both parties (or, in the case of ACM, by CCC on its behalf).

15. ACM hereby objects to any terms contained in any purchase order, acknowledgment, check endorsement or other writing prepared by you, which terms are inconsistent with these terms and conditions or CCC's Billing and Payment terms and conditions. These terms and conditions, together with CCC's Billing and Payment terms and conditions (which are incorporated herein), comprise the entire agreement between you and ACM (and CCC) concerning this licensing transaction. In the event of any conflict between your obligations established by these terms and conditions and those established by CCC's Billing and Payment terms and conditions, these terms and conditions shall control.

16. This license transaction shall be governed by and construed in accordance with the laws of New York State. You hereby agree to submit to the jurisdiction of the federal and state courts located in New York for purposes of resolving any disputes that may arise in connection with this licensing transaction.

17. There are additional terms and conditions, established by Copyright Clearance Center, Inc. ("CCC") as the administrator of this licensing service that relate to billing and payment for licenses provided through this service. Those terms and conditions apply to each transaction as if they were restated here. As a user of this service, you agreed to those terms and conditions at the time that you established your account, and you may see them again at any time at http://myaccount.copyright.com

18. Thesis/Dissertation: This type of use requires only the minimum administrative fee. It is not a fee for permission. Further reuse of ACM content, by ProQuest/UMI or other document delivery providers, or in republication requires a separate permission license and fee. Commercial resellers of your dissertation containing this article must acquire a separate license.

Special Terms:

If you would like to pay for this license now, please remit this license along with your payment made payable to "COPYRIGHT CLEARANCE CENTER" otherwise you will be invoiced within 48 hours of the license date. Payment should be in the form of a check or money order referencing your account number and this invoice number RLNK501084861.
Once you receive your invoice for this order, you may pay your invoice by credit card. Please follow instructions provided at that time.

Make Payment To:
Copyright Clearance Center
Dept 001
P.O. Box 843006
Boston, MA 02284-3006

For suggestions or comments regarding this order, contact RightsLink Customer Support: customercare@copyright.com or +1-877-622-5543 (toll free in the US) or +1-978-646-2777.

Gratis licenses (referencing $0 in the Total field) are free. Please retain this printable license for your reference. No payment is required.