EFFICIENT SYNCHRONIZATION FOR A LARGE-SCALE MULTI-CORE CHIP ARCHITECTURE

by

Weirong Zhu

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

Spring 2007

EFFICIENT SYNCHRONIZATION FOR A LARGE-SCALE MULTI-CORE CHIP ARCHITECTURE

by

Weirong Zhu

Approved: ____
Gonzalo R. Arce, Ph.D.
Chairperson of the Department of Electrical and Computer Engineering

Approved: ____
Eric W. Kaler, Ph.D.
Dean of the College of Engineering

Approved: ____
Carolyn A. Thoroughgood, Ph.D.
Vice Provost for Research and Graduate Studies

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: ______________________________

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: ______________________________

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: ______________________________

Li Liao, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

First of all, I would like to thank my advisor, Prof. Guang R. Gao, for his advisement, support, understanding, concern, and help through out all these years for my Ph.D. program. His vision, insight, attitude, and methodology on the research problems always teach me a lot. His enthusiasm, belief, and perseverance on the research of parallel processing and multithreading always inspire and encourage me to continue my work every time when I feel frustrated, confused, or desperate. In my academia life, he is my advisor for guiding me through the graduate study and research. In my personal life, he and his wife Peggy take good care of me in many ways.

I would also like to thank Dr. Kevin Theobold, who introduced me to the new world of parallel system and computer architecture and taught me lots of computing skills, which benefit my whole Ph.D. study.

I thank Dr. Fouad Kiamilev, Dr. Stephan Bohacek, and Dr. Li Liao, who served in my dissertation committee. Their comments and feedback on my dissertation are invaluable.

I want to acknowledge many present members of our group at CAPSL, University of Delaware. They have helped and contributed in many ways to the work of this dissertation. In particular, I would like to thanks Ziang Hu, Juan del Cuvillo, Yuan Zhang, and Ioannis E. Venetis for many useful and insightful discussions. I also got many help and guidance from some former CAPSL and ACAPS members. I would like to thanks Vugranam C. Sreedhar, and Parimala Thulasiraman for their selfless help, encouragement, and suggestions.

I would also like to thank Yanwei Niu, Chuan Shen, and Jizhu Lu for the collaboration during the course of the EARTH project, which lays the foundation for my understanding on fine-grain multithreading techniques.

Cyclops-64 is a big and exciting project with the collaboration of almost the whole CAPSL group. I am very lucky to be one of them. I would like to thank all my teammates, including Ziang Hu, Juan del Cuvillo, Fei Chen, Guangming Tang, Ioannis E. Venetis, Yanwei Niu, Long Chen, and many others. The team provides a stimulus environment for a lot of interesting scientific discussions and collaborations.

My internship in ETI is a valuable asset for gaining experience and understanding on Cyclops-64 architecture and its system software. I would like to thank my colleagues, Geoff Gerfin, Dobry Brice, John Tully, and Wesley Toland, for all the work we spent together on building up the Cyclops-64 system software toolchain, without which my dissertation can not be completed.

I would like to acknowledge the support from IBM, in particular, Monty Denneau, who is the architect of the Cyclops-64 architecture.

This work would not have been possible without the generous support of the research grants by IBM, ETI, DoD, NSF (CNS-0509332), DoE (DE-FC02-01ER25503), and other government sponsors.

Xinyi Zhang, gave me all of her love, understanding, and patience for all these years. She colored my life.

Finally, my parents and my sister, deserve the most credits for this accomplishment. Their love, support, and patience are the source of everything I could achieve. They gave me far more than what I could ever give back.

Dedicated to my parents, *Tianxiu Li* and *Peifu Zhu*.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Multi-core architectures are becoming mainstream, permitting increasing on-chip parallelism through hardware support for multithreading. Synchronization, especial fine-grain synchronization, is essential to the effective utilization of the computational power of high-performance large-scale multi-core architectures. However, designing and implementing fine-grain synchronization in such architectures presents several challenges, including issues of synchronization induced overhead, storage cost, scalability, and the level of granularity to which synchronization is applicable.

Using the 160-core IBM Cyclops-64 multi-core chip architecture as a case study, this dissertation first presents a thorough performance measurement, evaluation, and customization of a range of widely used synchronization mechanisms. This dissertation then proposes *Synchronization State Buffer* (SSB), a scalable architectural design for fine-grain synchronization that efficiently enforces word-level *mutual exclusion* and *read-after-write data-dependencies* between concurrent threads. The design of SSB is motivated by the following simple observation: *at any instance during the parallel execution only a small fraction of memory locations are actively participating in synchronization.* Based on this observation we present a fine-grain synchronization design that records and manages the states of frequently synchronized data using modest hardware support. We have implemented SSB design in the context of the IBM Cyclops-64 architecture. Using detailed simulation, the experimental results demonstrate significant performance gain due to the use of SSB-based fine-grain synchronization solution for a set of selected benchmarks with different workload characteristics.

Chapter 1

INTRODUCTION

As advances in IC processing technology allow the feature size to drop, density of transistors on silicon chips is to continue increasing for the next few years following Moore's Law. At this pace, a billion-transistor chip is approaching [30]. However, the delivered performance versus number of transistors integrated in a chip for microarchitecture keeps declining over time [150]. Power consumption and dissipation considerations also place additional obstacles to improve the performance. Due to fundamental circuit limitations, limited amounts of instruction level parallelism, and the memory wall problem, computer architects look for new designs, other than the single-thread wide-issue Superscalar approach, to utilize the transistor budget and mitigate the effects of high interconnect delays. On the other hand, by fabricating hundreds of millions of transistors on a single die, it becomes possible to put a complete multiprocessor, including both CPUs and memory, on a single chip, what is known as multi-core architecture.

1.1 Towards Multi-Core Chip Architectures and Beyond

Instead of devoting the entire die to a single and complex processor, the multi-core chip architecture design integrates a number of simple processors on a single die. Provided that hundreds of millions (towards billions) of transistors can be fabricated into a single chip die, it is believed that the multi-core architecture has many advantages over the conventional single core chip.

- By partitioning the chip resources into individual small, localized simple cores, the effect of the interconnect delay is limited.

- By enabling multiple cores to share the chip resources, such as on-chip memory (or L2 cache), interconnect network, and on-chip/off-chip memory bandwidth, the resource utilization is improved.

- Given the fact that chip power consumptions drop significantly with reductions in frequency, multi-core architectures can alleviate the power dissipation problem without reducing the computation capability by running multiple cores with moderate clock rate [150].

- Multi-core chip architectures naturally exploit thread-level and process-level parallelism, which are expected to be widespread in future applications and multiprocessor-aware operating systems and environments [75].

Not surprisingly, all major microprocessor manufacturers have already begun to move their microarchitecture towards multi-core design [27, 89, 101, 18, 113, 39, 91].

While the dual-core microprocessors begin to dominate the market of servers and personal computers, both the industry and academia are exploiting the design space of the future multi-core architectures by integrating a large number of cores (10s and beyond) into a single chip, which is referred to as large-scale multi-core chip (or many-core chip) [46, 27] throughout this dissertation. For example, Intel recently announced its research prototype many-core design with 80 cores on a single die [160]. Another example is the IBM Cyclops-64 chip architecture, which supports 160 hardware thread units in one chip [52, 53].

Unlike traditional uniprocessor chips where few architecture designs (for instance, Superscalar, and VLIW) dominate, researchers in multi-core chips have not yet reached (or even come close to reaching) a consensus on a architectural design that will be successful in the future. Recently we have seen many proposals from both the industry and academia actively exploiting the design space of multi-core chip design. The current multi-core architecture designs can be categorized as two types [61]. The first type glues together existing Superscalar/VLIW processor cores into a single chip with only minor changes. This is the approach that is now taken by major microprocessor manufacturers. For example, Intel Core Duo Processor [36], AMD Opteron dual-Core processor [12], IBM Power5 dual-Core processor [95], and many others that are now available on the market, can be attributed to this type of multi-core design. Instead of being based on existing microprocessor design, the other type of multi-core design takes a different approach to completely redesign the chip architecture by exploring the parallel architecture design space and searching for the most suitable chip architecture models. IBM Cell processor [72, 71, 55], IBM Cyclops-64 chip [52, 53], Intel Tera-Scale researcher chip [91], and some others are now exploring this type of multi-core design.

In this dissertation, the research targets to a class of homogeneous large-scale multi-core architectures which are designed for the domain of future high performance computational applications - pioneered by the original Bluegene Cyclops (BG/C) architecture proposed by researchers at IBM T.J. Watson Research Center [33]. Later, a derivation of the earlier design inspired and steered the design of the IBM Cyclops-64 chip [52, 53]. With such a chip design, massive on-chip parallelism are provided through a large number of on-chip cores (can be 100s and beyond).

1.2 Problem Description

With the emergence of the large-scale multi-core chip architectures, parallel processing techniques, especially multithreading, is undergoing a revival. It has been long realized that synchronization is crucial for the correctness and performance of parallel programs. For a multithreading program, the coordination of concurrent tasks is called synchronization. Synchronization controls concurrent access to shared data and resources, or enforces certain ordering between threads. Synchronization is critical for both the correctness and performance of a parallel program.

To understand the behavior and performance of parallel programs on the approaching large-scale multi-core architectures, it is important to re-evaluate the widely used synchronization mechanisms on multi-core chips with a large number of cores. Since little experience has been gained for multi-core chips with more than 100 cores, the evaluation and performance measurement of synchronization mechanisms, such as spin-lock algorithms, and lock-free concurrent data structures, can provide insight into the following aspects of software/hardware development of multi-core architectures:

- Provide application developers a better understanding of the behavior of various synchronization mechanisms on large-scale multi-core architectures.
- Give system software, library, and compiler developers hints regarding possible synchronization related optimizations and/or language extensions specific to multi-core architectures.
- Help computer architects to understand the pros and cons of the architecture design for hardware support for synchronization.

To complement the absence of such a study under the multi-core arena, this dissertation presents thorough performance measurement and evaluation of a range of most widely used synchronization mechanisms on the IBM 160-core Cyclops-64 chip architecture.

Moreover, in order to fully utilize the massive intra-chip parallelism provided by such large-scale multi-core chips, it is important to exploit the fine-grain parallelism inherent in the applications. The granularity of parallelism that can be efficiently exploited in such processors is often restricted by the lack of effective architectural support for efficient fine-grain synchronization. Software-only solution (with very limited architectural support) for fine-grain synchronization in such processors can often lead poor scalability, high synchronization overhead, and high storage cost.

Thus, on large-scale multi-core architectures, fine-grain synchronization is essential to the effective exploitation of fine-grain parallelism of applications. On large-scale multi-core chips (with 16 to 100 cores and beyond), the on-chip storage (memory) available per processor core is far less (often 1-2 orders less) than traditional single core microprocessors. On the other hand, there are plenty of distributed resources (e.g. large number of thread and memory units, ample on-chip interconnection bandwidth, etc.) available to facilitate efficient fine-grain coordination between processing cores and memory. Therefore, the following new challenges are emerging with respect to fine-grain synchronization solutions in large-scale multi-core architectures, they should:

- be scalable and can fully exploit the parallelism due to the distributed on-chip resources.
- be supported with limited on-chip resources.
- incur low synchronization overhead.
- be able to support a variety of synchronization functionalities with modest hardware cost.
- be able to smoothly and efficiently handle the cases where the precise synchronization point cannot be resolved statically at compile time.

The solution proposed in this dissertation, named *Synchronization State Buffer* (SSB), is a novel architectural extension to large-scale multi-core chip architectures. With only modest hardware cost, SSB can efficiently facilitate word-level fine-grain synchronization on multi-core chips with a large number of cores. A fine-grain synchronization solution is designed based on the effective interaction between software and SSB hardware.

1.3 Contributions

This dissertation works along the direction of achieving effective and efficient synchronization for large-scale multi-core architectures. It consists of two major parts: the performance measurement, evaluation, and customization of widely used synchronization mechanisms on large-core multi-core chips; a scalable and efficient fine-grain synchronization solution for multi-core chips with a large number of cores. The contributions made by this dissertation are:

- Present a thorough performance measurement and evaluation of a range of widely used synchronization mechanisms, including spin-lock algorithms, lock-free concurrent data structures, etc., on the IBM Cyclops-64 chip architectures.
- Based on the performance evaluation, this dissertation proposes customized algorithms/implementations of chosen synchronization mechanisms by taking advantage of underlying hardware features of large-scale multi-core chip architectures.
- Propose a novel synchronization architectural feature, with a modest hardware extension to large-scale multi-core architectures, called *Synchronization State Buffer* (SSB). SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of frequently synchronized data units to support and accelerate word-level fine-grain synchronization. An interesting aspect of our SSB design is that it avoids enormous on-chip memory storage cost, and yet creates an illusion that each word in memory is associated with a set of states by only attaching a small hardware buffer to the memory controller of each memory bank. SSB caches the access states of memory locations that are currently accessed by SSB synchronization operations.
- Present the design of an architectural model for SSB, that consists of the description of the various SSB states and the state transitions. Based on this architectural

model, SSB can be used to enforce mutual exclusion and read-after-write data dependencies between a large number of threads. In the case of mutual exclusion, SSB allows each memory word to be individually locked with minimal overhead. SSB supports various locks: read lock (shared lock), write lock (exclusive lock), as well as recursive lock. For data synchronization that enforces the read-after-write dependencies between threads, SSB allows fine-grained low-overhead synchronized read and write operation on word in memory. SSB supports several modes of data synchronization: two single-writer-single-reader modes, and one single-writer-multiple-reader mode. By coordinating with the software, SSB efficiently facilitates fine-grained synchronizations to help multithreading programs exploit fine-grained parallelism inherent in applications.

- Present the implementation of SSB on the simulator of IBM Cyclops-64 chip architecture, and the corresponding software support in the Cyclops-64 system software toolchain.

- Demonstrate the effectiveness and efficiency of the SSB-based fine-grain synchronization solution on IBM Cyclops-64 chip architecture. Using detailed simulation with microbenchmarks and application kernels, our experimental results demonstrate the effectiveness and efficiency of SSB solution for supporting fine-grain synchronization.

  - For mutual exclusion: our method exploits the ample parallelism that often exists in operations on different elements of the concurrent data structures. Using distributed fine-grain locking on each memory unit, we avoid the unnecessary serialization of those operations without incurring any extra memory usage. In addition, the SSB has also resulted in considerable reduction of the overhead of each individual lock/unlock pair. Also, compared to the software-only solutions, up to 50% performance improvement has been observed for the benchmarks we tested.

- For read-after-write dependence synchronization: our method encourages the exploration of do-across style loop-level parallelism - where *loop-carried* data dependence can often be directly implemented by the application of our fine-grain solutions and the removal of barriers. Our experimental results demonstrate significant performance gain due to the use of such fine-grain synchronization. For instance, by adopting a fine-grain synchronization based parallelization scheme, we observe a 312% performance improvement over the coarse-grain based approach when solving linear recurrence equations.
- The experiments also demonstrate that 1) a small SSB for each memory bank is normally sufficient to record and manage the access states of outstanding synchronizing data units for multithreading programs, and 2) most of fine-grain synchronizations are successful (e.g. successful lock acquisition, and synchronized read).

1.4 Synopsis

The next chapter presents the background knowledge and previous work for this dissertation, including the description of the IBM Cyclops-64 architecture, and the review of synchronization techniques proposed and used in last decade. Chapter 3 introduces the experimental infrastructure used for this dissertation. Chapter 4 presents the performance study, evaluation, and customization of the synchronization mechanisms on the IBM Cyclops-64 architecture. Chapter 5 proposes the SSB-based fine-grain synchronization solution for large-scale chip architectures. Chapter 6 demonstrates the effectiveness and efficiency of SSB by benchmarking on the simulator of IBM Cyclops-64 chip architecture. Chapter 7 concludes this dissertation.

Chapter 2

BACKGROUND AND PREVIOUS WORK

2.1 Synopsis

This chapter presents the background knowledge and previous work for this dissertation. Section 2.2 discusses the limits of single core based conventional microarchitecture. Section 2.3 introduces the multi-core architecture, including detailed description of the IBM Cyclops-64 architecture (Section 2.3.4), which is the target architecture for the research in this thesis. We then review a wide range of hardware/software based synchronization mechanisms from Section 2.4 to Section 2.9. Section 2.10 describes several synchronization mechanisms currently supported on the Cyclops-64 platform.

2.2 The Limits of Single Processor Chip Architecture

Advances in IC processing technology allow the feature size to continue dropping. As a result, the density of transistors on silicon chips are to continue increasing for the next years following Moore's Law [126]. However, it is increasingly clear that the huge number of transistors that can be put on a single chip (now reaching 1 billion and continues to grow) can no longer be effectively utilized by traditional microprocessor technology that only integrates a single processor on a chip.

Based on the data collected for two major families of microprocessors [149, 62, 41], IA32 architecture family from Intel and Power architecture family from IBM, Figure 2.1 shows that the technique of scaling performance of microprocessor by increasing clock rate has reached the point of diminishing returns, beyond which the increase of the

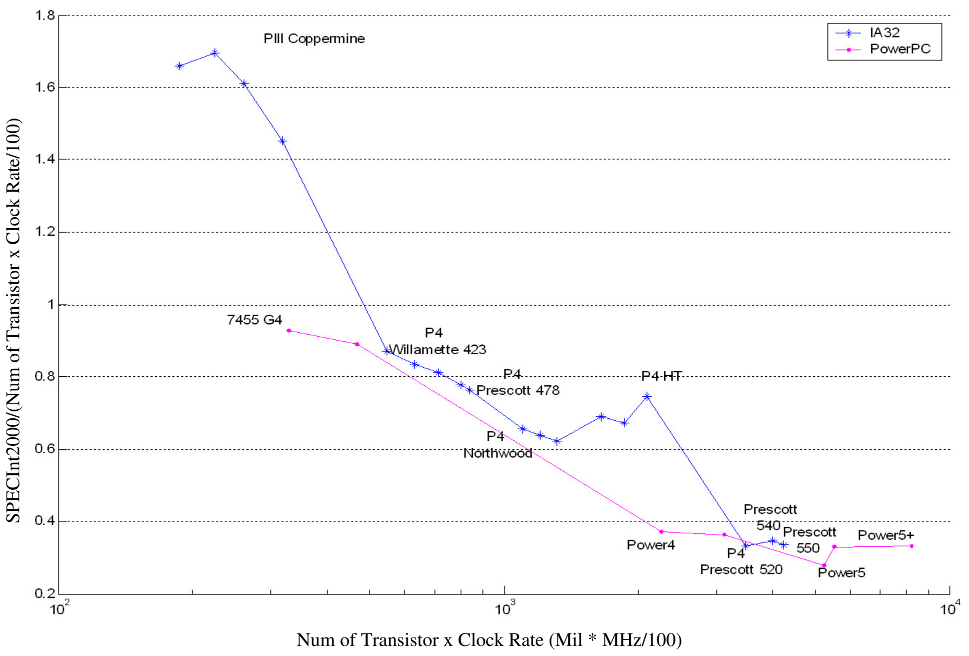


Figure 2.1: Diminishing Returns of Two Microprocessor Families

number of transistors fabricated on a single die yields less than proportional increases in performance. Although the increases in integration of transistor density sustained microprocessor performance growth for the last twenty years, the single-core based microprocessor architectures can no longer exploit resources effectively.

There are three fundamental limits of the single processor chip architecture:

- The reduced feature size does not only allow increasing density of transistors on silicon chips, but also causes shrinking wires, which suggests increasing wire delays. As the trend continues, this physical limitation of wire scaling is inevitable. In his study of on-chip interconnect delay, Doug Matzke predicted that only a small percentage of the chip can be reachable during a single clock cycle by 2012 [115]. The increasing wire delays limits the performance increase of a single complex core (e.g. Superscalar) based chip architecture [130], thus forces partitioning of on-chip hardware resources.

- One major bottleneck to improving performance is the “memory wall” problem [163]: the speed of processor increases faster than the speed of memory. Hence, the latencies for accessing memory are continuously increasing. Now it is common that the DRAM latency can be hundreds or thousands cycles [94]. One approach is to increase the memory bandwidth such that multiple outstanding memory accesses can be served at the same time. However, single-core based microprocessors can not effectively make use of memory width, since it is very rare to see more than a few speculative memory accesses bing performed simultaneously on conventional microprocessors [94].

- Most single-core processors are designed to exploit instruction level parallelism (ILP) in programs. ILP approach has a great advantage that it allows multiple instructions to execute simultaneously, but is still transparent to the programmer. However, the amounts of ILP that can be exploited in many applications are limited [80, 130]. Moreover, there is high-level parallelism naturally in applications that can not be exploited by ILP-based approaches [80]. Those high-level parallelism, such as thread-level parallelism and process-level parallelism, are expected to be widespread in future applications and multiprocessor-aware operating system and environments [75].

- Power consumption, both dynamic and static, has become one of the first-order design constraints in microprocessor architecture design. The power consumption of a microprocessor follows the almost cubic dependence on the clock frequency [65], so that it increases dramatically with growth in frequency. As a result, given the constraints on power efficiency, it becomes difficult to extract performance by just increasing clock rate for single core microprocessors.

Given the continuous advances in VLSI technologies and the fundamental limits of single-core based microprocessor design, it is important to exploit alternative design approaches for the new and next generation of microprocessors.

2.3 Multi-Core Chip Architectures

To address the limits presented for single-core based microprocessor designs, a new generation of technology is emerging by integrating a large number of tightly-coupled simple processor cores on a chip empowered by parallel system software technology that will coordinate these processors toward a scalable solution. This new approach puts a complete multiprocessor, including both CPUs and memory, on a single chip, which is known as multi-core architecture. Multi-core architecture is also called multiprocessor-on-a-chip architecture, or chip multiprocessor (CMP). Instead of devoting the entire die to a single and complex processor, multi-core chip architecture design integrates a number of simple processor cores on a single die.

It is believed that multi-core architecture has many advantages over the single processor chip:

- By partitioning the chip resources into individual small, localized simple cores, the effect of the interconnect delay is limited.
- By enabling multiple cores to share the chip resources, such as on-chip memory (or L2 cache), interconnect network, and on-chip/off-chip memory bandwidth, the resource utilization is improved.
- Given the fact that chip power consumptions drop significantly with reductions in frequency, multi-core architectures can alleviate the power consumption dissipation problem without reducing the computation capability by running multiple cores with moderate clock rate [150].
- Multi-core chip architectures naturally exploit thread-level and process-level parallelism, which are expected to be widespread in future applications and multiprocessor-aware operating system and environments [75].

Not surprisingly, all major microprocessor manufacturers have already begun to move their microarchitecture towards multi-core designs, and announced their roadmaps to bring multi-core based chip to the market [27, 89, 13, 96, 101, 18, 19, 113, 39].

2.3.1 Two Types of Multi-Core Architecture Designs

Unlike traditional uniprocessor chips where few architecture designs (for instances, Superscalar, and VLIW) dominate, researchers in multi-core chips have not yet reached (or even come close to reaching) a consensus on an architectural design that will be successful in the future. Recently we have seen many proposals from both the industry and academia actively exploiting the design space of multi-core chip design. In our vision, the current multi-core architecture designs can be categorized as two types [61, 100].

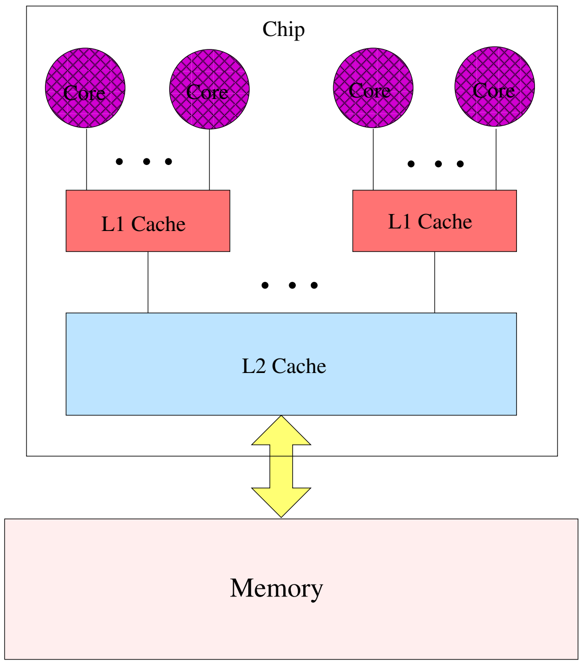


Figure 2.2: Hierarchical Multi-Core Design With Heavy Cores and Multiple Level Cache

- The first type glues together existing Superscalar/VLIW processor cores (heavy cores) into a single chip with only minor changes. Figure 2.2 shows a hierarchical multi-core design with a number of heavy cores that communicate through cache. This is the approach that is now taken by major microprocessor manufacturers. For example, Intel Core Duo Processor [36], AMD Opteron dual-Core processor [12], IBM Power5 dual-Core processor [95], and many others that are now available on the market, can be attributed to this type of multi-core design.

- Instead of being based on the existing microprocessor design, the second type of multi-core design takes a different approach to completely redesign the chip architecture by exploring the parallel architecture design space and searching for the most suitable chip architecture models. Figure 2.3 shows a multi-core design with many simple cores and on-chip memory modules connected through an on-chip interconnection network. IBM Cell processor [72, 71, 55], IBM Cyclops-64 chip [52, 53], Intel Tera-Scale researcher chip [91], and some others are now exploring this type of multi-core design. In this dissertation, the research targets the second type of multi-core chips.

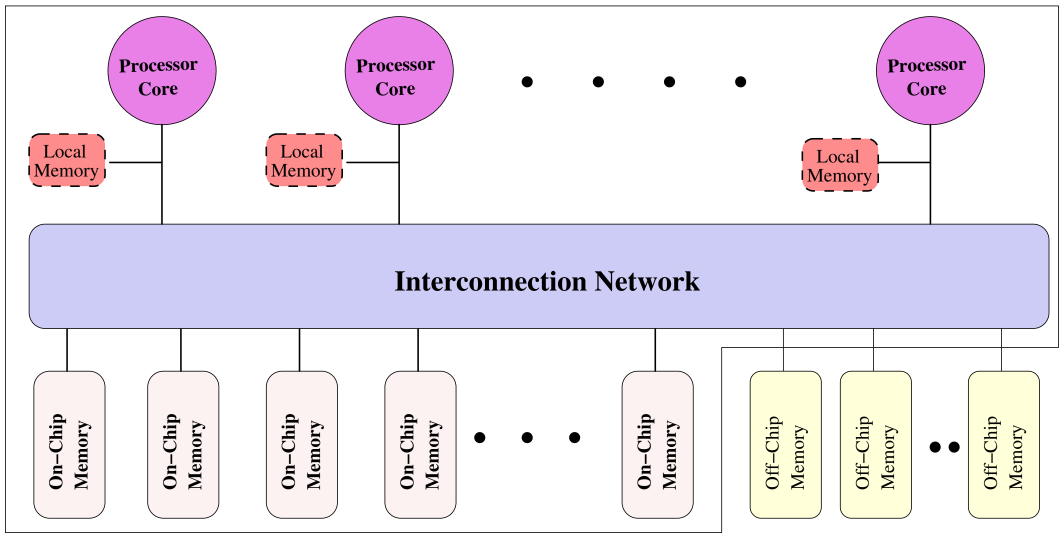


Figure 2.3: Multi-Core Design with Many Simple Cores and on-chip Memory Modules

2.3.2 Large-Scale Multi-Core Chip Architecture

While the dual-core microprocessors begin to dominate the market of servers and personal computers, both the industry and academia are exploiting the design space of the future multi-core architectures by integrating a large number of cores (10s and beyond) into a single chip, which is called large-scale multi-core chip (or many-core chip) [46, 27]. For example, Intel recently announced its research prototype many-core design with 80 cores on a single die [160]. Another example is the IBM Cyclops-64 chip architecture, which supports 160 hardware thread units in one chip [52, 53]. A recent technique report from University of California, Berkeley predicts that 1000 cores on a die can be achieved when 30nm technology is available [21].

Throughout this dissertation, we will use the term *large-scale multi-core architecture* to refer to multi-core chips with a large number of processing cores (10s and beyond).

2.3.3 Cellular Architecture

The IBM BlueGene/C architecture (also known as Cyclops) pioneers a new class of massively parallel architecture, based on advanced multi-core chips, called cellular architecture [7, 10, 8, 33]. The design of the Cyclops architecture is based on three main principles [33]:

- a cellular approach is used to build the system at various levels, from chips to large systems.
- the integration of processing logic and memory in the same piece of silicon;
- the use of massive intra-chip parallelism to tolerate latencies;

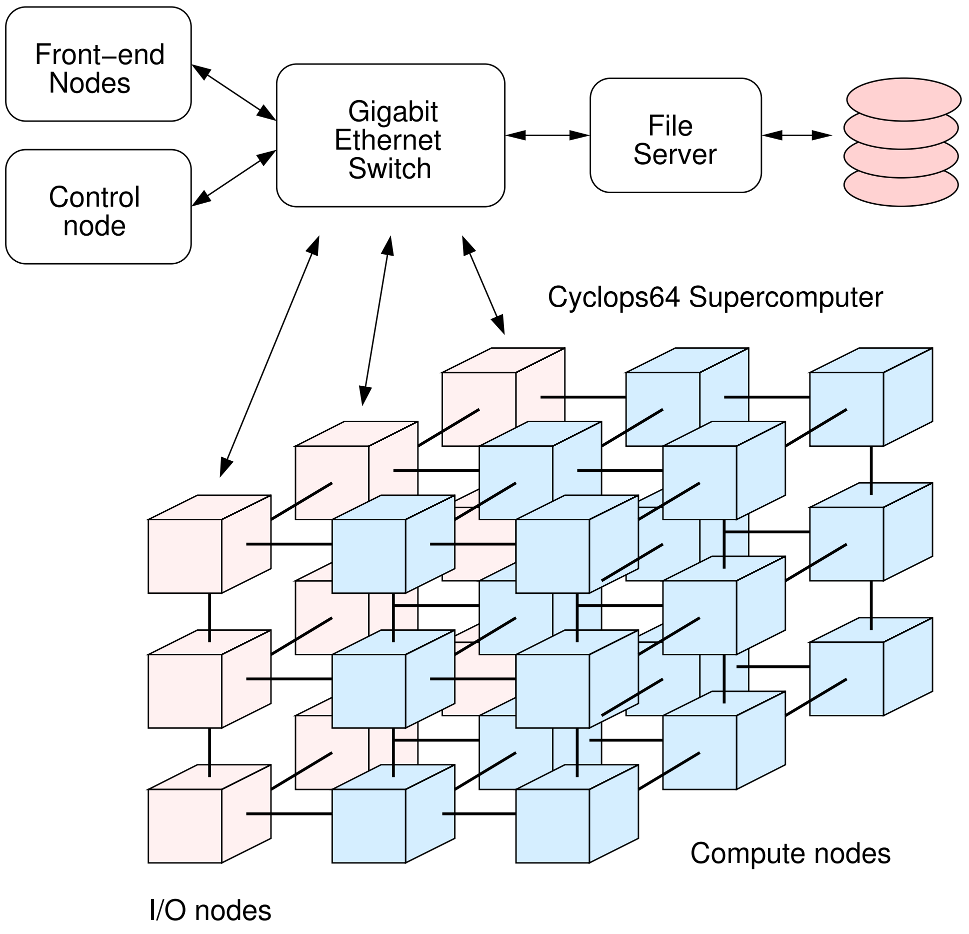
At the chip level, the cellular architecture uses a cellular organization interconnecting a large number of very light-weight processors (called processing cells, or cells). Instead of out-of-order, wide issue Superscalar approach, the processing cell is very simple by only employing simple in-order issue to reduce hardware complexity. A thread running on the processing cell carries very little state. The on-chip communication network provides rich interconnection and sufficient bandwidth for inter-cell communication and synchronization among the processing cells and shard memory. The processing cell is normally running at a modest clock rate. Although the performance of each individual cell is not particularly high, the aggregate performance of a whole chip is much higher than a conventional microarchitecture with the same amount of transistors. Large scalable supercomputing systems can be built with a cellular approach using such chip as a building block [33]. In other words, within a chip, a processor is viewed as a cell, whereas, within a system, a chip is viewed as a cell.

An example of the cellular architecture, is the Cyclops architecture [10, 8, 33]. The Cyclops chip integrates 128 *32-bit* processing cores (thread units), each four of which share a floating point unit. For the memory hierarchy, all thread units share 16 on-chip 512KB DRAM banks, and each four of the thread units share a 16-KB data cache. A Cyclops chip provides six input and six output links, which allow a chip to be connected in a 3D topology. The links together give a maximum I/O bandwidth of 12GB/s. A large system can be built by connecting many Cyclops chips together using the links directly without additional hardware.

2.3.4 IBM Cyclops-64 Cellular Architecture

The target multi-core architecture in this thesis is the Cyclops-64 architecture, which is evolved from the Cyclops architecture. Cyclops-64 (C64) is the latest version of the Cyclops cellular architecture designed to serve as a dedicated petaflop computing engine for running high performance applications [52, 53]. A C64 supercomputer is attached — through a number of Gigabit Ethernet links — to a host system. The host system provides a familiar computing environment to application software developers and end users.

A C64 is built out of tens of thousands of C64 processing nodes arranged in a 3D-mesh network (see Figure 2.4). Each processing node consists of a C64 chip, external



Courtesy: Juan del Cuvillo, and Ziang Hu

Figure 2.4: Cyclops-64 Computing Environment

DRAM, and a small amount of external interface logic. A C64 chip employs a large-scale multi-core design with a large number of hardware thread units, half as many floating point units, embedded memory, an interface to the off-chip DDR SDRAM memory and bidirectional inter-chip routing ports (see Figure 2.5).

A C64 chip has 80 processors, each with two thread units, a floating-point unit and two SRAM memory banks of approximately 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The C64 chip has no data cache.

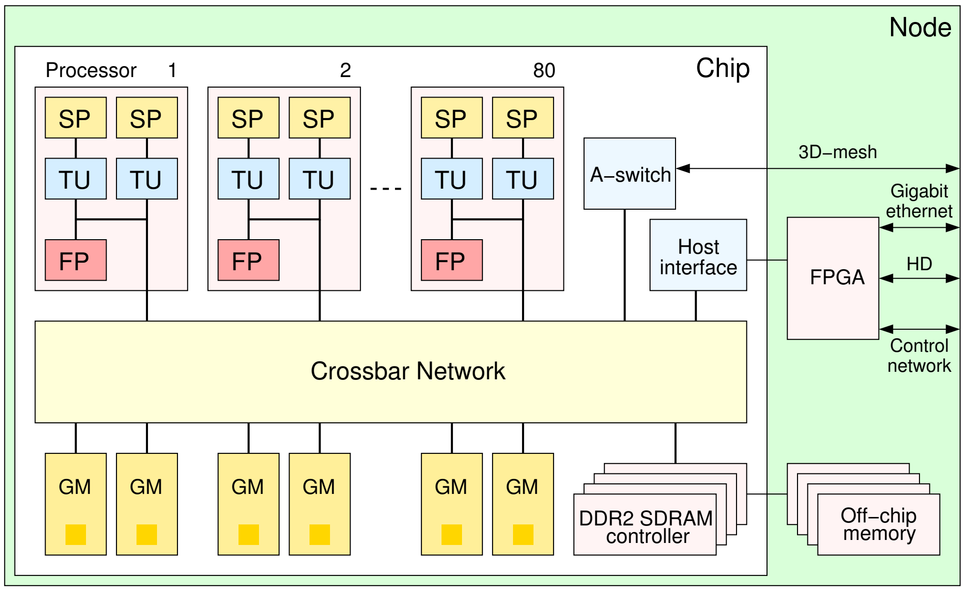
Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections of SRAM together form the global memory (GM) that is uniformly addressable from all thread units. All memory words are 8 bytes wide and the memory is byte-addressable. The memory accesses to contiguous address space are interleaved. For example, the access to GM is interleaved to SRAM banks by a 64-byte boundary, which ensures the full utilization of the bandwidth to all memory banks.

On-chip resources are connected to a 96×96 crossbar network, which sustains all the intra-chip traffic communication and provides access to the routing ports that connect each C64 chip to its nearest neighbors in the 3D-mesh network. The intra-chip network also facilitates access to special devices such as the Gigabit Ethernet port and the serial ATA disk drive attached to each C64 node.

C64 chip architecture contains a large amount (160) of thread units and one can exploit such massive intra-chip multithreading by maintaining a large number of active threads. A preemptive thread model can incur high context-switching cost for two reasons. First since on-chip memory is precious and limited, saving the context of a large number of threads in on-chip memory can be prohibitively expensive or impossible. Second saving the context in off-chip memory suffers from high latency and low bandwidth. Therefore, C64 supports a non-preemptive thread model: the core on which a thread is running is simply made idle when the thread is suspended.

The C64 instruction set architecture incorporates efficient support for thread level execution. For instance, it provides a sleep instruction, such that a thread can stop executing instructions for a number of cycles or indefinitely. If a thread is expected to wait on an external event or synchronization, i.e. a long-latency operation, it would be judicious to put the thread to sleep and get notified as soon as the long wait is over. A thread is woken up by another thread through a hardware interrupt/signal. Such a wakeup signal is generated when a store into a memory-mapped port is executed. This operation takes as little as 20 cycles when there is no contention in the crossbar network. Additionally, a rich set of hardware supported in-memory atomic operations is available to the programmer. Locks and mutexes can be efficiently implemented using this type of instructions. When an atomic operation is executed in C64 architecture, the crossbar network only blocks the memory bank where the atomic instruction is operating. Meanwhile, the remaining on-chip memory banks operate normally.

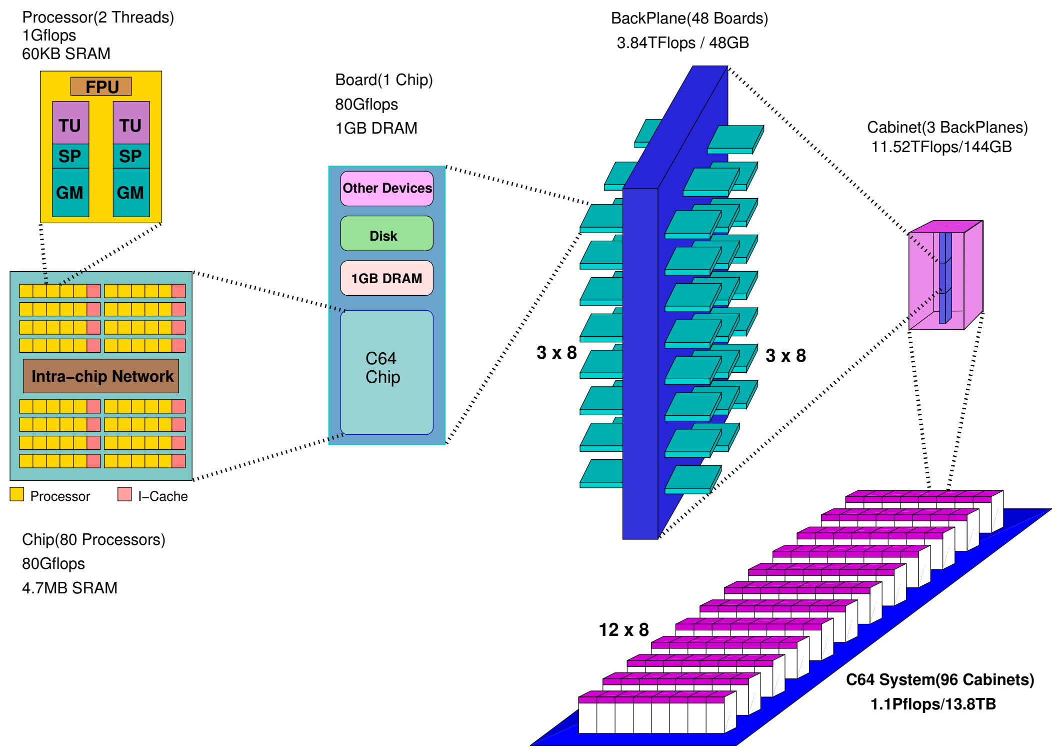
In regard to intra-chip communication bandwidth, each processor within a C64 chip is connected to a crossbar network that can deliver 4GB/s per port, totaling 300GB/s in each direction. The bandwidth provided by the crossbar supports intra-chip communication, i.e. access to other processor's on-chip memory and off-chip DRAM, as well as inter-chip communication via the A-switch device, which connects each C64 chip to its neighbors in the 3D-mesh.



Courtesy: This figure was first created by Alban Douillet and then revised by Juan del Cuvillo. Information in the figure is provided by Monty Denneau.

Figure 2.5: Cyclops-64 Chip Architecture

Finally, Figure 2.6 illustrates an instance of a C64 supercomputer architecture with $24 \times 24 \times 24$ logically arranged C64 nodes in the 3D-mesh configuration. Notice the physical distribution is somewhat different.



Courtesy: This figure was first created by Juan del Cuvillo and then successively revised by the author of this dissertation and Long Chen. Information in the figure is provided by Monty Denneau.

Figure 2.6: Cyclops-64 Supercomputer

In summary, the C64 chip architecture represents a major departure from mainstream microprocessor design in several aspects:

1. The C64 chip integrates multiple (160) processing elements, embedded memory and communication hardware in the same piece of silicon.

2. A thread unit (TU), the C64 computational cell, is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate (500MHz).

3. C64 employs a non-preemptive thread execution model, thus does not support context switch. The kernel will not interrupt the user thread running on a thread unit unless the user explicitly specifies termination or an exception occurs.

4. C64 incorporates efficient support for thread level execution. For instance, a thread can voluntarily stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through a hardware interrupt. When a thread stops its execution, the state of the thread, i.e., the context, is not switched out. The thread unit just keeps idle until it is awaken up. On a common commodity processor, when a thread is preempted from the processor after context switching, we regard this thread as being "sleeping". In C64, since a thread only goes to idle without context switching, it can be efficiently waked up through hardware interrupt, and it can resume execution immediately right after it receives the wakeup signal, we regard the thread as being "napping". All the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers.

5. The C64 features a three-level (Scratchpad (SP) memory, on-chip SRAM, off-chip DRAM) memory hierarchy without data cache. Instead a portion of each thread unit's corresponding on-chip SRAM bank is configured as the scratchpad memory (SP). Therefore, the thread unit can access to its own SP through a dedicated data path with very low latency, which provides a fast temporary storage to exploit locality under software control. The remaining sections of all on-chip SRAM banks together form the global memory (GM) that is uniformly addressable from all thread units. It is worth noting that both the SP and GM are globally addressable through the crossbar network by all TUs. However, a TU can access its own SP with very low latency through a dedicated data path as a "backdoor". When a TU reads from/write to the GM or the SPs of other thread units, the access goes through the crossbar network. In other words, for a TU, the SP of another TU is treated the

same as the GM. There are 4 off-chip memory controllers connected to 4 off-chip DRAM banks, which is also globally addressable by all TUs. The current design size for DRAM is 1GB. Figure 2.7 shows the latency and bandwidth for accessing different segments in the C64 memory hierarchy.

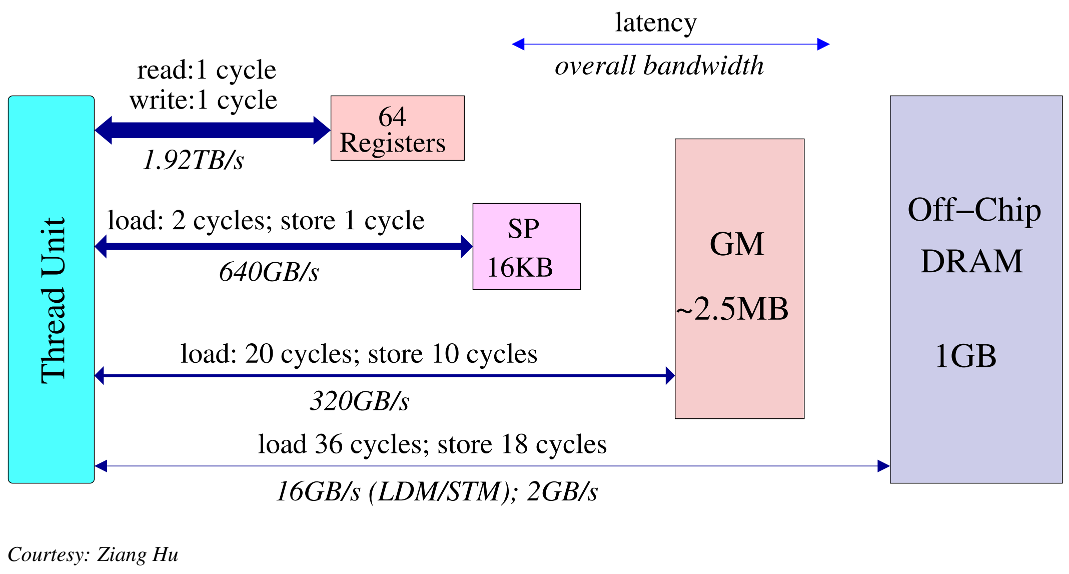


Figure 2.7: Cyclops-64 Memory Hierarchy

6. In C64, there is no hardware virtual memory manager. The three-level memory hierarchy of the C64 chip is explicitly *visible* to the programmer.

7. In C64, all on-chip resources are connected to an on-chip crossbar network, which sustains a 384 GB/s bandwidth per direction in total. The crossbar network also guarantees that C64 chip architecture is *sequentially consistent*. Thus, there is no need to issue fence-like instructions after each memory operation to ensure the order between them [168].

2.4 Synchronization

In a multiprocessing/multithreading environment, coordination of concurrent tasks is called synchronization. Synchronization controls concurrent access to shared data and resources, or enforces certain ordering between threads. For a parallel system, synchronization is critical for achieving scalable performance.

According to the classification in [45], there are mainly three types of synchronization operations: *mutual exclusion*, *point-to-point event*, and *global event*.

For a multithreading program:

- *Mutual exclusion* enforces that a number of operations on certain shared resources (for instance, shared data) are performed by only one thread at a time. Mutual exclusion is normally achieved with lock/unlock operations.
- *Point-to-point event* is used to enforce the certain dependencies (for instance, read-after-write data dependencies) between threads.
- *Global event* involves a group of threads. It is normally known as *barrier*, which enforces a group of threads to stop at certain point without proceeding until all the threads in the group reach the point.

In the subsequent sections, different techniques that are used to implement synchronization operations, will be reviewed.

2.5 Atomic Instructions

In addition to atomic read/write, current mainstream processor architectures normally support a set of atomic memory instructions. In the research literature and practice, those atomic instructions are the basic primitives used to implement software spin lock algorithms (see Section 2.6.1), lock-free concurrent objects (see Section 2.7.1), and software transactional memory systems (see Section 2.7.2.2). In this section, we briefly review widely available atomic primitives in nowadays processor architectures.

For simplicity and clarity, the operational semantics of atomic primitives will be introduced using pseudocode. Please be aware that in the pseudocode, the entire function is executed atomically: no other processes can interrupt the execution of the function and observe an intermediate state during the execution of the function.

fetch_and_store:

```
boolean fetch_and_store(boolean var) {
    boolean old = var;
    var = true;
    return old;
}
```

The *fetch_and_store* primitive can actually be regarded the same as the *test_and_set*.

fetch_and_increment:

```
integer fetch_and_increment(integer var) {
    integer old = var;
    var = var + 1;
    return old;
}
```

atomic_operator:

```
atomic_operator(integer var, integer const) {
     var = var operator const;
}
```

The “operator” in the pseudocode can be $+$, $-$, and, or, xor, etc.

compare_and_swap (CAS)

CAS is introduced on the IBM System 370 [34]. It is now supported on Intel (IA-32 and IA-64) and Sun SPARC architectures.

```
boolean compare_and_swap(type var, \
                         type old, type new) {
     if( var == old){
         var = new;
         return true;
     }
     else{
         return false;
     }
}
```

In the pseudocode, var, old, and new are type consistent. CAS is the most important atomic primitive, which is intensively used to design lock-free data structures and software transactional memory systems.

load_linked/store_conditional (LL/SC):

LL and SC are supported on PowerPC, MIPS, and Alpha architectures.

```
type load_linked(type var){
     return var;
}

boolean store_conditional(type var, type new){
     if( var is not updated since last LL){
          var = new;
          return true;
     }
     else
          return false;
}
```

In the pseudocode, var, old, and new are type consistent. The CAS primitive can be easily implemented using LL/SC:

```
boolean compare_and_swap(type var, \
                         type old, type new) {
     if( load_linked(var) == old ){
         return store_conditional(var, new);
```

```
        }
        else{
            return false;
        }
    }
```

Most current mainstream processor architectures support either CAS or LL/SC on aligned single words. Support for CAS or LL/SC on aligned 64-bit blocks is available on both 32-bit and 64-bit architectures. However, wider block sizes than 64-bit is normally not supported even on 64-bit architectures. Moreover, no processor architecture can support the ideal semantics of LL/SC in practice. "None allows nesting or interleaving of LL/SC pairs, and most prohibit any memory access between LL and SC" [122].

double_compare_and_swap (DCAS):

```
    boolean double_compare_and_swap(type var1, \
                                    type old1, \
                                    type new1, \
                                    type var2, \
                                    type old2, \
                                    type new2) {
        if( var1 == old1 && var2 == old2){
            var1 = new1;
            var2 = new2;
            return true;
        }
        else{
            return false;
        }
    }
```

In the pseudocode, $var1$, $var2$, $old1$, $old2$, $new1$, and $new2$ are type consistent. Although assumed in many research literatures, DCAS is not actually supported on any current processor architecture. Simulating DCAS with weaker CAS or LL/SC primitives causes prohibitive performance overheads.

2.6 Spin Lock

Spin lock is one of the most widely used synchronization primitives in parallel programming. Spin lock is usually used to achieve mutual exclusions, which resolve conflicting accesses to shared resources by concurrent processes or threads. Spin lock algorithms adopt a busy-wait approach to repeatedly test one or more shared variables to determine when forward progress can be made.

Several general performance goals for designing lock algorithms is given by Culler et. al. [45]:

Low latency: In the absence of contention, process should be acquire the lock with low latency.

Low traffic: In case of high contention, the lock algorithm should generate as less memory traffic or bus transaction as possible.

Scalability: The latency and traffic should not increase quickly with the increase of number of processors.

Low storage cost: The memory usage of the lock algorithm should be small and not increase quickly with the increase of number of processors.

Fairness: Avoid starvation and live-lock.

2.6.1 Software Based Locking Algorithms

During the last two decades, the spin lock algorithms for shared memory multiprocessors system have been intensively studied in the literature. Research focus on how to design the lock algorithm such that the "spin variable" can be accessed by generating as less memory traffic as possible, meanwhile still keeps a low overhead of the algorithms itself.

2.6.1.1 Test_and_Set Lock

The simplest algorithm is the *test_and_set* lock, which make use of the atomic *test_and_set* instruction to repeatedly access a boolean flag (see Figure 2.8).

```
boolean flag;

flag = false; /* initialized  as false */

acquire_lock:
          while(fetch_and_store(flag) == true);
```

Figure 2.8: Test_and_Set Spin Lock

Since the single shared boolean flag is repeatedly tested by all concurrent threads, the performance degrades dramatically with the increase of number of threads. *Test-and-test_and_set* [143] approach and different backoff schemes [16] are used to alleviate the high contention for accessing the flag. Because of its simplicity, *test_and_set* lock has a very low overhead in the absence of contention.

2.6.1.2 Ticket Lock

Using the atomic *fetch_and_increment* instruction, a ticket lock can be implemented. The ticket lock uses two counters: one is used to count the number of releases of a lock and the other is used to count the number of lock requests. When a processor acquires a lock, it obtains a request number (ticket) by performing a *fetch_and_increment* operation on the request counter. When a processor releases a lock, it increments the release counter by one. Each waiting thread is spinning on the release counter until it is equal to the ticket.

Compared with *Test_and_set* lock, ticket lock reduces the number of atomic memory instructions by only allowing one thread to acquire the lock when it is available. Moreover, ticket lock also ensures that the lock is granted in the order that it is requested (FIFO). However, ticket lock still causes memory contention due to spinning on a common memory location – the release counter. Backoff mechanism can also be used to reduce the memory traffic.

2.6.1.3 Array Based Locks

In order to reduce the amount of memory traffic and bus transactions in the presence of high contention, array based queueing locks [16, 68] are introduced. Instead of polling a common memory location, each processor spins on a different location by using array-based lock. In both algorithms, the processor uses the atomic operation to obtain the address of a location to spin on and it is ensured that each processor gets a different location. In a cache coherent multiprocessors system, each processor can spin locally on different cache line in its private cache. In a shared memory system without data cache or a distributed shared memory system, since the order of lock request can not be predetermined, the spin variable can not be statically allocated such that all processor can spin locally. Cyclops-64 chip architecture is such an example. Array based locks also introduce more storage cost than the test_and_set and ticket lock, since an array as large as the number of processors needs to be allocated for each lock.

2.6.1.4 Linked-List Based Lock

Mellor-Crummey and Scott improved the array-based queueing locks with a linked-list-based queueing lock (MCS) that requires a small constant amount of space per processor and ensures that the each processor can spin locally in both cache-coherent and distributed shared memory systems [118]. In MCS, the lock release requires a strong atomic instruction – *compared_and_swap*, which may not be available everywhere. To overcome this problem, they also presented an alternative release procedure that requires only *fetch_and_store* atomic instructions. However, this variant is not starvation-free.

2.6.1.5 Reactive Lock

The *test_and_set* is simple and efficient in the absence of contention, however the performance degrades dramatically under high contention. On the contrary, the queueing locks, such as MCS lock, reduces memory traffic and bus transactions in the presence of high contention on the lock. However, queueing lock has a much higher latency and overhead than *test_and_set* lock when contention is low or absent. Lim and Agarwal [110] presents a reactive synchronization algorithm, which is able to dynamically and adaptively switch between *test_and_set* lock and MCS lock according to runtime contention level. However, their reactive algorithm relies on the so called unique "consensus object" to complete the lock algorithm switch. For a distributed shared memory system without cache, the "consensus object" itself introduces contention. Therefore, this reactive algorithm is not applicable to multiprocessors system without cache, such as Cyclops-64. Even for a cache coherent multiprocessors system, enforcing all processors continuously access the system object may introduce large amount of bus transactions for cache invalidation.

2.6.2 Hardware Based Locking Mechanisms

Hardware solutions, such as hardware queue based QOLB [93], MAOs on SGI Origin [107], lock box [155] for SMT processor, SoC lock cache [6], AMO [166] and others, normally deliver much better performance than the spin-lock algorithms developed in software with the complexity of hardware design and cost. We will review some of hardware locking schemes in this subsection.

2.6.2.1 QOLB

Queue-On-Lock-Bit primitive (QOLB) is a distributed, queue-based locking scheme directly supported by hardware [66, 93]. In the proposal of QOLB, each memory line [1] is associated with a *synchronization bit (syncbit)*. The synchronization operation (e.g. lock acquisition) is performed on the syncbit. With QOLB, the waiting processors are kept as a queue in the cache line. In the queue, pointers to adjacent queue entries are held in the cache line. Waiting processors spin locally on a "shadow" copy of the line

[1] The term "line" implies the aligned unit of memory over which consistency is maintained [66].

in the local cache. By enabling local spinning, QOLB prevents unnecessary interconnect traffic or interference with the lock holder. Since the syncbit is associated with a memory line, i.e., the shared data, QOLB allows collocation – shared data can be transferred to the waiting processor at the same time with the lock hand-off.

The disadvantage of QOLB is mainly its hardware cost. It complicates the cache-coherence protocol design with additional states. It needs direct cache-to-cache transfer mechanism during the lock hand-off from releaser to acquirer. QOLB also requires the capability for multiple nodes to perform operations on the same address (the "shadow line") without invoking the cache coherence protocol. Moreover, the benefit of collocation is dependent on the cache line size. If the shared data does not fit into the cache line, it can not benefit from collocation. The number of QOLB operation on per memory line is also limited to one.

2.6.2.2 SMT Lock-Box

Lock-box was proposed on simultaneous multithreaded (SMT) processor to permit cheap inter-thread synchronization within the processor [156]. Lock-box, a small processor structure associated with a single function unit, has one entry per hardware thread context. Each entry of the lock-box contains the address of the lock, a pointer to the lock acquisition instruction, and a valid bit. In SMT, when a thread fails to acquire a lock, the lock address and the program counter (a pointer) are stored in the lock-box entry of that thread. The thread then is flushed from the processor. When the lock is released by another thread, hardware attempts to search lock-box entries using the released address. If a blocked thread is found, the hardware resumes the execution of the thread, and invalidates the corresponding lock box entry.

The common lock acquire and release primitives can be built on instructions accessing lock-box hardware. Unlike a software *test_and_set* scheme, failure in acquiring a lock with lock-box does not invoke any bus transactions. Instead, the thread is blocked, and re-scheduled later in the lock-box. The design and implementation of lock-box is efficient and inexpensive, because the threads in an SMT processor share the same scheduling core. It is clear that lock-box is not widely applicable to processor architectures other than SMT.

2.6.2.3 SoCLC: System-on-a-Chip Lock Cache

System-on-a-chip lock cache (SoCLC) is proposed as hardware support for efficient lock operation on SoC architectures [6]. Unlike QOLB [66, 93], SMT Lock-Box [156], and other hardware-based locking mechanisms, SoCLC is a processor/memory/cache hierarchy independent hardware solution. The implementation of SoCLC does not require any architectural modifications and extensions, such as extended cache protocol, and extra cache lines/tags, etc., to the processor core.

SoCLC is basically a hardware array of bit entries. Within the SoCLC unit, each bit can be used to represent a lock. For example, an SoCLC with 256 entries can support up to 256 locks. Lock variables are allocated at a specific address range mapped to the address space of every processor. General load/store instructions are used in acquiring/release a lock in an atomic fashion in SoCLC. SoCLC hardware unit is connected to processors on the same SoC chip via the system bus. Each processor accesses the SoCLC unit to acquire or release lock variables by operating on the corresponding hardware bits.

Assuming an SoC with N processors, within the SoCLC, each lock variable (a bit) is associated with a set of N 1-bit locations, each of which stands for a processor. A boolean “1” in such a location indicates that the corresponding processor has unsuccessfully tried to acquire the lock and is waiting for the lock to be released. Therefore, when a lock is released, the associated N locations of the lock bit is checked in order to determine which processor is waiting for the lock. As a result, an interrupt can be sent to one waiting processor, which is blocked while waiting for the lock.

Although supporting a large number of hardware locks, the SoCLC takes an centralized approach: all the hardware locks are managed by a single SoCLC unit. If the number of on-chip processors is very large (10s or beyond), the contention of accessing the SoCLC unit will be very high. Therefore, the SoCLC itself becomes a performance bottleneck.

It is worth noting that the locks supported by QOLB, SMT Lock-box, and SoCLC hardware schemes still needs to be mapped to software allocated/managed lock variables. Although these hardware mechanisms provides efficient alternatives for software spin-locks, they does not provide the data-level fine-grain synchronization capabilities as the full/empty bits like hardware solutions, which will be briefly reviewed in Section 2.8.

2.7 Non-Blocking Synchronization

Non-blocking synchronization algorithms have been studied and developed as a way of avoiding the *wait* (or *spin*) to gain access to a concurrent object during contention. A synchronization algorithms is non-blocking if the suspension or failure of any number of threads cannot prevent the remaining threads from making progress [59]. A non-blocking synchronization algorithms allow asynchronous and concurrent access to concurrent objects, but still guarantees the consistent updates. In contrast, blocking synchronization algorithms serialize the access to concurrent objects using mutual exclusive critical sections.

Non-blocking algorithms can be classified into three categories according to their algorithmic progress guarantees:

- **wait-freedom** is the strongest guarantee: even if experience contention, *all* threads can make progress in a finite number of their own time steps [81]. With wait-freedom, neither deadlock nor starvation can happen.

- **lock-freedom** is a weaker guarantee: even if there is contention, *at least one thread* makes progress in a finite number of its own time steps. With lock-freedom, deadlock can be avoided, but not starvation.

- **obstruction-freedom** is the weakest guarantee: in the absence of contention, a thread makes progress in a finite number of its own time steps [82]. With obstruction-freedom, deadlock can be ruled out. However, mechanisms, such as backoff, have to be used to avoid live-lock.

In this section, we will review major works for achieving non-blocking synchronization.

2.7.1 Lock-Free Concurrent Data Structures

In a shared memory multi-programming and multi-threading environment, spin lock techniques reviewed in Section 2.6.1 can be employed to achieve mutual exclusion to resolve conflicting accesses to shared resources. However, the use of locking technique causes many problems and limitations [84]:

- *Priority Inversion:* The higher priority processes are waiting for the lock held by a preempted lower priority process.
- *Convoying:* A process, which is hold the lock, is descheduled because of either exhausting its schedule quantum or some kind of interrupt. And other processes, which happen to be scheduled to run, have to wait for the release of the lock.
- *Deadlock:* Deadlock occurs if two processes are waiting for the other to release its lock, or more than two processes are waiting for locks in a circular chain. Deadlock avoidance mechanism is hard to be efficiently designed and implemented.

To avoid above problems and limitations of locking techniques, many lock-free concurrent data structures and algorithms are proposed in the literature. A lock-free concurrent data structure is "one that guarantees that if multiple threads concurrently access that data structure, then some thread will complete its operation in a finite number of steps, despite the delay or failure of other threads" [83].

Lock-free data structures/objects are designed and implemented in a way that allow multiple threads to read and write shared data concurrently without making it inconsistent. Hardware atomic instructions, especially CAS or LL/SC (see Section 2.5) is the basis to build lock-free data structures. The CAS instruction allows the algorithm to atomically (1) verify that a previous read shared data is not modified by any other threads; (2) in case of success, write the new version of value into the shared data. This is a general mechanism for an algorithm to read a datum from memory, modify it, and write it back only if no other thread modified it in the meantime.

Unlike the lock-based implementations, which is very straightforward, the lock-free concurrent data structures have to be very carefully designed and implemented to guarantee the correctness without introducing significant performance overhead. The difficulties mainly come from:

1. The current mainstream processor architectures only provide at most 64-bit wide CAS (or equivalent LL/SC) instruction. Even the simplest data structure employes memory storage beyond the capability of the CAS instruction. Therefore, the lock-free algorithm has to be correctly designed not to corrupt the data structure in the presence of concurrent access from multiple threads.

2. The implementation of the lock-free data structure should satisfy the *linearizability*. "Linearizability provides the illusion that each operation takes effect instantaneously at some point between its invocation and its response" [85].

3. The ABA problem is a fundamental problem for designing lock-free algorithms. This hazard is associated with the usage of CAS instruction. The ABA problem occurs when a thread reads a value A from a shared memory location, and then other threads change the value of the location to B, and then change back to A again. Later, when the original thread check the location, using CAS, in which the comparison succeeds, and then the thread erroneously writes a new value into

location under the assumption that the value of the location is not changed since its last read. As a consequence, the lock-free data structure's consistency may get corrupted [34, 121].

4. The reclamation of the memory occupied by removed nodes from lock-free data structures is a major concern for correct concurrent execution. "The memory reclamation problem is how to allow memory of removed nodes to be freed, while guaranteeing that no thread access free memory, and how to do so in a lock-free manner" [122]. This problem is not trivial for designing lock-free data structures. Recently there are two similar techniques independently developed by Michael [122] and Herlihy et. al. [83] to allow safe memory reclamation.

Due to the difficulty and complexity of designing lock-free concurrent data structures, most of the work has focus on lock-free version of specified basic data structures, such as stacks [154, 79], queues [154, 114, 159, 158, 123, 70, 5, 54, 120], and sets [106, 114, 159, 70, 78, 119], etc. It is worth noting that several lock-free data structure designs assume the presence of DCAS atomic instruction [114, 70, 5, 54], which is not supported by any current processor architecture. Therefore, those designs are not actually practical. And some early work is not aware of the memory reclamation problem or uses problematic memory management method. Also, Herlihy and Moss [84] states that experimental evidences suggests that in the absence of priority inversion, convoying, and deadlock, lock-free data structures often not perform as well as their locking-based counterparts. This claim is also confirmed by our experiments on Cyclops-64 architecture, which does not support preemption of threads.

2.7.2 Transactional Memory

By extending the transaction concept from transaction processing theory [69], which has been widely used in the design and implementation of database management systems, Herlihy and Moss [84] proposed transactional memory to facilitate general multithreading programming.

A *transaction* is finite sequence of instructions, that is executed by a single process/thread, and used to access or modify locations in shared memory. A *transaction* satisfies the *serializability* and *atomicity*, which are defined as:

- *Serializability:* Even though transactions execute concurrently, they appear to execute *as if* in a one-at-a-time order. This means that the steps of one transaction never appear to be interleaved with the steps of other transactions.

- *Atomicity:* A transaction's changes to shared memory are atomic: either all happen or none happen (All-or-Nothing). A transaction either completes and commit, making its changes take effect instantaneously, or aborts, discarding all its changes.

Transactional memory allows programmer to define a finite sequence of read-modify-write operations to multiple, independent shared memory locations as a transaction, which is serializable and atomic. Transactional memory provides a programming paradigms simpler than lock-based critical section by specifying atomicity without lock assignment task. Transactional memory allows transactions to execute concurrently in a look-free manner, and rollback due to dynamic inter-transaction conflicts.

In summary, the transactional memory system attempts to achieve:

- *Productivity:* as simple synchronization paradigm as a coarse-grained global lock;

- *Scalability:* as much parallelism as fine-grained locks based implementation;

- *Efficiency:* lower overhead than fine-grained locks.

2.7.2.1 Hardware Transactional Memory

Researchers propose efficient hardware support for synchronization via transactions, which is supposed to be efficient while still keeps a simple programming paradigms.

Herlihy and Moss [84] proposed to build transactional memory by extending standard multiprocessor write-invalidate cache coherence protocols. In their design, transactions executes speculatively by buffering intermediate states in the cache. Once upon detecting the data conflicts through the enhanced cache coherence protocol, the speculative execution of transaction rolls back and discards all its states. If the transaction finishes without detecting any conflict, it commits and makes its updates to shared memory. There are two limitations for this design: (1) an important limitation is that the size of the a transaction's state is limited by the fixed hardware cache size; (2) a transaction executes only once. In case of abort due to conflicts, it is programmer's responsibility to retry the transaction.

Rajwar and Goodman propose *Speculative Lock Elision (SLE)* [139] and *Transactional Lock Removal (TLR)* [140] to leverage the state-of-the-art microarchitecture speculative techniques to buffer speculative register and memory states. Upon an abort, the transaction is able to restore its register states as well as memory states. Therefore, if a transaction's states exceed available hardware resources, it automatically aborts, restore the states, and re-executes by acquiring a lock. Their design ensures backward compatibility with old codes using locking techniques. The hardware executes the lock based critical section in a lock-free (transactional) manner without any code change.

To address the limitation of Herlihy and Moss's original design, that the size of transaction's states can not exceed that of cache's, Ananian et. al. propose the *Unbounded Transactional Memory (UTM)* [14]. UTM supports transactions whose memory footprint can be as large as virtual memory. The same view is shared by the *Virtual Transaction Memory (VTM)* [141], which is proposed to virtualize transactional memory in the same way that virtual memory virtualize the physical memory. VTM system transparently hides resource exhaustion both in space (cache size) and time (schedule quantum). However, both UTM and VTM assume hardware argumentation far from modest. Major functionalities required in the designs are not provided by any existing processor architectures.

Thread-Level Transactional Memory (TTM) proposes the use of thread-level log as thread-private memory to allow multiple concurrent transactions to store both the new and old value without updating the main memory. In the event of transaction aborts, the thread can be easily recovered based on the log. The TTM's thread-level log is stored in a cacheable thread virtual address space, allowing transactions to be tied to thread instead of the processor, and independent of cache hardware limits. Again, the hardware and operating system requirements to implement TTM is not modest.

Stanford's *Transactional Memory Consistency and Coherence (TCC)* [73, 74, 76, 117] employs an approach similar to database management systems. Unlike all other transactional memory proposals, TCC requires all code must resides in a transaction. In TCC, the transaction becomes the basic unit of parallel work, communication, memory coherence, and memory consistency [73]. As a result, rather than extends multiprocessors' cache coherence protocol and consistency model, TCC re-defines a new transaction-grained coherence and consistency model for multiprocessors system. TCC also permits unbounded size transactions by serializing all transaction commits when a transaction overflows its hardware buffer. Like UTM, VTM, the hardware change requested by TCC is not modest. Especially, TCC makes use of high bandwidth system interconnect to broadcast all of the transaction's write to the rest of the system. During the broadcast, the calling processor does not release the bus until the entire transaction completes. However, the broadcast is inherently not scalable. Therefore TCC may only be applicable to small scale multiprocessors system.

Transactional memory systems provide great potential to facilitate multithreading programming, however the above hardware proposals may not be practical in a near future. First, most of the design extends the cache coherence protocols, whose scalability with large number of processors are in doubt. The enhanced cache coherence protocols are augmented with more states, thus more complexity. This may further affects the scalability. Second, proposals like UTM, VTM, TTM, and TCC require far from modest hardware modifications. As a result, most of the hardware based transactional memory design can only be simulated with software simulator. It is not likely to present a real hardware implementation in the near future.

2.7.2.2 Software Transactional Memory

Software transactional memory (STM) adopts the transactional approach but use software based implementation. STM can be defined as a generic non-blocking transaction based synchronization construct that allows correct sequential objects to be translated automatically into correct concurrent objects [112]. Three representative STM designs will be reviewed.

Shavit and Touitou's STM

After Herlihy and Moss's proposal of a transactional memory system based on hardware extension [84], Shavit and Touitou proposed a software equivalent to support flexible transactional programming of synchronization operations [146]. In their software mechanism design, a transaction acquires the ownership of a concurrent object before making updates to it. The ownership is acquired atomically in a non-blocking fashion using atomic instructions like CAS and LL/SC [34]. In Shavit and Touitou's design, each shared memory word, which is treated as a concurrent object, has a distinct associated ownership record. The ownership record stores either a NULL value or a reference to its owner's transaction record data structure. The ownership is exclusive, i.e., at any time at most one transaction own a shared memory word. A transaction may need to acquire the ownership for multiple memory words to proceed. If a transaction fails to acquire one of the ownerships, it aborts and releases all its already acquired ownerships. If a transaction succeeds to acquire all desired ownerships, it proceed to make updates, change its state to COMMITTED, and release all the acquired ownerships, which requires a multi-word CAS operation. The major limitation of this STM design is that it can only be applied to static transactions, whose memory usage and accesses are known in advance. It requires multi-word CAS instruction, which is not support by any modern processors. Moreover, because each shared memory word has an associated ownership record, the memory requirement for this STM design is doubled.

Harris and Fraser's STM

Harris and Fraser proposed another word-based STM implementation, which allows dynamically non-conflicting execution to operate concurrently [77]. Their STM design makes use of hash table to store the ownership records. The STM system consists of three kinds of data structures: *application heap*, *ownership records (orec)* stored in hash table, and *transaction descriptors*. Application heap holds the shared memory that concurrent threads intend to access. Shared memory locations (word-based) are hashed into the orec table. An orec either contains a version number for the corresponding word or a reference to a transaction descriptor of the transaction, which holds the ownership of that record. A transaction descriptor consists of a status field, which indicates the transaction is either ACTIVE, COMMITTED, ABORTED or SLEEP, multiple transaction entries, one for each shared memory access. A transaction entry specifies the shared memory location accessed, the old and new values, and the old and new version numbers of those values. After a transaction starts, a read from or write to shared memory creates a transaction entry if one does not already exist in the transaction descriptor. The exclusive ownership of orecs referred in the transaction descriptor are acquired in the commit phase. The transaction uses atomic CAS operation on each orec to acquire the ownership. After all attempts to acquire the ownership success, the transaction changes its status field from ACTIVE to COMMITTED, makes updates to application heap, and then proceed to release all the ownership records.

During the execution of a transaction, if it finds that another transaction's descriptor is referenced by the orec that it tries to read or acquire. The transaction reacts to the conflict according to the status of the conflicting transaction. If the conflicting transaction is ACTIVE, the current transaction aborts it. This may incur live lock, therefore this STM design is *obstruction-free*. When the status of the conflicting transaction is either ABORTED or COMMITTED, if current transaction is reading, the orec version number is obtained from the descriptor of the conflicting transaction, and the contents of the memory location to be read is obtained either from the conflicting transaction's descriptor or the application heap; If current transaction is acquiring, a stealing mechanism is used to *steal* the ownership and merge the transaction entries from the conflicting transaction's descriptor into its own one. To ensure the consistency of the updates, a reference count is introduced to each orec to record number of transactions are in the process to updates to the memory location it manages. In order to atomically update both reference count and version number of anxs orec, a two-word wide CAS operation is required.

The limitations of this STM design are: (1) its contention resolution policy is "aggressive"– grant permission immediately to abort the conflicting transaction, which intends to incur live-locks when contention is high; (2) its stealing mechanism may cause long merge chains of transaction entries in a transaction descriptor; (3) it makes use of two-word wide CAS operation, which is not widely available off-the-shelf microprocessor.

DSTM

Dynamic software transactional memory (DSTM) proposed by Herlihy et. al. [82] is an object-based obstruction-free STM design, which supports transactions accessing dynamic-sized data structures. In this Java-based design, a transactional memory object (TM object) is a container for a regular Java object. A transaction must access a data object via the TM object. As shown in Figure 2.9, each TM object has a single reference field *start* that points to a *Locator* object. The locator object consists of three fields as depicted in Figure 2.9: the *transaction* points to the most recent transaction that tries to

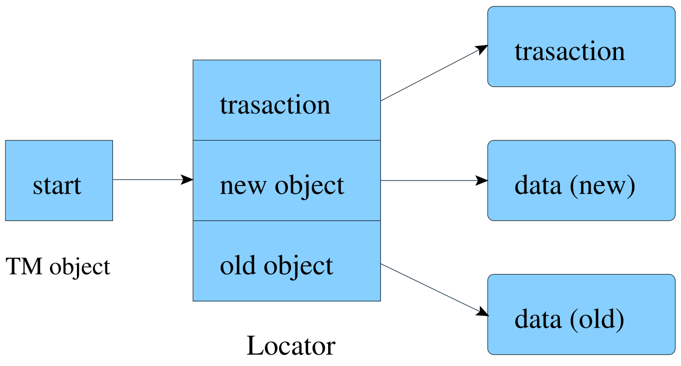


Figure 2.9: DSTM Transactional object structure

modify the TM object; the *new object* and *old object* point to the new and old object version. A transaction has three states: ACTIVE, ABORTED, or COMMITTED. The version of a TM object is determined by the status of the transaction referenced by the *Locator* object: if the status is ACTIVE, the *old object* points to the current version, and the *new object* points to transaction's tentative working-on version; if it is ABORTED, the *old object* points to the current version, and the *new object* is meaningless; if it is COMMITTED, the *new object* points to the current version, the *old object* is meaningless.

When a transaction tries to open a TM object, it creates a new copy of the *Locator* object, and makes itself referenced by the *transaction* field. The new locator's contents are determined by the state of the transaction pointed to by the old locator. If the old locator points to an ABORTED transaction, the new locator's *old object* field points to the old version object referenced by the old locator, and the new locator's *new object* field points to a copy of the old version object. If the old locator points to a COMMITTED transaction, the new locator's *old object* points to the new version object referenced by the old locator, and the *new object* field points to a copy of this new version. In case that the state of the transaction referenced by the old locator is ACTIVE, it means a conflict. Herlihy designs a *Contention Manager* protocol to decides which transaction should be aborted. Different polices can be implemented within the *Contention Manager* without changing the interface. After the new locator is setup, the transaction tries to replace the old locator with the new locator using an atomic single word-wide CAS operation. If the CAS successes, the current transaction obtains the ownership of the TM object and visible to all other threads. Otherwise, the current transaction must retry to open the TM object.

In DSTM, a transaction acquires the ownership of a transaction object when opening it. This eager acquire design decisions may lead to unnecessary abortion of transactions [112]. DSTM introduces indirection overhead by placing the *Locator* object within the TM object, and the *Locator* points to the transaction descriptor and concurrent data object. Since the each *Locator* always keeps two copies of the same data object (new and old version), the memory usage of DSTM is not efficient.

Other STMs

Besides the three representative STM systems that we just reviewed, there are also other proposals for software transactional memory and similar constructs appeared in the literature [58, 3, 15, 92, 125, 111, 162, 145]. Marathe and Scott [112] presents a qualitative survey of modern software transaction systems.

2.8 Data-Level Fine-Grain Synchronization

The granularity of a synchronization mechanism is determined by the smallest unit of memory that it can operate on. Fine-grain synchronization allows synchronization at the level of the memory words. For mutual exclusion, it allows each memory word to be individually locked and unlocked. For point-to-point event, it allows the synchronized write and read to perform on a single memory word. Given it can be efficiently implemented, fine-grain synchronization can be effectively used to exploit high degree of parallelism of many applications. To achieve the efficiency, fine-grain synchronization mechanism is normally provided by hardware.

2.8.1 Full/Empty Bits

Hardware support for fine-grain synchronization has been explored in several architectures built or proposed before. HEP [147], Tera [11], MDP [47], Alewife [102, 4], M-Machine [98], Cray MTA-2 [1], the MT processor in Eldorado [57], and others use hardware bits (e.g., *full/empty bits*) as tags to support word-level fine-grain synchronization. Often by default the entire memory of the machine is tagged by associating additional access state bits with each word in memory. Fine-grain synchronization is achieved by accessing those word-level state bits in memory.

We take Tera [11] as an example to explain how word-level synchronization is supported with full/empty bits. Each memory location (64-bits) in Tera computer system is associated with four access state bits. One of the bits is called full/empty bit, which is used for controlling synchronization behavior of memory references. The full/empty can represents the state of the associated memory location as either full (available) or empty (unavailable). A producer-consumer style of synchronization can be easily realized with full/empty bit: a synchronized load waits for full and then sets empty as it reads; on the other hand, a synchronized store waits for empty and then sets full as it writes. Tera provides hardware retry mechanism, when a synchronization attempt fails. For example, if a synchronized load request detects a empty state. The hardware is responsible for retrying the request automatically. The hardware also maintains a retry counter. When the counter exceeds certain threshold, a trap occurs.

Experience of parallelizing scientific code with full/empty bits fine-grain synchronization mechanism on Tera [11] or its successor Cray MTA-2 [1] has been reported in many literatures [148, 32, 17]. Agarwal et. al have reported their experience on the MIT Alewife machine [4]. They evaluated the performance of scientific applications, such as SOR, and MICCG3D, parallelized using J-structure and L-structure supported by hardware full/empty bits [102, 164].

2.8.2 Register-Register Communication

The M-Machine [98] architecture does not only tags every memory location with full/empty bits, but also allows fast fine-grain synchronization between three on-chip processors through register-register communication.

The MAP chip of the M-Machine maintains a register scoreboard to determine when values in register are valid. Each entry in the register scoreboard is a full/empty bit, which represents the state of the corresponding register. While the state of a register is empty, any operation that attempts to use the register will stall until it is full. The MAP chip architecture allows remote register write from one processor to another processor. The full/empty bit in register scoreboard is used to control the produce-consumer synchronization behavior between two processors. Such register-register communication efficiently completes fine-grain synchronization with data transfer between processors in a single operation.

2.8.3 I-Structure and M-Structure

An I-structure is a data structure proposed to facilitate parallel computing [20] on dataflow model based systems. An I-structure element can be in one of three states: empty, full, and deferred. I-structure uses single assignment semantics – an I-structure element can only be written once, but it can be read many times. Producer-consumer type of fine-grain data synchronization is achieved by interacting with the state of an I-structure when accessing it. Unlike I-structure, which regards the redefinition of an element as an error, the M-structure is a fully mutable data structure such that an element can be redefined repeatedly [25].

2.9 Synchronization Optimization with Compiler

2.9.1 Synchronization Optimization for DOACROSS Loops

In order to exploit loop-level parallelism of DOACROSS loops, iterations of a loop can be executed in parallel with multiple threads. In such cases, the cross-iteration data dependencies need to be enforced with fine-grain data synchronization operations. However, the excessive use of data synchronization can introduce significant overhead for both execution time and memory usage. Various compiler optimization techniques have been developed to minimize the amount of fine-grain synchronization added for parallelized do-across loops [108, 124, 103, 37, 129, 15, 138] and others. Those techniques try to minimize/reduce the amount of fine-grain synchronization operations inserted, but still preserve the parallelism that can be extracted from the loop.

2.9.2 Compiler-Automated Lock Assignment

Recently, Sreedhar and Zhang et. al. presented a new framework for analysis and optimization of shared memory parallel programs [151]. Based on concurrency relation and atomicity semantics, they present a framework to perform the data flow analysis for shared memory parallel programs. By applying the results of concurrency analysis and pointer analysis, the compiler can automatically assign locks to critical sections [167]. While the programmer assumes a single global lock for all critical sections, which eases parallel programming, the compiler finds the minimum number of locks that can be assigned to critical sections in a parallel program without reducing its parallelism.

2.10 Synchronization on Cyclops-64

Several synchronization mechanisms have been implemented for current Cyclops-64 chip architecture design:

- Atomic in-memory instructions, such as *fetch-and-add*, and *swap* can be used to implement various widely accepted *spin-locks*, such as *test-and-set*, *ticket lock*, and linked-list based *MCS* [118]. In C64, in-memory atomic instructions only block the memory bank where they operate upon while the remaining banks continue servicing other memory requests.

- The C64 sleep/wakeup instructions can be used to efficiently implement post/wait type of synchronization.
- The C64 chip architecture also provides a 16-bit signal bus to which all thread units within a chip are connected, that provides a means to efficiently implement barriers.

It is worth noting that the *compare-and-swap* (CAS) [34], *linked-load*, and *store-conditional* instructions are not currently supported in the design of C64. However, for the purpose of comparison, we implemented the CAS instruction in the C64 simulator [49].

Chapter 3

PROBLEM FORMATION AND EXPERIMENTAL INFRASTRUCTURE

3.1 Synopsis

In this chapter, we first describe and define the problems that the research in this dissertation targets (Section 3.2). We then present the experimental infrastructure that is employed for the research (Section 3.3).

3.2 Problem Formation

The design of high-performance processor chips is rapidly moving towards large-scale multi-core architectures that integrate 10s (or beyond) of tightly-coupled processing cores on a single silicon die [46, 27]. For example, Intel just announced its research prototype many-core design with 80 cores in a single chip [91]. A recent technique report from University of California at Berkeley predicts that 1000 cores can be fit into a die when 30nm technology is available [21]. For such large-scale multi-core architectures that contain a large number of cores, one can exploit massive intra-chip parallelism by maintaining a large number of active threads using multithreading techniques. It has been long realized that synchronization is crucial for the correctness and performance of multithreaded parallel programs. In this dissertation we revisit several important research problems in synchronization techniques in the emerging multi-core era.

3.2.1 Performance Characteristics of Synchronization Mechanisms on Large-Scale Multi-Core Architectures

To understand the performance behavior of parallel programs on the approaching large-scale multi-core architectures, it is important to comprehend the performance characteristics of synchronization mechanisms on such architectures. Since little experience has been gained for multi-core chips with more than 100 cores, the performance measurement and evaluation of widely used synchronization mechanisms, such as spin-lock algorithms, lock-free concurrent data structures, can reveal insights regarding following aspects of software/hardware development of large-scale multi-core architectures:

- Provide application developers a better understanding of the behavior of various synchronization mechanisms on large-scale multi-core architectures. Accordingly, programmers can choose the synchronization mechanisms that are most appropriate for the computation patterns of the target applications.

- Give system software (e.g., OS, library, compiler, etc.) developers hints regarding possible synchronization related optimizations and/or language extensions specific to multi-core architectures. System software plays a critical role for the success of multi-core architectures, because it functions as the interface between hardware and high-level software. The emerging multi-core architecture also presents many new challenges for the development of system software. Synchronization is definitely one of them. It is important that 1) the synchronization in system software itself should be implemented correctly (e.g. thread safety, etc.) and efficiently by taking advantage of underlying hardware features; 2) the interface provided to application developers should contain a rich set of efficient synchronization primitives. Therefore, it is crucial to provide system software developers a sound understanding on the performance characteristics of synchronization mechanisms on large-scale multi-core architectures.

- Since the significance of synchronization for the success of a multi-core architecture, it is important to help computer architects to understand the pros and cons of the architecture design in the aspects of hardware support for synchronization.

To complement the absence of such a study under the large-scale multi-core arena, this dissertation presents thorough performance measurement and evaluation for a range of widely used synchronization mechanisms. To this end, we select a state-of-the-art large-scale multi-core architecture, IBM 160-core Cyclops-64 (C64) chip architecture, as the target platform to conduct such a study. In Section 4.1 we will study the performance characteristics of several most widely used spin-lock algorithms on C64. Based on the performance evaluation, we will propose customized algorithm/implementation of a chosen spin-lock algorithm by taking advantage of underlying hardware features of C64 large-scale multi-core chip architecture. In Section 4.2, we will investigate three common concurrent data structures by comparing the performance of lock-free based implementations to the lock-based ones. Based on the observations drawn from our experimentation, the operating system or runtime system developers, library developers, and application developers for C64-like multi-core architectures could choose the appropriate version of the concurrent data structure to implement, according to the design requirements.

3.2.2 Performance Characteristics of OpenMP Language Construct on Large-Scale Multi-Core Architectures

Given a large-scale multi-core chip that integrates a large number of tightly-coupled simple processor cores, the challenge is to use this massive intra-chip parallelism to obtain highly sustainable performance. To meet such challenge, it is important to leverage system software techniques to coordinate these on-chip processors towards a scalable solution. High-level programming execution model is essential for providing an architecture abstraction for application development. As an industry de facto standard for writing parallel programs on shared memory systems, OpenMP seems to be a reasonable candidate to conduct a prototype study on high level parallel programming models. OpenMP specification [131, 132] provides a collection of compiler directives, library functions and environment variables, suitable for incremental and portable development of parallel applications. Parallel application developers express parallelism, work sharing, and synchronization through the OpenMP language constructs.

In section 4.3, in order to help the application developer and system software designer to increase the understanding on the performance behavior of OpenMP programs on large-scale multi-core architecture, we will measure and evaluate the performance characteristics of major OpenMP language constructs for synchronization as well as other types of language constructs on the C64 large-scale multi-core architecture.

3.2.3 Fine-Grain Synchronization on Large-Scale Multi-Core Architectures

In order to fully utilize the massive intra-chip parallelism provided by large-scale multi-core chips, it is important to exploit the fine-grain parallelism inherent in the applications. It has been long realized that the granularity of parallelism that can be efficiently exploited in such processors is often restricted by the lack of effective architectural support for efficient fine-grain synchronization. Software-only solutions, with very limited architectural support, often leads to poor scalability, high synchronization overhead, and high storage cost. It is often difficult or even impossible to harness fine-grain parallelism at the compilation time.

As a result, on large-scale multi-core architectures, fine-grain synchronization with hardware support is essential to the effective exploitation of fine-grain parallelism of applications. On large-scale multi-core chips (with 16 to 100 cores and beyond), the on-chip storage (memory) available per processor core is far less (often 1-2 orders less) than traditional single core microprocessors. On the other hand, there are plenty of distributed resources (e.g. large number of thread and memory units, ample on-chip interconnection bandwidth, etc.) available to facilitate efficient fine-grain coordination between processing cores and memory. Therefore, the following new challenges are emerging with respect to fine-grain synchronization solutions in large-scale multi-core architectures. Such solution should:

- be scalable and can fully exploit the parallelism due to the distributed on-chip resources.
- be supported with limited on-chip resources.
- incur low synchronization overhead.
- be able to support a variety of synchronization functionalities with modest hardware cost.
- be able to smoothly and efficiently handle the cases where the precise synchronization point cannot be resolved statically at compile time.

There are several design choices that one can implement fine-grain synchronization in hardware. For instance, HEP [147], Tera [11], MDP [47], Sparcle [4], M-Machine [98], the MT processor in Eldorado [57], and others use hardware bits as tags (e.g., *full/empty bits*) to support word-level fine-grain synchronization. These designs tag the entire memory of the machine by associating additional access state bits with each word in memory. Dataflow model-based architectures that use the I-structure [20] and M-structure [25] like fine-grain synchronization also exploit similar designs. Given that on-chip memory is one of the most precious resources for many-core chips, one down side of such design choices is the overhead and the cost associated with tagging every word in the memory.

To address the problem of such high-cost synchronization mechanisms, we propose, *Synchronization State Buffer (SSB)*, a novel architecture extension to large-scale multi-core chip architectures in Chapter 5. In Chapter 6, we will use IBM Cyclops-64 chip architecture as a case study to illustrate the characteristics of SSB and verify the efficiency and effectiveness of SSB.

In next section, the experimentation infrastructure for all the performance studies in this dissertation will be presented.

3.3 Experimental Infrastructure

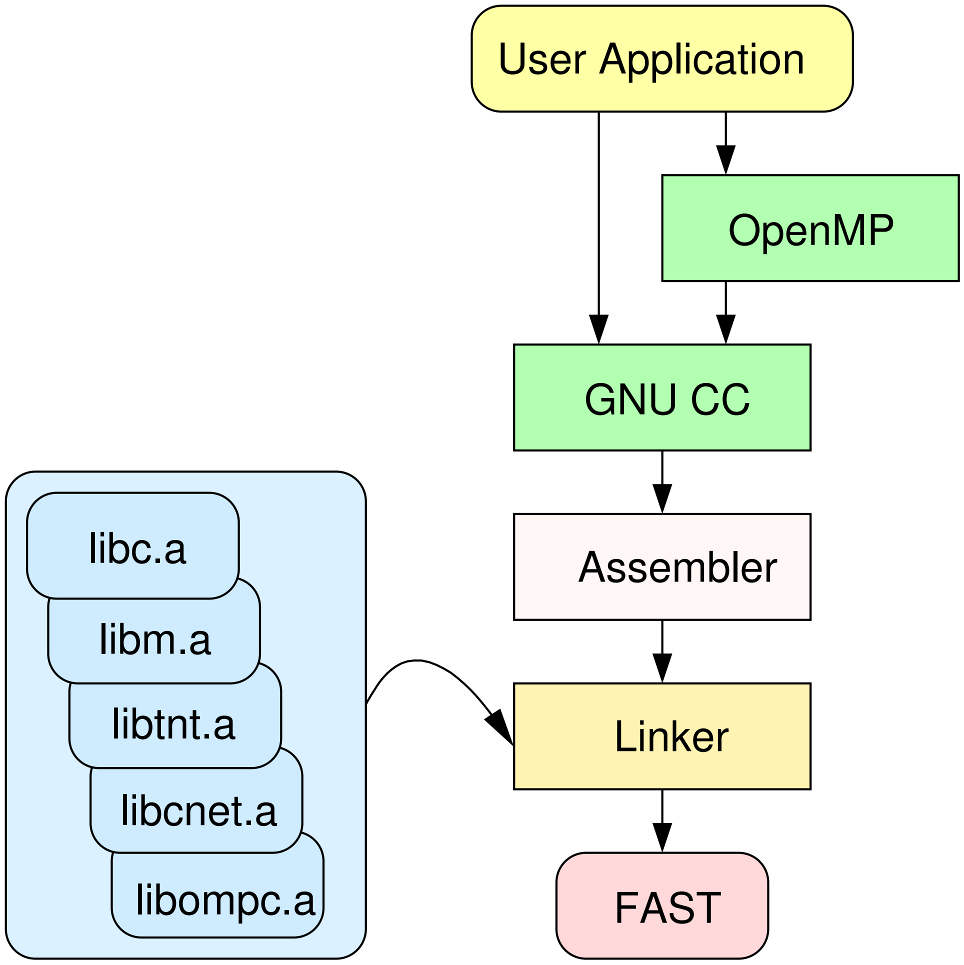
The IBM Cyclops-64 architecture [52, 53] is the target large-scale multi-core architecture to conduct experiments. The experimental framework include following parts:

- The C64 system software toolchain [51] based on the C64 TiNy Threads (TNT) virtual machine [50] (Section 3.3.1).
- The C64 FAST simulator [49] (Section 3.3.2).
- The Omni OpenMP Compiler [105, 153], which has been ported to C64 [48] (Section 3.3.3).

3.3.1 Cyclops-64 System Software Toolchain

Figure 3.1 shows the C64 system software toolchain [51], which is used for software and application development on the C64 system. The toolchain consists of following basic components:

- **Binary utilities (binutils):** assembler, linker, and objdump, etc. The binutils is ported from GNU binutils-2.11.2 [64].
- **GNU CC compilers:** C and Fortran compilers, which are ported from GCC-3.2.3/GCC-4.0.0 suite [63, 67]. Unlike system built on conventional off-the-shelf microprocessors, where virtual memory provides each process a continuous linear address space, C64 employs an explicitly addressable memory hierarchy without virtual memory management. Programmer should be aware of not only the size limitations of each memory segment, but also the different latencies and bandwidth



Courtesy: Juan del Cuvillo.

Figure 3.1: Cyclops-64 System Software Toolchain

for accessing different segments, as shown in Figure 2.7. To fully exploit such multi-layered memory hierarchy of C64, the compiler, assembler, and linker are enhanced to support segmented memory spaces that are not contiguous. In other words, multiple sections of code, initialized and uninitialized data can be allocated on different memory regions. To direct the allocation of sections, *pragmas* are provided for programmers to specify the memory segments where the user would like to place certain variables or procedures. For instance, frequently used data structures can be put in the scratchpad memory, which is close to the processor/thread unit. In general, application developers should be aware of the latency and bandwidth of different memory segments, so that in the end they make the best use out of the memory. The current toolchain with *pragma* support for segmented memory spaces is the first step towards this goal.

- **Standard C and math libraries:** the libraries are derived from those in newlib-1.10.0 [128]. Functions (libc/libm) are thread safe, i.e. multiple threads can call any of the functions at the same time. In addition, memory copy functions have been optimized by taking into account the memory hierarchy and C64 ISA support for multiple load and store instructions that make more efficient use of the memory bandwidth [87].

- **TNT microkernel/runtime system library:** provides the software and application developer with the functionality to write multithreaded programs: thread management, support for mutual exclusion, synchronization among threads, etc. In order to achieve high performance and scalability, the implementation of such functionality tries to match as close as possible the architecture underneath the microkernel/RTS [50].

- **CNET communication protocol and library:** The CNET communication library is used to manage the A-switch communication hardware [53] to provide user-level remote memory read/write functionality.

- **SHMEM:** The SHMEM [43, 134] shared memory access library, which is built on CNET, is developed to support high-level shared memory programming model across C64 nodes. SHMEM provides a shared global address space, data movement operations between locations in that address space, and synchronization primitives that greatly simplify programming for a multi-chip system such as C64. We extended the SHMEM API to support both SPMD and Non-SPMD programming model on C64 environment. For SPMD model, the “main” function of user code is invoked at all available thread units on all chips when the program starts. The SHMEM operations are responsible for the communication and synchronization among all threads. For the Non-SPMD model, the “main” is only invoked on one thread unit on each chip. In this model, the SHMEM operations are only responsible for inter-chip communication and synchronization.

Before the actual C64 chip is available, the development and research of system software and scientific and engineering applications are conducted on an execution-driven, binary-compatible simulator of a multi-chip multithreaded C64 system, which is called FAST. FAST accurately reproduces the functional behavior and count of hardware components such as chips, thread units, on-chip and off-chip memories, and the 3D-mesh network [49]. More detail of FAST simulator will be introduced in Section 3.3.2.

3.3.1.1 TiNy Threads

The cornerstone of the C64 system software toolchain is a thread virtual machine (TVM), called TNT (or TiNy-Threads) [50]. The TNT TVM includes the TNT non-preemptive thread model, memory model, and synchronization model. Based on TNT TVM, a microkernel and the TiNy Threads^{TM}(TNT) runtime system are customized for the unique features of the C64 architecture [50]. The TNT library provides user and library developers an efficient Pthreads-like API for thread level parallel programming purpose. We will briefly review the thread model, memory model, and synchronization model of TNT TVM in this subsection.

TNT has been designed and developed to support a multithreaded programming model for a large-scale multithreaded multi-core architecture such as Cyclops-64. One of the most remarkable feature of the C64-like large-scale multi-core architecture is its high computation to memory ratio. For example, a C64 chip consists of 160 thread units and approximately 4.7MB on-chip SRAM. Although the total amount of on-chip memory is comparable to the on-chip data cache of common off-the-shelf processors, the amount of memory per thread unit is small (about 30KB per thread unit). Moreover, C64 does not employ any data cache but an explicitly addressable memory hierarchy. Given these special features of C64, an conventional OS would put a considerable overhead on top of it. Instead, TNT is designed and implemented directly on top of the hardware architecture as a micro-kernel/run-time system library that takes advantage of C64 hardware features while providing an interface that shields application programmers and system software developers from the complexities of the architecture wherever possible.

3.3.1.1.1 TNT Thread model and API

Thread execution on C64 is *non-preemptive*. That means once a thread starts running on a thread unit of C64 there is no mechanism available to interrupt the thread unless an exception occurs. However, the C64 instruction set architecture design includes efficient support for thread level execution. For instance, it provides a sleep instruction, such that a thread can stop executing instructions for a number of cycles or indefinitely. When asleep, a thread can be woken up by another thread through a hardware interrupt. Such an interrupt is generated through a store instruction to a thread-specific memory-mapped port.

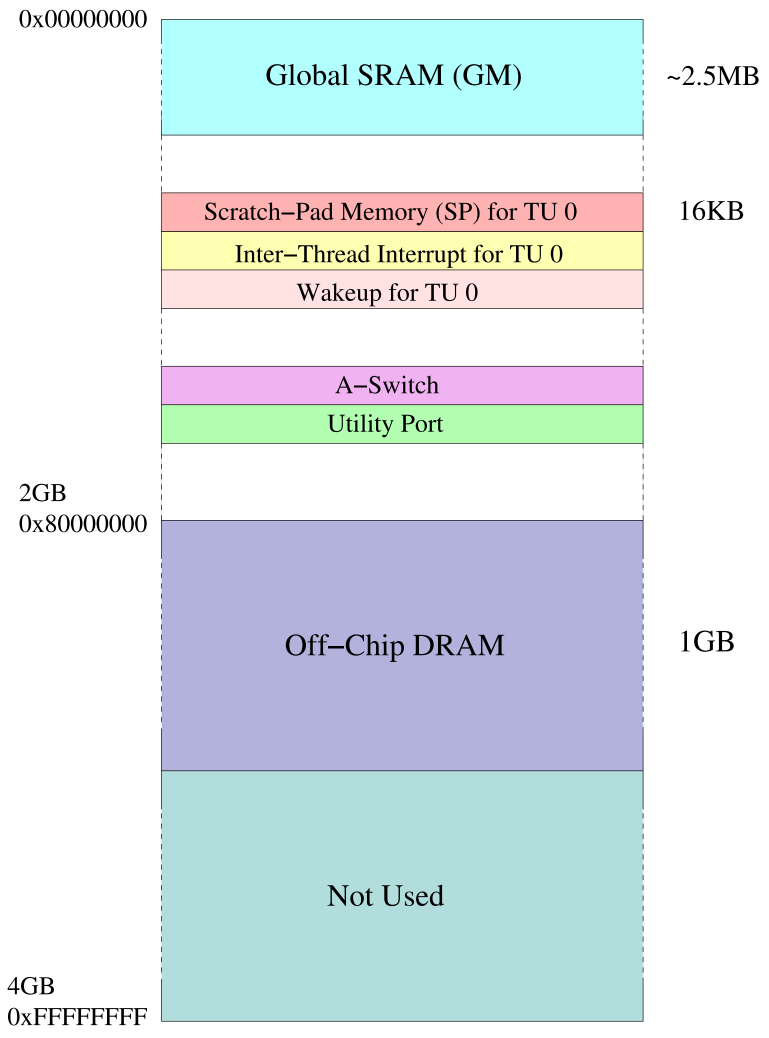
In the TNT thread model, thread execution is non-preemptive and software threads map directly to hardware thread units. In other words, after a software thread is assigned to a hardware thread unit, it will run on that hardware thread unit until completion. Furthermore, a sleeping thread will not be swapped out so that idle hardware resources can be assigned to another software thread. As in other thread models, a waiting thread (waiting on an external event/synchronization) goes to sleep; such a thread is woken up by another thread through the hardware signal.

An API of TNT thread model inspired by that of the popular Pthreads model, is provided to ease the application and system software development. With TNT, user can choose either SPMD or Non-SPMD execution model. With the SPMD model, the “main” function of the user code is launched on all available thread units when the execution starts. User does not need to explicitly create and join threads. Instead, the RTS takes the responsibility for starting threads on the thread units. With the Non-SPMD model, like the Pthreads model, user is responsible for explicitly creating, joining, terminating threads by inserting appropriate function calls to the TNT runtime library. In both models, inter-thread synchronizations, such as barrier, mutex, etc., are managed by user directly.

3.3.1.1.2 TNT Memory model

On C64 there is no hardware virtual memory manager, which means the three-level memory hierarchy of the C64 chip is exposed to the programmer directly. The C64 hardware chip supports direct memory access from all thread units/processors to the shared address space covering the on-chip memory (interleaved and scratchpad sections) and the off-chip DRAM banks associated with the chip. That is, all threads see a single non-uniform shared address space, which is shown in Figure 3.2. The On-chip SRAM memory space is limited in the current technology to around 5MB, so it should be viewed and used as temporary storage during computation. There is no hardware data cache used in the C64 design. Off-chip DRAM should be considered as the main memory.

In addition, it has been proven that the C64 architecture behaves as sequentially consistent for the interleaved and off-chip memories [168]. However, hardware cannot



Courtesy: Ziang Hu.

Figure 3.2: Memory Image of a Cyclops-64 Chip

guarantee a “Lamport order” of the accesses to the scratchpad memory space, hence no sequential consistency can be assumed.

TNT is a memory-aware runtime library that takes advantage of C64 explicit memory hierarchy by placing frequently used data in scratchpad memory that is closer to the processor/thread unit. Upon initialization, each software thread is given control over a well determined region of the scratchpad memory, which is allocated to every physical thread unit at boot time. Such a section of memory holds the thread descriptor, a fixed-size structure (192 bytes) that holds all the information required to properly handle the thread, including its stack pointer, and a small amount of thread local data that is directly managed by the user.

3.3.1.1.3 TNT Synchronization model

C64 architecture has a rich set of hardware supported in-memory atomic instructions. Atomic instructions in the C64 only block the memory bank where they operate upon while other banks can still continue servicing memory requests. In addition, threads within a C64 chip are connected to a 16-bit signal bus that provides a means for very fast communication of a small amount of information, which can be used to efficiently implement barriers. Thread units also have an inter-thread interrupt device, which is mapped to memory. This device allows one thread to interrupt another (or itself).

TNT provides a unique spin lock algorithm for shared memory synchronization designed to make best use of the C64 in-memory atomic instructions and thread sleep/wake-up mechanisms (see Section 2.3.4). For collective synchronization, TNT library provides direct access to the signal bus interface register. Besides significant improvements in the execution time of barrier operations, the signal bus reduces memory traffic and power consumption, as spinning waiting for a signal bus line to drop does not interfere with other thread units or generate excessive heat. A third type of synchronization in TNT is introduced to express precedence relations between operations from two different threads. In the first version of the C64 TVM, we provide a coarse-grain signal-wait type of synchronization based on the inter-thread interrupt that should be placed between a pair of specific program points within the two threads. In Chapter 5, we will extend the C64 synchronization model with a fine-grain synchronization mechanism, called *synchronization state buffer (SSB)*.

3.3.2 FAST Simulator

Due to the increasing complexity of computer systems, architecture researchers are now significantly relying on simulators to analyze and understand the impact of various architectural parameters and components as well as study the application performance and get detailed statistics. Simulation frameworks for microarchitecture research and design exploration, such as SimpleScalar [31, 22], Microlib [135], Liberty [157], RSIM [88] and Turantdot [127], concentrate on accurately modeling the architecture design and normally they are cycle accurate. However, given the complexity of large-scale multi-core C64 chip architecture, cycle accurate simulation would be too slow for a system consisting one or more fully-populated C64 chips. As a practical approach suggested by the C64 architect, a functional simulator, FAST, is designed and built.

FAST (Functionally Accurate Simulator Toolset) [49] is an instruction-set level simulator for the IBM C64 architecture. FAST is designed for the following goals (1) architecture design verification; (1) architecture related research; (3) system software development and testing; and (4) application software development and testing.

FAST is a functionally-accurate, execution-driven, binary-compatible, and full-system simulator of a multi-chip multithreaded C64 system. FAST has been developed following a modular approach, such that additional features could be easily incorporated into the existing design. It accurately reproduces the functional behavior of hardware components such thread units, on-chip and off-chip memory banks, and the 3D-mesh network. To help the architecture team with the verification of the C64 chip design, the simulator 1) executes instructions, and architecture exceptions; 2) reproduces the C64

Table 3.1: FAST Simulation parameters

Component	# of units	Params./unit
Threads	160	single in-order issue, 500MHz
FPUs	80	floating point/MAC, divide/square root
I-cache	16	32KB
SRAM (on-chip)	160	32KB
DRAM (off-chip)	4	256MB
Crossbar	1	96 ports, 4GB/s port
A-switch	1	6 ports, 4GB/s port

memory map; 3) produces histograms of the instruction mix; and 4) generates detailed traces of all instructions executed. In addition, FAST models all details in the memory hierarchy, including contention on memory banks and in the crossbar network. FAST also models instruction cache, supports intra-chip communication through the A-switch device, and incorporates debugging facilities. Table 3.1 shows the major simulation parameters of FAST.

For the propose of this dissertation, it is important to accurately model the segmented memory space, and the memory and interconnect contention. FAST simulator accurately models the C64 three-level memory hierarchy, and accounts for the contention in the crossbar network and in the memory system.

In the C64 chip architecture, each thread unit has an associated SRAM bank. Each memory bank can be partitioned (configured) into two sections: one called “global” (or “interleaved”) section, the other “local” (or “scratchpad”) section. All such global sections together form the (on-chip) global memory in an interleaved fashion that is free of holes and uniformly addressable from all thread units. Although scratchpad memory, global memory and off-chip DRAM memory are addressable from any thread within the chip, the access latency to different segments of memory is not uniform. Furthermore,

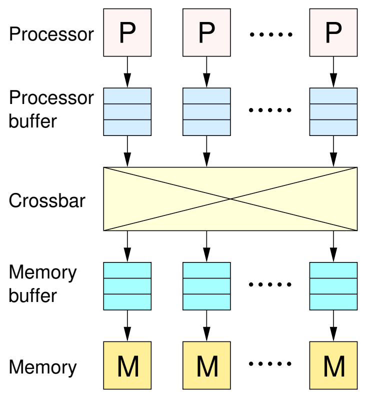


Figure 3.3: Interconnection to the On-Chip Crossbar

there is no hardware support for virtual memory in the C64 architecture, hence this memory hierarchy is directly exposed to programmers.

The FAST simulator accurately models the C64 memory map by implementing the non-uniform shared address space. It also includes the address upper limit special purpose registers (AULx) that define the highest existing location in scratchpad memory, global memory and DRAM memory, respectively. Nonetheless, all memory-specific parameters such as the number of banks, size of each bank, latency, and bandwidth are easily configurable. In addition, it considers three protection boundary special purpose registers (PBx). These registers define regions in scratchpad, interleaved and DRAM memory that can only be written in supervisor state, which effectively provide a basic mechanism to protect the kernel against malign user code.

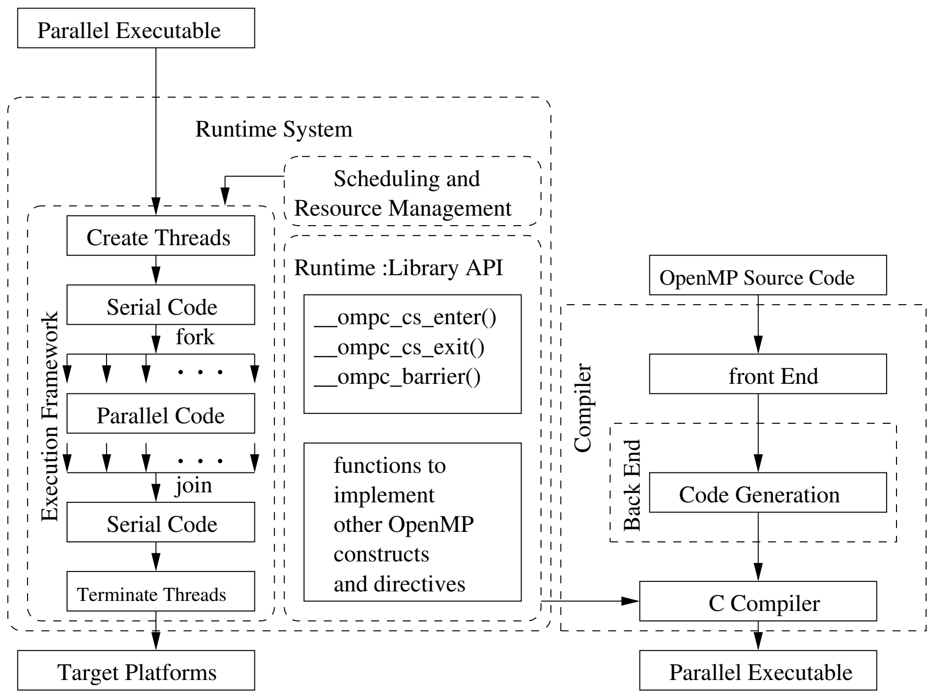
Figure 3.3 illustrates the data path between processors and memory banks on a C64 chip. Except the access to a thread's local SPM, every memory instruction executed on a processor results in a network packet delivered by the crossbar network to the appropriate memory bank (global SRAM or off-chip DRAM). For load operations, the memory replies with another packet containing the data retrieved from memory.

FAST models the following sources of contention: (1) Packets issued by threads on the same processor are queued on a 7-slot FIFO (processor buffer) until they are retrieved by the crossbar. If a thread issues a memory operation when the FIFO is full, the pipeline will stall until space is available; (2) The crossbar retrieves packets from the input ports and delivers packets to the output ports, one per cycle. If at the same cycle, two packets are to be delivered to the same output port, the crossbar blocks one of them arbitrarily; (3) Between the crossbar and each memory bank there is another 7-slot FIFO (memory buffer) where packets are held until processed by the memory. Whenever this buffer becomes full, the crossbar stops delivering packets to this destination. At the same time, it stops retrieving packet from any input that tries to send packets to the blocked output port; (4) Memory latencies are also taken into account. SRAM memory banks can perform a load or store operation every cycle, i.e., 4GB/s per bank. Whereas DRAM memory can sustain a much lower bandwidth. DRAM memory consists of four banks and each bank is subdivided into four subbanks. Subbanks can service requests simultaneously, one every 32 cycles. While a memory subbank is in service, an incoming request is held pending in the memory buffer. Therefore, the DRAM bandwidth is 2GB/s for single loads and stores. For multiple transfers, using load multiple (LDM) and store multiple (STM) instructions, the DRAM bandwidth is 16GB/s instead.

For details about other modules in FAST simulator, please refer to [49]. In the work of this dissertation, we will extend FAST simulator to model new hardware features proposed.

3.3.3 Omni OpenMP Compiler

OpenMP [35] is a widely accepted parallel programming API for shared memory machine. OpenMP allows programmer to explicitly construct multi-threaded parallel programs through compiler pragma. OpenMP employs the fork-join model for parallel execution. An OpenMP program begins as a single thread - master thread, which executes sequentially until it encounters the first parallel region. At the entry of parallel region, the



Courtesy: This figure was first created by Yuan Zhang, and then revised by the author of this dissertation.

Figure 3.4: Omni OpenMP Compiler Structure

master thread creates a team of parallel threads. The statement block enclosed by the parallel region construct is then executed in parallel by those threads. When all the threads in the team complete, they synchronize and terminate, leaving only the master thread.

The Omni OpenMP compiler [105, 153] is a source-to-source compiler, which translates the OpenMP (C/C++ or Fortran) program into C program with Omni runtime library function calls. A general C compiler (for example, GCC) then compile it into parallel executable. As shown in Figure 3.4, The Omni runtime systems is composed of three parts. The runtime library API provides functions for implementing OpenMP constructs and directives. The execution framework part executes the parallel executable generated by the compiler in a fork-join model in the target platforms, with the help of scheduling and resource management part.

The C64 OpenMP compiler and runtime environment is ported from Omni-1.6 [105]. For parallel execution, Omni relies on the POSIX thread library, which makes porting to other platforms easy. On C64 there is not a POSIX thread library. However, for the purpose of this work we extended the C64 native microkernel and multi-threaded runtime (TNT) with macros that provide the POSIX thread API. Hence, the OpenMP runtime library is built on top of TNT. Therefore, the OpenMP runtime library obtained from this straightforward porting is already efficient in the sense that it brings the runtime library closer to the underlying hardware, making use of the efficient thread management techniques provided by TNT, for instance.

We then further investigated and optimized the Omni OpenMP runtime library by exploring C64 hardware features [48]. Our approach is comprised of three steps:

1. A memory aware runtime library that takes advantage of the C64 explicit memory hierarchy by placing frequently used data structures in scratchpad memories that are closer to the processor/thread units. We privatize the descriptors defined within the library to handle both the processors (hardware thread units) and threads (OpenMP threads). The benefit of this relocation is simple: faster access to the descriptor. Load from local, global and off-chip memory takes 2, 20 and 36 cycles, respectively. Besides faster access to its contents, the new location of the descriptor also provides faster self-identification and less frequent access to the master thread descriptor.

2. A unique spin lock algorithm designed to make best use of the C64 architecture supported in-memory atomic instructions and thread sleep/wake-up mechanisms. For more detail, please see Section 4.1.4.

3. The signal bus on C64-chip provides a means for very fast communication of a small amount of information among thread units within a chip. We use the signal-bus to implement the barrier synchronization. The original Omni barrier function

implements a 1-read/n-write busy-wait algorithm [118]. Obviously, the hardware mechanism for barrier synchronization available on C64 should outperform this software implementation. Hence, the default barrier function has been replaced with calls to the TNT library that access the signal bus interface register. Besides significant improvements in execution time, the signal bus reduces memory traffic and power consumption, as spinning waiting for a signal bus line to drop does not interfere with other thread units or generate excessive heat.

The above optimizations of the Omni OpenMP runtime library result in a significant performance improvement (e.g. overhead reduction as high as up to 2 orders of magnitude, and at least 80% for OpenMP language constructs) [48]. Due to the drastic reduction of overheads in the OpenMP runtime library, the optimized Omni OpenMP runtime library paves the way for a productive use of OpenMP as a high-level parallel programming model for the Cyclops-64 platform.

In the work of this dissertation, to implement multithreaded programs on C64, we will either use the low-level API of TNT library directly or the high-level OpenMP programming model.

Chapter 4

EVALUATION OF SYNCHRONIZATION MECHANISMS ON CYCLOPS-64

To understand the behavior and performance of parallel programs on the approaching large-scale multi-core architectures, it is important to understand the performance characteristics of widely used synchronization mechanisms on this new generation of microarchitectures. Since little experience has been gained for multi-core chips with more than 100 cores, the performance measurement and evaluation of synchronization mechanisms, such as spin-lock algorithms, and lock-free concurrent data structures, can provide insight regarding following aspects of software/hardware development of multi-core architectures:

- Provide application developers a better understanding of the behavior of various synchronization mechanisms on large-scale multi-core architectures. Accordingly, programmers can choose the synchronization mechanisms that are most appropriate for the computation patterns of the target applications.

- Give system software, library, and compiler developers hints on possible synchronization related optimizations and language extensions specific to multi-core architectures.

- Help computer architects to understand the pros and cons of the architecture design on the hardware support for synchronization.

To complement the absence of such a study, this chapter presents a thorough performance measurement and evaluation of a range of widely used synchronization mechanisms on the IBM 160-core Cyclops-64 (C64) chip architecture. In Section 4.1, we implement and compare several most widely used spin-lock algorithm on C64. Based on the performance evaluation using microbenchmarks, we present a customized version of MCS spin-lock algorithm [118] on C64, which takes advantage of underlying hardware features of C64-like multi-core chips. In Section 4.2, we investigate the performance of three most widely studied concurrent data structures on C64. Section 4.3 presents our performance evaluation of language constructs in OpenMP on C64. Section 4.4 concludes this chapter.

4.1 Evaluation of Spin Lock Algorithms on C64

Spin lock is one of the most widely used synchronization primitives in parallel programming. Spin lock is usually used to achieve mutual exclusions, which resolve conflicting accesses to shared resources by concurrent processes or threads. In this section, we study the performance characteristics of spin-lock algorithms on a large-scale multi-core architecture – the IBM Cyclops-64 (C64) chip architecture.

4.1.1 Spin Lock Algorithms

For the purpose of performance evaluation, we implemented following six most widely used spin lock algorithms on C64:

Test-and-set (TS) With TS lock, a processor repeatedly checks the lock to see if it is available and, if available, marks it as unavailable. A hardware *test-and-set* instruction is used to perform the check-and-mark-if-available actions atomically. In our implementation, the lock object is allocated in the on-chip global memory. We do not employ the well known *test-and-test-and-set* approach [143], because there is no data cache on C64.

Test-and-set with exponential backoff (TS-exp) The same as TS lock, but instead of retrying immediately after failing to acquire the lock, an exponential increasing backoff is performed before the next attempt.

Ticket Before acquiring the lock, a processor increments a global counter to determine its position in a waiting list. All processors spin on a second global counter, which will be incremented when the lock is released. Both counters are allocated in on-chip global memory. Only the increment-and-get-ticket operation on the first global counter requires the use of a *fetch-and-inc* hardware atomic instruction, the spin uses a normal load instruction. In our implementation, an exponential increasing backoff is also used between two spins.

TA TA is an array-based lock algorithm proposed by T. Anderson [16]. When a thread acquires the lock, a slot in the array is assigned incrementally. The thread then spins on its slot. The lock owner releases the lock by setting the next slot as available.

GT GT is another array-based lock algorithm invented by G. Graunke and S. Thakkar [68]. With the TA algorithm, a slot is assigned to a thread at runtime. For GT, every thread has its own fixed slot in the array. When a thread acquires the lock, it fetches the address of its predecessor's slot from a tail structure, which keeps track of the tail of the lock waiting queue. The thread spins on its predecessor's slot. The lock owner release the lock by setting its own slot.

MCS An MCS lock [118] uses a distributed linked list to maintain the queue of waiting threads. Each thread spins on a separate node of the linked list. In our implementation, the node where a thread spins on is allocated in its own scratchpad memory. Therefore, there is no memory traffic generated to the crossbar network when a thread spins locally.

4.1.2 Microbenchmarks

To evaluate the efficiency of different spin lock algorithms, we use two microbenchmarks proposed in [104]: *lock-delay*, and *lock-null*. In both benchmarks, each thread repeatedly performs 10,000 pairs of lock acquires and releases. The lock-delay microbenchmark uses fixed delays both inside and outside the critical section. The delay (D_i) inside the critical section is large enough (we use $D_i = 3 \times D_0$, where D_o is the delay outside the critical section) such that the last thread that released the lock is already waiting to acquire the lock before the lock is released. Therefore, the amount of lock contention is guaranteed to be $P - 1$, when using P threads. For this microbenchmark, the overhead of a lock can be computed as follows:

$$\text{Overhead} = \begin{cases} \frac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i - D_o \;, P = 1 \\ \frac{\text{Execution Time}}{\text{No. Lock Acquires}} - D_i \qquad\;\;\, , P > 1 \end{cases}$$

$$\text{No. Lock Acquires} = \text{Iterations Per Thread} \times \text{No. Threads}$$

The second microbenchmark, lock-null, does not use any delay at all. Each thread continuously acquires and releases the lock 1,000 times. This is also the benchmark used in other studies [118]. On a conventional SMP architecture, unless a fair lock algorithm is used, a processor that releases a lock is favored to re-acquire the lock again, because of the difference between network latency and local cache access time. This is not the case for C64, since there is no data cache on C64. All threads have equal chance to acquire the lock located in global on-chip SRAM. For this microbenchmark, the overhead of a lock can be computed as follows:

$$\text{Overhead} = \frac{\text{Execution Time}}{\text{Iterations Per Thread} \times \text{No. Threads}}$$

4.1.3 Evaluation

We measured the overhead and contention of the six spin lock algorithms (see Section 4.1.1) using the lock-delay, and lock-null microbenchmarks. The overhead is calculated using the equations provided in Section 4.1.2. The amount of contention is reported by the simulator. When two or more threads compete for the same resource at the same cycle, the simulator increments the contention counter by one. In Figure 4.3 and Figure 4.4, we normalize the number of contentions by the product of number of threads and number of iterations. Therefore, the data reported is the average number of contention that one thread encounters for a pair of lock acquisition and release. Low contention is important because it does not affect the overhead of the lock acquire/release only, but the normal execution of the user program as well.

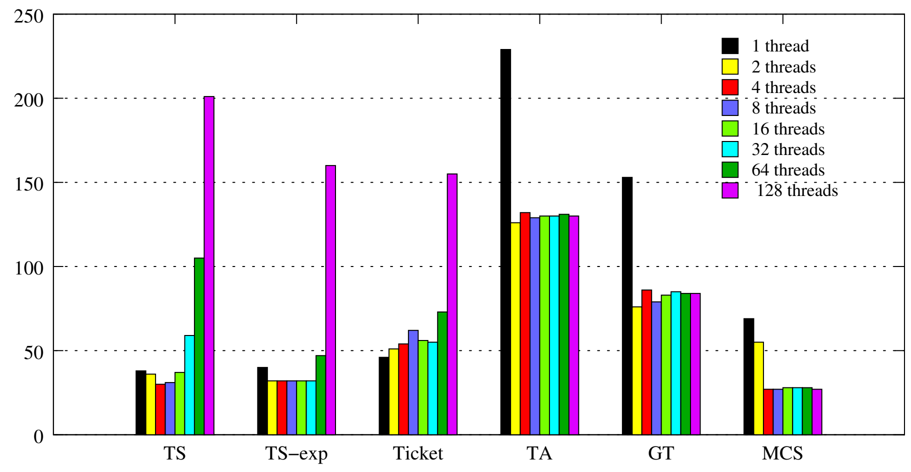


Figure 4.1: Overhead of Spin Lock Algorithms [clock cycles] with lock-null Microbenchmark

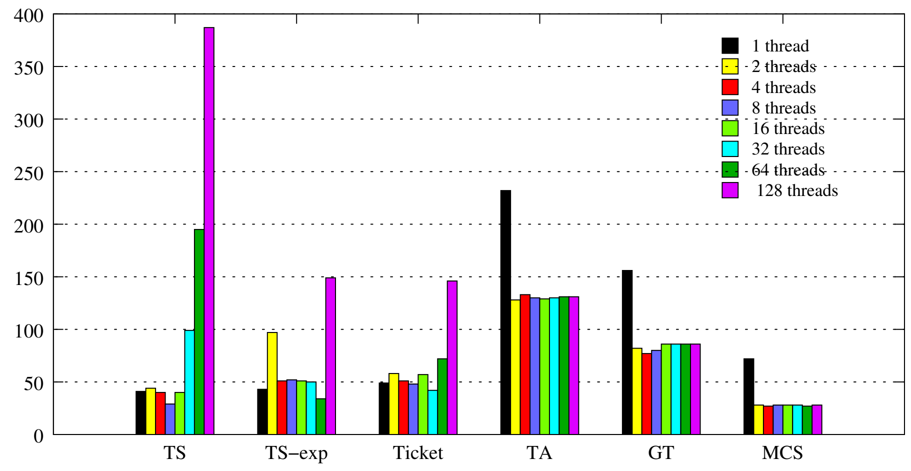


Figure 4.2: Overhead of Spin Lock Algorithms [clock cycles] with lock-delay Microbenchmark

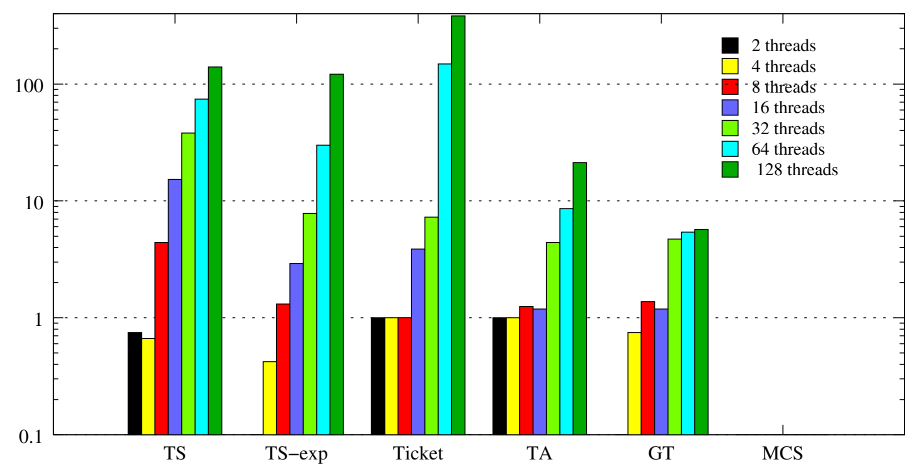


Figure 4.3: Normalized Contention of Spin Lock Algorithms with lock-null Microbenchmark

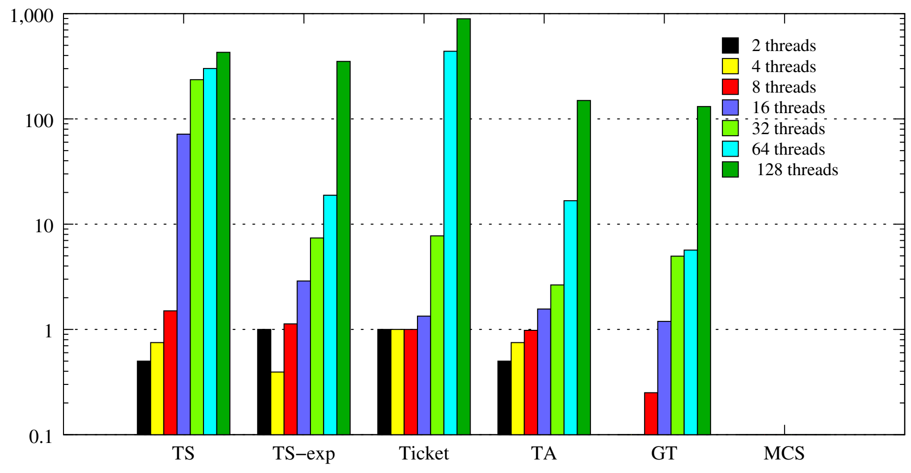


Figure 4.4: Normalized Contention of Spin Lock Algorithms with lock-delay Microbenchmark

From Figure 4.1, 4.2, 4.3 and and 4.4, we can conclude the following observations:

- When there is no lock contention (i.e. running with 1 thread), the TS algorithm has the smallest overhead. As a result, TS is suitable for low contention scenario.
- When the number of threads is larger than two, the MCS algorithm always incurs the lowest overhead. Therefore, MCS is suitable for implementing high-contention lock.
- TS, TS-exp, Ticket algorithms show very high level of contention when number of threads are large. The TA and GT algorithm also show certain level of contention when number of threads reaches 32.
- Since only spin locally, the MCS algorithms does not incur any contention no matter how large the number of threads is.
- Ticket algorithm has lower overhead but higher contention than TS-exp lock. As we mentioned earlier, TS-exp always spins with the *test-and-set* instruction, which holds a memory module for three cycles on C64. However, Ticket uses *fetch-and-increment* once to get its ticket and increment the counter. Then it spins using normal load instructions, which are served by memory in one cycle. As a result, with contention at the memory module, TS-exp experiences a longer delay than Ticket.
- The array-based TA and GT algorithms can not pre-determine the slot of the array that a thread will spin on before runtime. For system without data cache, like C64, these two algorithm can not guarantee local spin. The spin actually happens in global SRAM. On the contrary, for the linked-list based MCS, each thread can place its node structure into its own scratchpad memory, thus ensures local spin.

- Although local spin can not be achieved, the array-based TA and GT algorithms can ensure threads to spin on different slots in the array. Thus the contention is lower than the centralized TS, TS-exp, Ticket lock. The TA and GT algorithm requires that each slot of the array lies in a different memory bank. However, on C64, the memory addresses are interleaved to different memory banks with a 64-byte boundary and the maximum size of data that can be operated by an instruction is 8 bytes – a doubleword. As a result, 8 consecutive slots (each is 8 bytes) in the array are located in the same memory bank. When several threads spin on the slots located in the same memory banks, contention still happens. This explains the reason why TA and GT also experience high contention when number of threads are large than 32. It is possible to reduce the amount of contention by allocating 64 bytes for each element in the array such that each of them lies in a different memory bank. The algorithm can only operate on the first 8 bytes of each slot. Therefore, this approach, which causes prohibitive space needs for implementing the TA and GT locks, is not practical.

- TS, TS-exp, and Ticket spins on a centralized lock, thus the memory requirement for implementing a lock is small – a doubleword for TS, TS-exp, two doublewords for Ticket. The array-based TA and GT, and the linked list based MCS consumes much more memory, which is proportional to the number of threads.

In summary, on C64-like large-scale multi-core architectures, the simple *test_and_set* based lock achieves best performance when there is no contention; the MCS lock algorithm is suitable for high-contention scenario.

4.1.4 Customizing MCS for C64

Previous subsection shows that the MCS spin lock algorithm is the best one when number of threads are greater than two. The advantage of MCS can be attributed to the feature that local spinning is ensured when a thread is waiting for a lock.

```
typedef void * qnode;
typedef qnode tLock;

void init_lock(tLock **lock){ *lock = NULL; }

void acquire_lock(tLock **lock, qnode *I)
{
  qnode *predecessor;
  *I = NULL;
  predecessor = fetch_and_store(lock, I);
  if(predecessor != NULL){
    *predecessor = I;
    suspend();
  }
}

void release_lock(tLock **lock, qnode *I)
{
  qnode *old_tail,*usurper,*ptr;
  if(*I == NULL){
    ptr = NULL;
    old_tail = fetch_and_store(lock, ptr);
    if(old_tail == I) return;
    usurper = fetch_and_store(lock, old_tail);
    while(*I == NULL);
    if(usurper != NULL) *usurper = *I;
    else
      wakeup(*I);
  }
  else
      wakeup(*I);
}
```

Figure 4.5: Customized MCS Spin Lock Algorithm on C64

To further improve the performance and power-awareness of the algorithm, we modify the original MCS algorithm using the C64 ISA-level sleep/wake-up support as introduced in Section 2.3.4. Instead of spinning, a thread that is waiting on a lock goes to sleep after it adds its node (allocated in its scratchpad memory) to the linked list. When a lock owner releases the lock, it wakes up its successor by sending a wake-up signal, which has the same cost of as a store instruction. We call this customized MCS algorithm as MCS with sleep/wakeup (MCS-SW). The pseudo code of the MCS-SW algorithm is shown in Figure 4.5 in a C language style. The algorithm is very similar to the original MCS algorithm [118]. For the detail about the MCS algorithm, please refer to [118]. Here we only point out the differences:

- **Line 1-2** Define the lock data structure. Compared to the MCS algorithm, the **qnode** data structure does not contain a flag. Thus the memory usage for the lock is reduced by 50%.
- **Line 6, and 17** When **acquire_lock** and **release_lock** is called, the calling function pass the address of a **qnode** as a pointer **I**. Assume **I** points to a qnode structure – **local_node**, which is a local variable of the calling function. Therefore, **local_node** is allocated in the stack, i.e., it is automatically allocated in the scratchpad memory of the calling threads. In the algorithm, the **local_node** (pointed by **I**) from each thread is used to built the linked-list based waiting queue. This protocol ensures that each thread in the waiting queue only spins locally on its scratchpad memory without generating any crossbar traffic to global memory.
- **Line 13** The lock is not available yet, the calling thread put itself into indefinitely sleep by suspending itself. In C64, such suspending can be completed with a single **sleep** instruction. After being suspended, the thread does not issue any more instructions unless it is waked up by another thread.

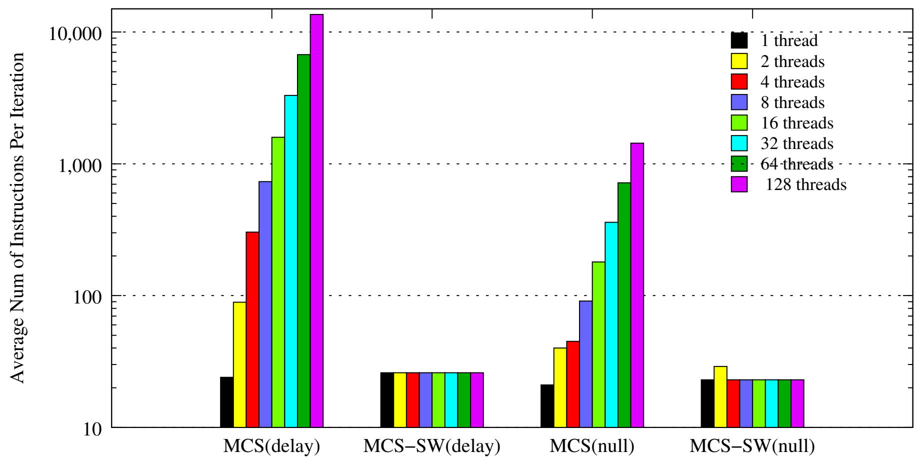


Figure 4.6: MCS vs MCS-SW with lock-null Microbenchmark

- **Line 28, and 31** When a thread tries to release the lock to its successor in the linked-list, it gets the address of its successor's **local_node** through the pointer **I**. Assume the address is **addr**. It is worth noting that **addr** is an address in the scratchpad memory of the successor thread. In C64 **wakeup()** can be conducted by generating a wakeup address of the successor thread from the **addr** via simple masking operation (taking two logical operation instructions). Then a **0** can be stored to the wakeup address of the successor thread to generate a hardware interrupt to wake it up. Please notice that the successor thread is waiting at **Line 13**. After waken up, it automatically becomes the owner of the lock.

Compared to MCS, which needs to operate on the "flag" in its **qnode** data structure, MCS-SW shows several cycles lower overhead than MCS when running with more than one thread. And they both show the same level of contention. Additionally, MCS-SW executes much less instructions per pair of lock/release than the original MCS lock. Actually with MCS-SW, each lock acquire/release operation takes a constant number of instructions, no matter how many threads contend for the lock. Unlike MCS, with which a thread keeps spinning on the local flag in scratchpad memory, with MCS-SW, a thread suspends after adding itself to the waiting queue. During the wait, the thread remains asleep and stops executing instructions until it is woken up by its predecessor. As shown in Figure 4.6, MCS-SW executes far less number of instruction per iteration compared with MCS. This is an important observation. Because when a thread is suspended, it consumes much less power. Because of the removing of the "flag" from the **qnode** data structure, the memory usage of MCS-SW is half of the original MCS algorithm. Given that scratchpad memory is one of the most important resource for a thread unit, this improvement is not trivial. In summary, MCS-SW is a time, memory, and power efficient spin lock algorithm for C64, and it is recommended to be used in the implementation of libraries and user applications.

4.2 Lock-based and Lock-Free Concurrent Data Structures on C64

In the last decade, lock-free concurrent data structures and algorithms have emerged in literature. A lock-free concurrent data structure is "one that guarantees that if multiple threads concurrently access that data structure, then some thread will complete its operation in a finite number of steps, despite the delay or failure of other threads" [83]. It is argued that lock-free concurrent data structures do not only avoid the inherent problem with locks, i.e. *priority inversion*, *convoying*, and *deadlock* [84], but also scale better and achieve higher performances than their lock-based counterparts. The main drawback seems to be the difficulty and complexity of designing general lock-free concurrent data structures. For that reason, most of the work has only focused on lock-free versions of commonly used basic data structures, such as stacks [90, 79], queues [159, 123, 70, 120], sets [106, 159, 70, 78, 119]. Since queues, stacks, and hash tables are precisely the data structures used in the runtime library that are protected with locks to guarantee mutual exclusion, we compare lock-free with lock-based implementations on C64. Because there is no *priority inversion* and *convoying* problem in C64 (due to the non-preemptive thread execution model), performance and memory contention are the only factors that we take into consideration.

For our study, we implement the *lock-free version* of the FIFO [123], LIFO [90], and hash table [119]. All these lock-free implementations adapt Michael's *Hazard Pointers* mechanism to guarantee safe memory reclamation of lock-free objects [122]. We implement the *lock-based* counterparts, which are straightforward, using our best spin lock algorithm: *MCS-SW* (see Section 4.1.4). We also implement a *lock-free-backoff version* for each data structure, which uses the same algorithms but an exponential increasing backoff is added before each retry, when a fail is encountered in the algorithm.

To evaluate the performance of these implementations we use microbenchmarks similar to those described in the spin lock study:

- For FIFO and LIFO, at each iteration, a thread performs either one enqueue/push or dequeue/pop operation randomly. A thread finishes after it completes 1,000 pairs of enqueue/push and dequeue/pop operations. After each operation, a small random delay is inserted before performing the next operation.

- The hash table is initialized with 25 buckets, and each bucket manages an ordered linked list. The hash table is also initialized with a load factor of l, which represents the average number of items per bucket. Each thread performs 10,000 operations, of which 20% are insertions, 20% deletions, and 60% searches. At each iteration, the operation to be performed is randomly determined, after which a small random delay is inserted. For the lock-based version, each bucket in the hash table is protected with a different lock to avoid unnecessary serialization. We experiment with three different load factors ($l = 5, 10, 50$).

For all microbenchmarks, we report the normalized execution time and number of contention. Both are normalized by number of threads. The amount of contention is reported by the simulator. When two or more threads compete for the same resource at the same cycle, the simulator increments the contention counter by one.

Figure 4.7 shows the normalized execution time of three versions of concurrent and thread-safe FIFO data structure, and Figure 4.8 shows the normalized number of

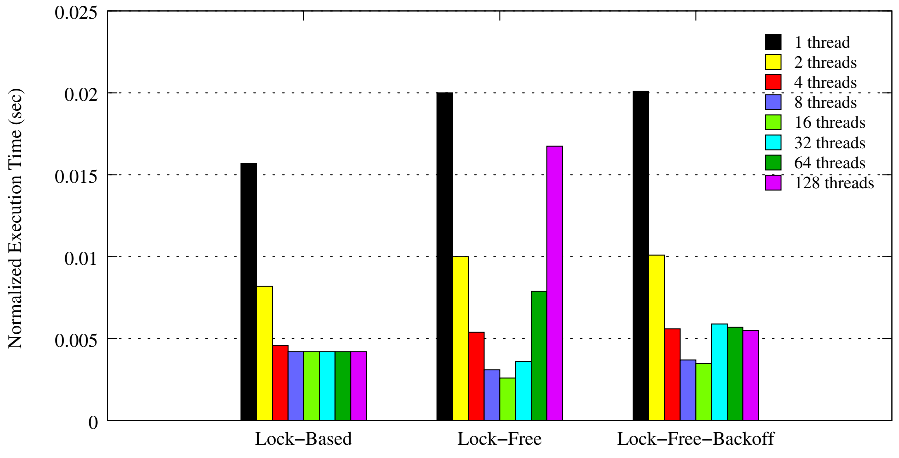


Figure 4.7: Normalized Execution Time of Lock and Lock-Free based FIFO Algorithms [sec] (Normalized by Number of Threads)

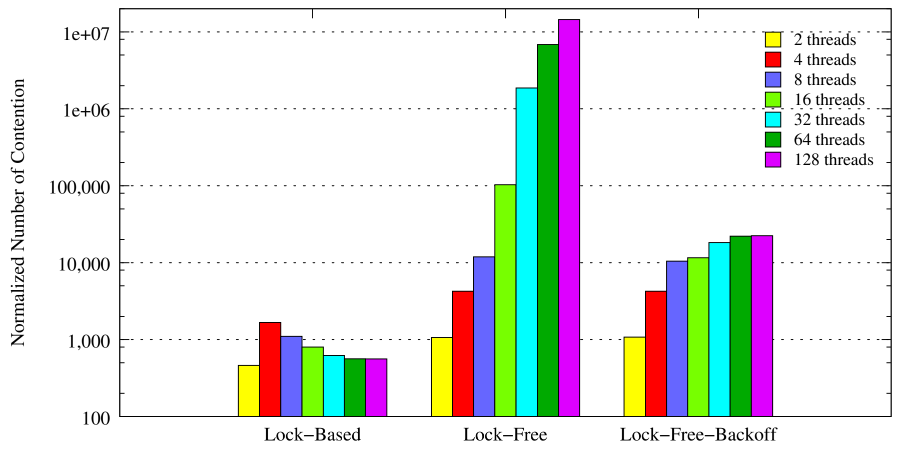


Figure 4.8: Normalized Contention of Lock and Lock-Free based FIFO Algorithms (Normalized by Number of Threads)

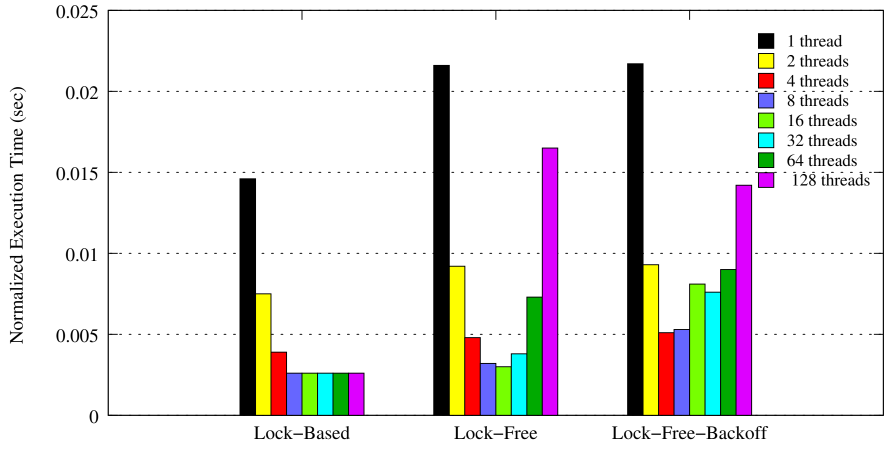


Figure 4.9: Normalized Execution Time of Lock and Lock-Free based LIFO Algorithms [sec] (Normalized by Number of Threads)

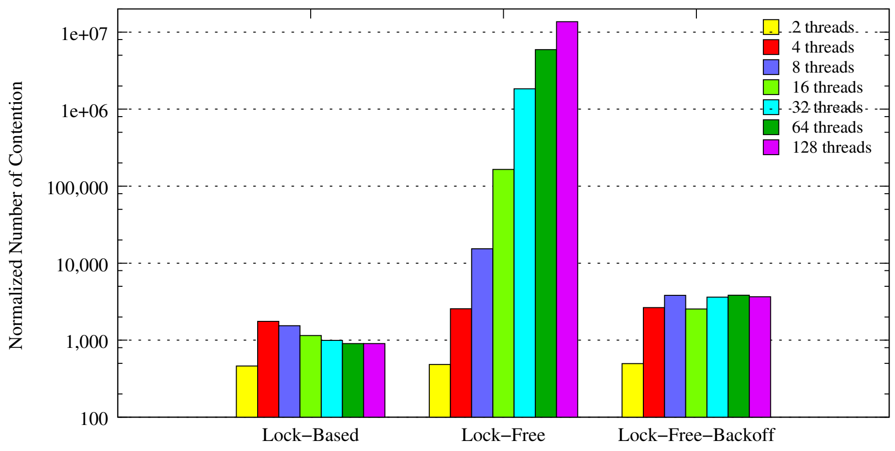


Figure 4.10: Normalized Contention of Lock and Lock-Free based LIFO Algorithms (Normalized by Number of Threads)

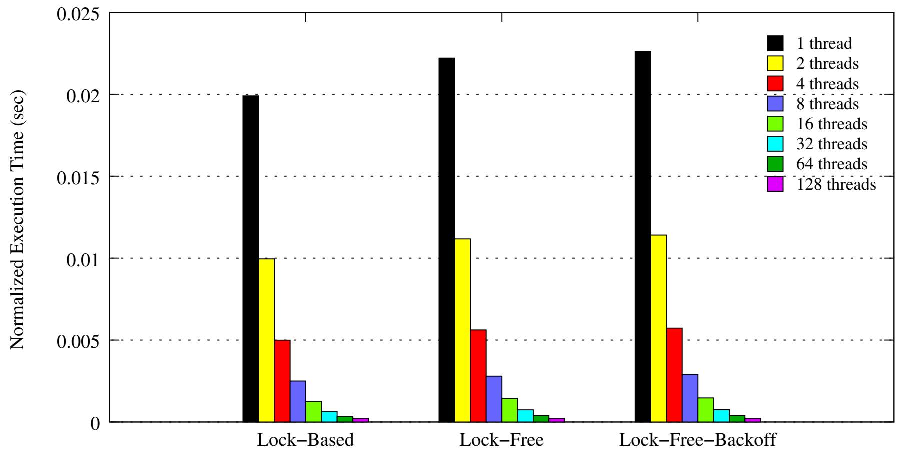


Figure 4.11: Normalized Execution Time of Lock and Lock-Free based HASH Algorithms [sec] (Normalized by Number of Threads, Average Load = 5)

contention for the three versions. From the figures, it can be observed that the lock-free version performs slightly faster than the lock-based version only when executed on medium number of threads (i.e. 8 threads, 16 threads, and 32 threads). When the number of threads is small or large, the lock-based version always costs much less execution time. In all cases, the lock-free version generates much higher contention (up to several orders of magnitude) than the lock-based version. The lock-free-backoff version alleviates the contention problem, but still generates up to one magnitude higher contention than the lock-based one. Therefore, in general, the lock-based version would be preferred for implementing the concurrent and thread-safe FIFO data structure for C64-like multi-core architecture. From Figure 4.9 and 4.10, we can draw the same conclusion for the LIFO data structure.

For the hash table, since it is initialized with 25 buckets, there is plenty of parallelism to be exploited when number of threads are small (smaller than 32). As shown in Figure 4.11, 4.13, and 4.15, all three versions achieves similar execution time when the number of threads are equivalent to or smaller than 32. When the average load is small, for instance, 5 nodes per bucket (see Figure 4.11), the lock-based version always executes

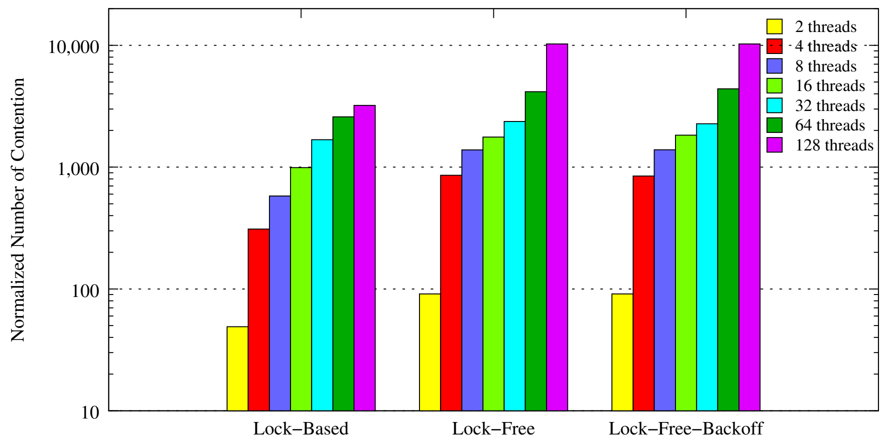


Figure 4.12: Normalized Contention of Lock and Lock-Free based HASH Algorithms (Normalized by Number of Threads, Average Load = 5)

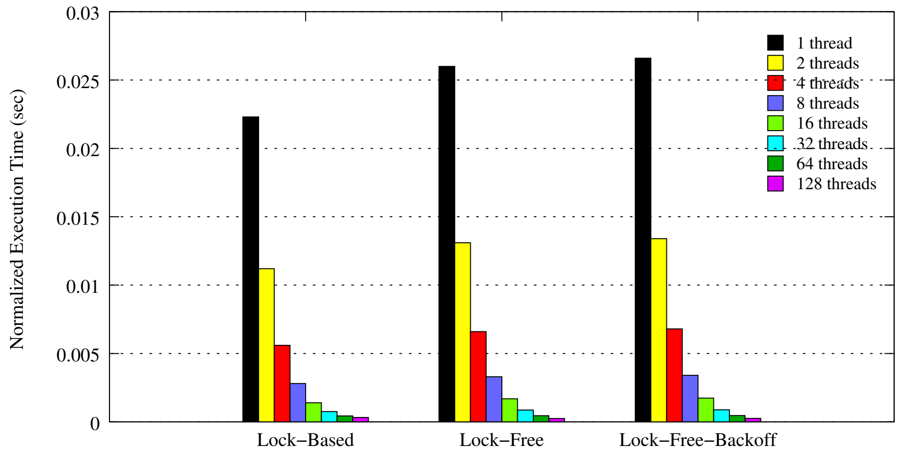


Figure 4.13: Normalized Execution Time of Lock and Lock-Free based HASH Algorithms [sec] (Normalized by Number of Threads, Average Load = 10)

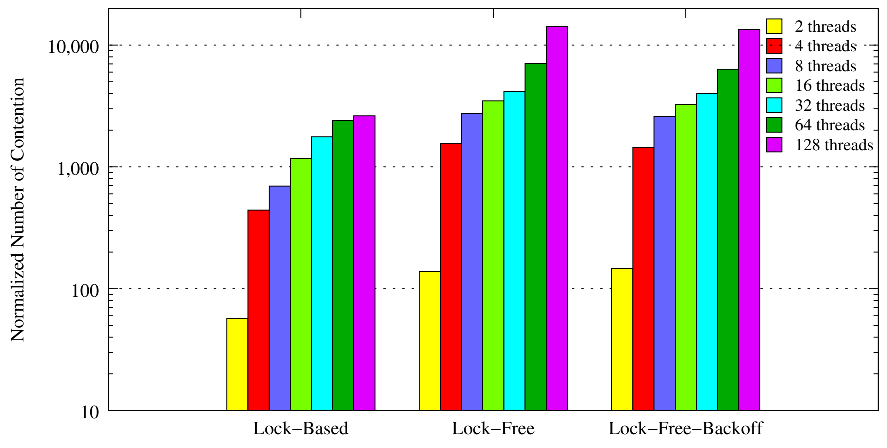


Figure 4.14: Normalized Contention of Lock and Lock-Free based HASH Algorithms (Normalized by Number of Threads, Average Load = 10)

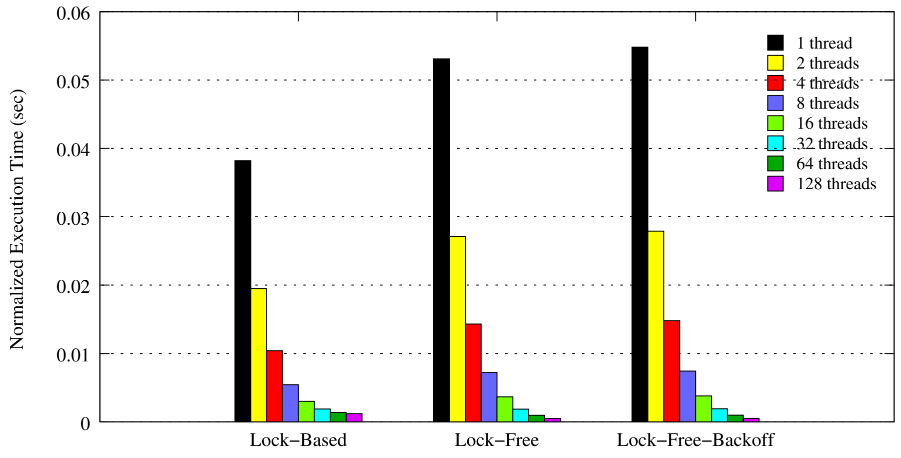


Figure 4.15: Normalized Execution Time of Lock and Lock-Free based HASH Algorithms [sec] (Normalized by Number of Threads, Average Load = 50)

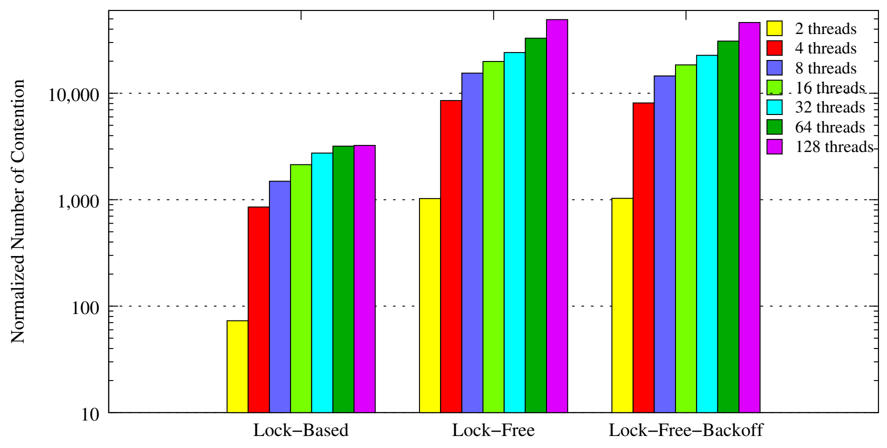


Figure 4.16: Normalized Contention of Lock and Lock-Free based HASH Algorithms (Normalized by Number of Threads, Average Load = 50)

faster than or equivalent to the lock-free versions due to the limited parallelism. The lock-based version also has the advantage of less memory contention (see Figure 4.12).

When the average load increases, for instance, 50 nodes per bucket, the parallelism can be exploited by the lock-based version is still limited, because each bucket is protected with a single lock. At any moment, only 1 thread can work on a bucket. If other threads also try to access the same bucket, they have to wait for the lock before proceeding. The lock-free version, on the other hand, allows multiple threads work on the same bucket as long as they do not modify (write to) the same node. Therefore, the lock-free version can exploit the fine-grain parallelism. As shown in Figure 4.13 and 4.15, when the number of threads are large, the lock-free versions demonstrate big performance advantage over the lock-based ones. For example:

- When the number of threads is 128 and the average load is initialized as 10, the lock-free version is 22.5% faster than the lock-based one.

- When the number of threads is 64 and the average load is initialized as 50, the lock-free version is 30.7% faster than the lock-based one.

- When the number of threads is 128 and the average load is initialized as 50, the lock-free version is 59.1% faster than the lock-based one.

In all cases (see Figure 4.12, 4.14, and 4.16), the lock-free versions always generates higher contention than the lock-based version. However, compared to the LIFO and FIFO cases, the contention generated by the lock-free algorithms is still in a reasonable range. The lock-free-backoff version does not improved the level of memory contention compared to the lock-free version, which means the commitment failure does not happen frequently in the lock-free hash table algorithm.

From the experimental results of three common concurrent data structures, we can draw the following observations for C64-like large-scale multi-core architectures:

- Contrary to common belief, the lock-free concurrent data structures do not always perform better than the lock-based counterparts on C64 chip architecture, not mentioning the complexity for designing and maintaining such lock-free algorithms.

- For data structures with limited parallelism, for example, FIFO, LIFO, and hash table with small load, the lock-based versions is a preferable choice because of the faster execution time, less memory contention, and algorithmic simplicity.

- For data structures with abundant fine-grain parallelism, for example, the hash table with large load, the lock-free version can be a candidate, if the scenario of a large number of threads accessing the same data structure is expected. In such case, the lock-based version suffers from the inability to exploit the fine-grain parallelism inherent in the data structure.

Based on above observations, the operating system or runtime system developers, library developers, and application developers for C64-like multi-core architectures should choose the appropriate implementation of the concurrent data structure, according to the design requirements, such as the common execution scenario, the maintainability, and other factors.

4.3 Evaluation of Language Constructs in OpenMP on C64

Given the intra-chip parallelism presented by a large-scale multi-core architecture, such as C64, it is important and challenging to provide high level parallel programming models for application developers to efficiently map the inherent parallelism in applications to a large number of on-chip processing cores. As a de facto industry standard for writing parallel programs on shared memory systems, OpenMP [133] is considered as one of the possible candidates. Parallel application developers express parallelism, work sharing, and synchronization through the OpenMP language constructs.

For the purpose of understanding the behavior and performance characteristics of OpenMP-based parallel programs on C64 like large-scale multi-core architectures, it is important to evaluate the performance of OpenMP language constructs, whose overhead accounts for up to 12% of the total execution time in some instances [60]. In this chapter, we focus on the performance characteristics of *synchronization language constructs/directives* of OpenMP. For completeness of an in-depth study on the performance behavior of OpenMP-based multithreading programs on multi-core architecture, we also presents the experimental results and analysis for OpenMP *scheduling policies* and *array clauses*.

To conduct a prototype study on high level parallel programming models, we ported the Omni-1.6 OpenMP compiler [105] to C64, and optimized the Omni OpenMP runtime system to adapt to the C64 hardware features [48] (see Section 3.3.3). Based on the number reported by the EPCC microbenchmarks [28], we measure and evaluate the performance characteristics of major synchronization OpenMP language constructs on the C64 large-scale multi-core architecture. In addition, we compare our results to previous work on conventional SMP systems and find remarkable differences. In some instances, the overhead on C64 is one order of magnitude lower.

4.3.1 EPCC OpenMP Microbenchmark

In order to understand the performance behavior of an OpenMP application, we use EPCC microbenchmarks [28] to measure the overhead of OpenMP language constructs. The basic methodology employed by EPCC is as follows. First, a reference time is obtained by executing a loop (or loop nests) sequentially without using any OpenMP directive. Then, the overhead is calculated by comparing this reference time with the execution time of the same code extended with OpenMP constructs.

There are three components of the EPCC microbenchmark. The *synchronization benchmark* measures the overhead of OpenMP work-sharing and mutual exclusion directives, such as PARALLEL, PARALLEL FOR, BARRIER, CRITICAL, ATOMIC, and REDUCTION etc.. The *scheduling benchmark* compares different scheduling policies – STATIC, DYNAMIC, and GUIDED. The *array benchmark* measures the overhead of the PARALLEL directives with the PRIVATE, FIRSTPRIVATE, and COPYIN clauses. We execute all three benchmarks on a single C64 chip with up to 128 threads and report the experiment results in the next section.

4.3.2 Synchronization Constructs

Figure 4.17 compares the overhead of the PARALLEL, the FOR loop, and the combined parallel work-sharing PARALLEL FOR constructs. It shows that the PARALLEL FOR construct has overhead similar to that of PARALLEL. This is because the overhead of the FOR construct is much smaller than PARALLEL and remains almost constant. From Figure 4.17 and Figure 4.18, we can also see that the overhead of FOR is only slightly higher than the overhead of BARRIER, which implies that the cost of FOR is mainly due to the implicit BARRIER at the end of the loop.

Note the high overhead of the SINGLE directive, especially when the number of threads increases to 128. This is because the implementation of SINGLE is very expensive in order to guarantee the semantics of SINGLE. The memory contention incurred to

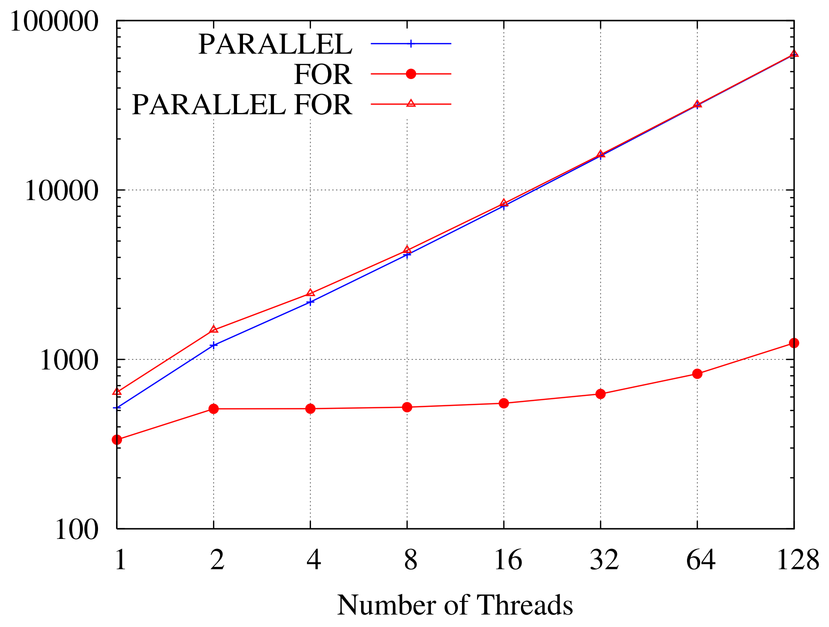


Figure 4.17: Overhead (cycles) of Synchronization Directives: PARALLEL, FOR, PARALLEL FOR

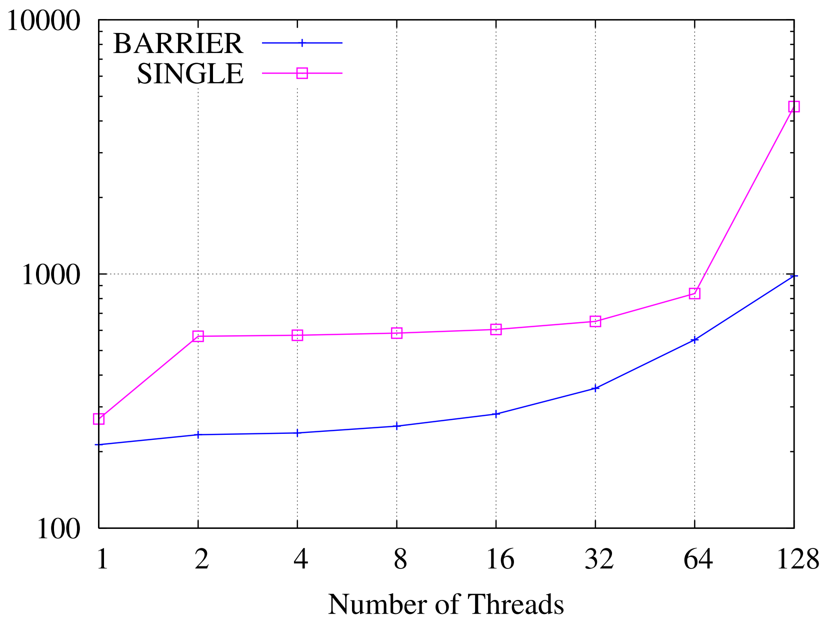


Figure 4.18: Overhead (cycles) of Synchronization Directives: BARRIER, SINGLE

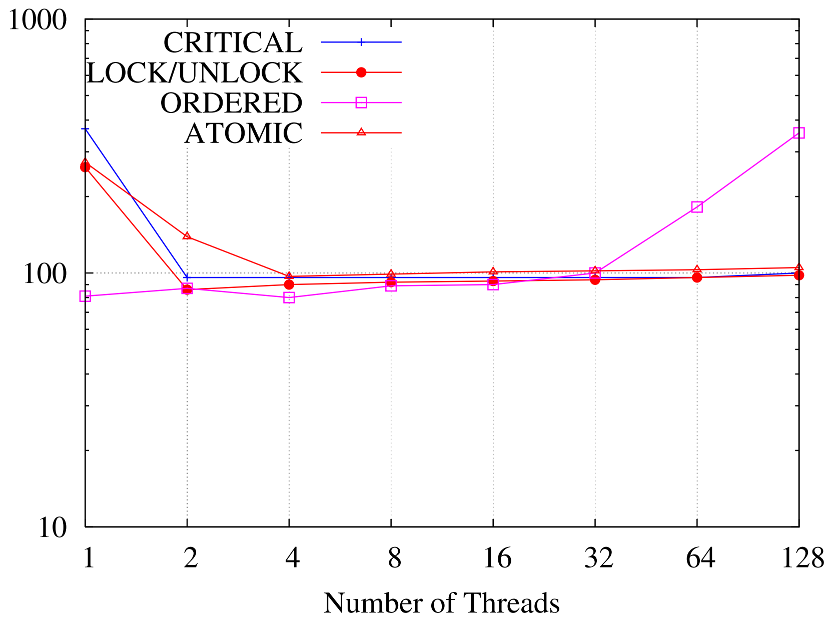


Figure 4.19: Overhead (cycles) of Synchronization Directives: Mutual Exclusion

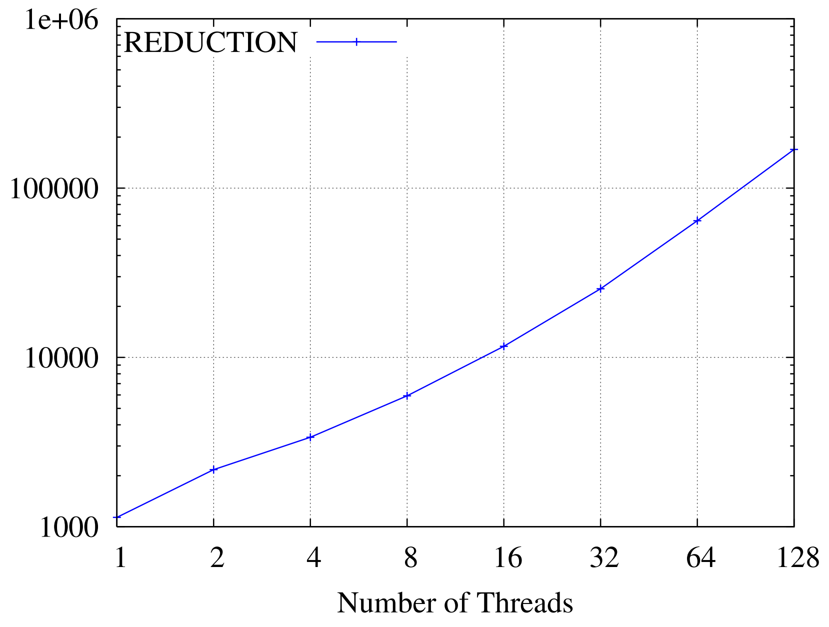


Figure 4.20: Overhead (cycles) of Synchronization Directives: Reduction

complete the SINGLE operation rises dramatically when the number of threads increases. SINGLE also suggests an implicit barrier.

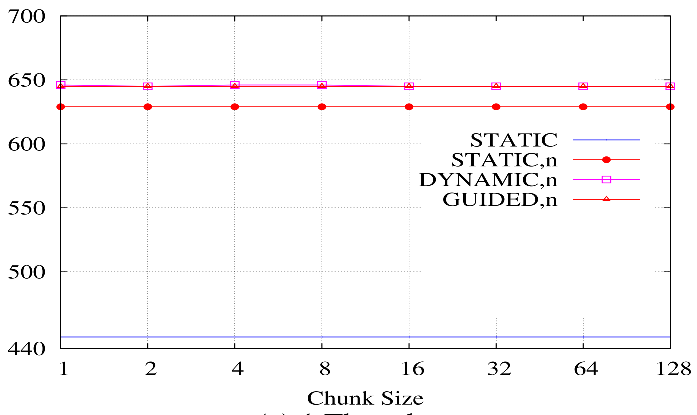
Because the OpenMP runtime library is carefully designed and tuned to map to the C64 hardware features, and the hardware components of C64 are tightly coupled in a single chip, the PARALLEL and BARRIER constructs incur much lower overhead than on conventional SMP systems. For example, a previous study [60] shows that the overhead of the PARALLEL construct reaches 120 microseconds (108,000 cycles) when running with 70 threads on a 72-node Sun Fire 15K system. Even while running with 128 threads, the same construct only presents a 63,020 cycles overhead. This observation implies that the thread management on a C64 like many-core architecture is much more efficient than common SMP environments.

We customized the well-known linked-list-based MCS spin-lock algorithm [118] to implement the low level lock acquisition and release primitives in the OpenMP runtime library [48]. Unlike common SMP systems where the overhead of lock increases with the number of threads, Figure 4.19 shows that the overhead of mutual exclusion constructs in OpenMP remain within the same range without increasing dramatically. Even for 128 threads, the CRITICAL directive costs only 154 cycles.

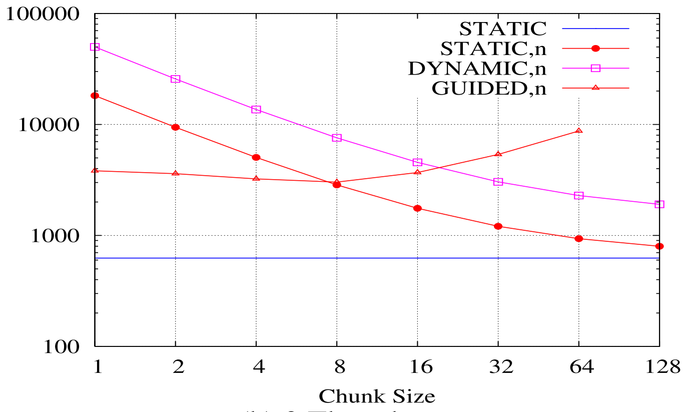
The of overhead of the REDUCTION construct increases exponentially, as shown in Figure 4.20. As future work, the reduction operation can be optimized in the runtime library by taking advantage of the C64's rich set of in-memory atomic instructions, which can perform certain operations, such as addition, subtraction, and various logical operations, atomically in memory. From our previous experiences with other benchmarks, such as Table Toy [49], we expect to improve the performance of REDUCTION dramatically.

4.3.3 Scheduling Policies

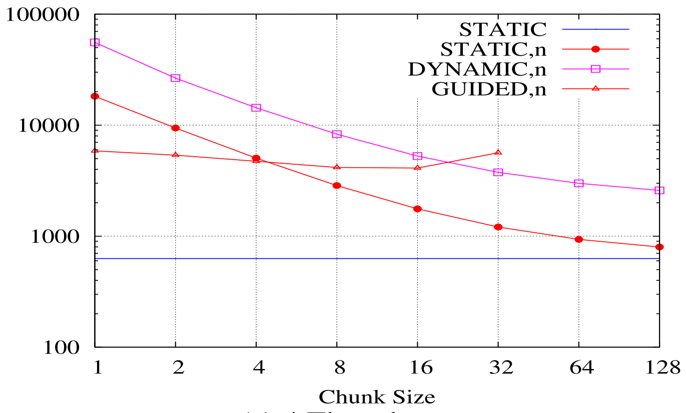
In OpenMP, there are three means for scheduling loop iterations among threads: STATIC, DYNAMIC, and GUIDED [133]. Please note that EPCC only reports the overhead of the GUIDED(n) scheduling policy for small values of n. Figure 4.21, Figure 4.22,



(a) 1 Thread



(b) 2 Threads



(c) 4 Threads

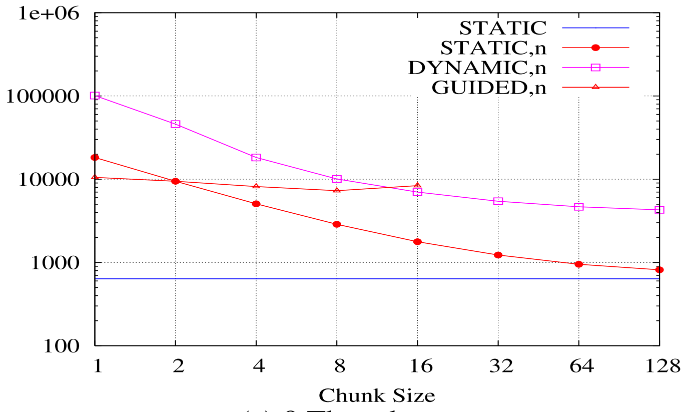
Figure 4.21: Overhead (cycles) of Scheduling Policies with 1 Thread 2 Threads, and 4 Threads

and Figure 4.23 compare different loop scheduling policies when running on 1 to 128 threads. It is apparent that STATIC and STATIC(128) always incur the lowest overhead in all cases. For the STATIC(n) policy, STATIC(1) causes the largest overhead, and the overhead decreases to the overhead of STATIC with increasing chunk size. Actually, the overhead of STATIC and STATIC(n) increases slowly for runs from 2 threads to 64 threads. When 128 threads are executed concurrently, the overhead is much larger than running with 64 threads because of the high memory contention.

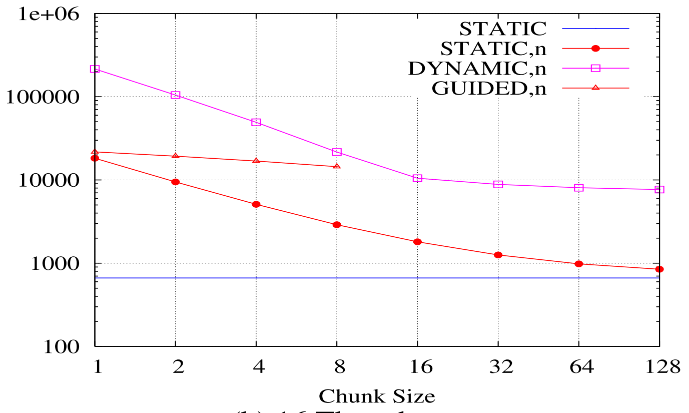
DYNAMIC(1), which is the most fine-grained scheduling policy, generates huge overheads (3,621 microseconds) when running on 128 threads. This is because the small chunk size causes frequent dynamic scheduling function calls, whose execution time is counted as the overhead. As a result, the overhead of static scheduling is multiple orders of magnitude smaller than dynamic scheduling.

The overhead of the GUIDED(n) scheduling is always better than the DYNAMIC(n). The GUIDED(n) policy starts with a large chunk size, then gradually decreases it to n. Figure 4.21, Figure 4.22, and Figure 4.23 also demonstrate that the STATIC policy always incurs lower overhead than the GUIDED policy. The overheads measurement suggests that on C64 OpenMP programmer should consider the STATIC scheduling policy as the first option for loop scheduling, given the tasks can be statically balanced. Only if the benefit of dynamic load balancing surpasses the scheduling overhead, the dynamic and guided scheduling policy are worth being chosen.

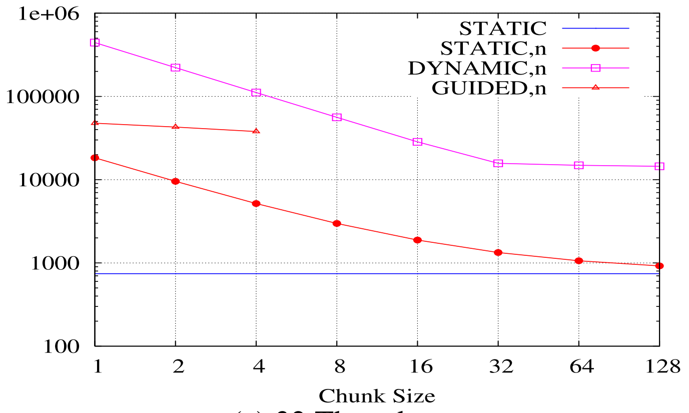
In the OpenMP runtime library, the dynamic and guided scheduling functions are implemented to frequently access the thread descriptor, and sometimes access the master thread's descriptor by acquiring a lock first. By taking advantage of the explicit programmable multi-level memory hierarchy of C64, we place the thread descriptor of each work thread into its own scratchpad memory, which guarantees very fast accesses, i.e., 1 cycle for a store, 2 cycles for a load. The master thread's descriptor is placed in on-chip global memory, whose access latency is longer than scratchpad but smaller than



(a) 8 Threads



(b) 16 Threads



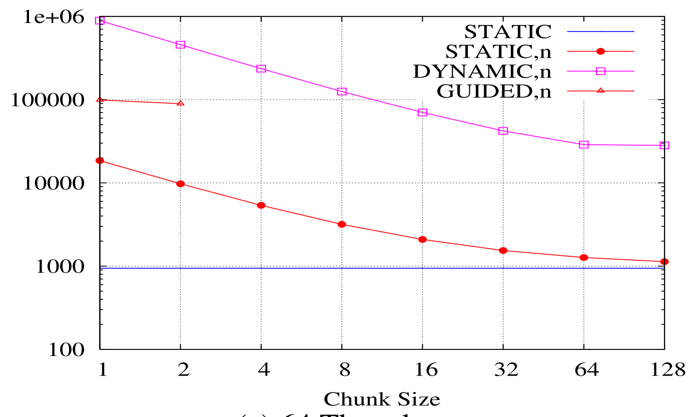
(c) 32 Threads

Figure 4.22: Overhead (cycles) of Scheduling Policies with 8 Threads 16 Threads, and 32 Threads

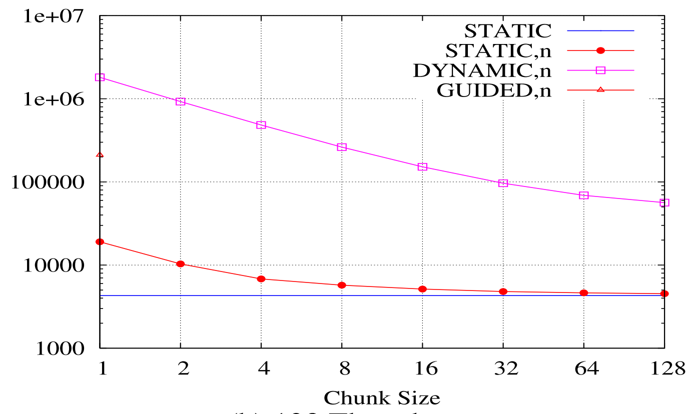
off-chip memory. By leveraging the C64's in-memory atomic instruction and thread level execution support, the lock/unlock primitives used to guarantee the mutual exclusion for accessing the master thread's descriptor are efficiently implemented as demonstrated in Figure 4.19 [48]. Therefore, compared with common SMP systems, the overhead of loop scheduling is at least an order of magnitude lower on a C64-like large-scale multi-core architecture. For example, as reported in [60], when running on a 72-node Sun Fire 15K, the DYNAMIC(1) incurs an overhead of around 27M cycles (30,000 microseconds) with 24 threads, while on C64 it costs 0.44M cycles with 32 threads, and 1.8M cycles with 128 threads. The overhead of STATIC scheduling is 9,000 cycles with 24 threads on a Sun Fire 15K [60], but only 743 cycles with 32 threads, and 4,298 cycles with 128 threads on C64.

4.3.4 Array Clause for PARALLEL

The *array microbenchmark* measures the overhead of the PARALLEL directive with the PRIVATE, FIRSTPRIVATE, and COPYIN clauses. In the current design of C64 system software, the stack of a thread is placed in its own scratchpad memory and the size of the stack is limited. As a result, in our experiments, we can only run the benchmark with an array size smaller than or equal to 729. As a work in progress, the C64 toolchain will provide support for automatic stack extension, a feature that allows applications that require more stack than available to continue running at the expense of performance. When the stack area is exhausted, the runtime system automatically relocates the stack into off-chip memory. Notice the relocation is performed very quickly, as it requires setting a few registers and copying a few locations from the stack (but not all). If at a later point, the stack shrinks, the runtime system undoes the changes and sets the program stack back to scratchpad memory. However, in order to achieve good performance, it is not recommended to declare large arrays on the stack (as automatic variables), or make deep recursive function invocations in the program.



(a) 64 Threads



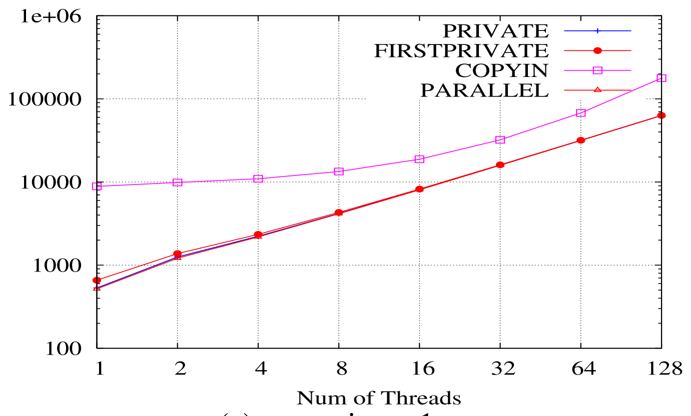
(b) 128 Threads

Figure 4.23: Overhead (cycles) of Scheduling Policies with 64 Threads and 128 Threads

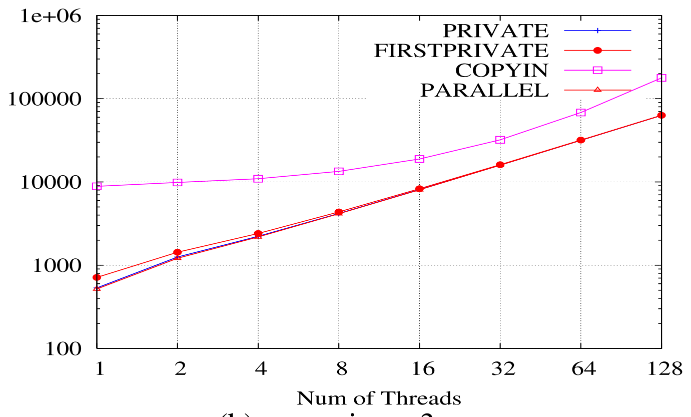
As shown in Figure 4.24, Figure 4.25, and Figure 4.26, the PRIVATE and FIRST-PRIVATE clauses have similar overheads (the overhead of FIRSTPRIVATE is slightly higher). Compared with the PARALLEL constructs without any data-sharing attribute and data copying clauses, it is also clear that the curves of PRIVATE and FIRSTPRIVATE almost match the curve of PARALLEL constructs. This means attaching the PRIVATE or FIRSTPRIVATE clause to the PARALLEL construct incurs negligible costs. In both cases, the compiler directly allocates the private array in the stack of each thread, which incurs no overhead at runtime.

For FIRSTPRIVATE, the C library function *bcopy* is used to initialize the private array by copying the contents of a global array. In the standard C library of C64, routines like *memcpy*, and *bcopy*, are optimized and fine tuned. They are aware of the explicit memory hierarchy. The C64 load and store multiple instructions are used to exploit the memory bandwidth and save cycles from not issuing multiple instructions. In addition, the instruction sequences are manually scheduled to hide memory accessing latencies. Since the array size used in our experiments is small, the copying is performed very efficiently. Therefore, no significant overhead is observed for FIRSTPRIVATE.

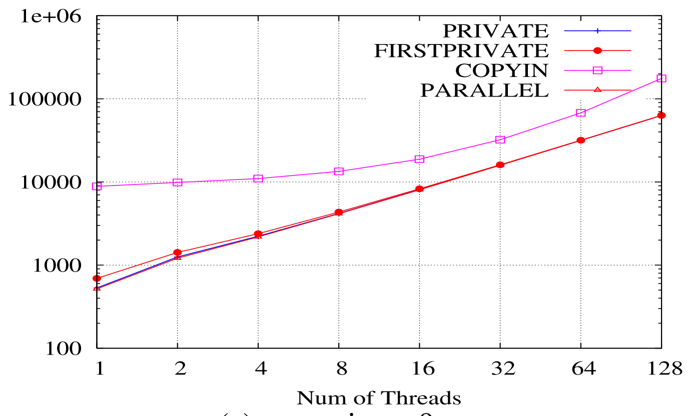
From Figure 4.24, Figure 4.25, and Figure 4.26, the COPYIN clause generates one order of magnitude larger overhead than the other two clauses. By attaching the COPYIN clause to the PARALLEL directive, the Omni OpenMP compiler generates codes that dynamically allocate the storage for thread private data. The heap manager allocates the thread private data in the on-chip global memory. There are also overheads from lock/unlock operations for using the memory allocator. Moreover, since the data is allocated in the global memory at runtime, the latency of memory accesses in the loop body is much higher than accessing scratchpad memory. This is the reason why COPYIN has much larger overhead than PRIVATE and FIRSTPRIVATE. This suggests a scope for possible optimizations either in the compiler or the runtime system.



(a) array size = 1

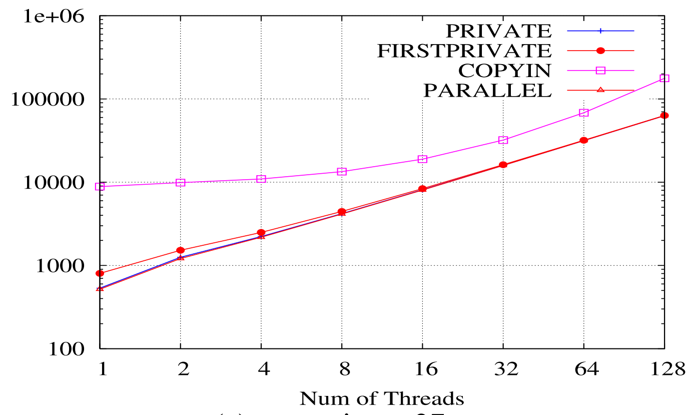


(b) array size = 3

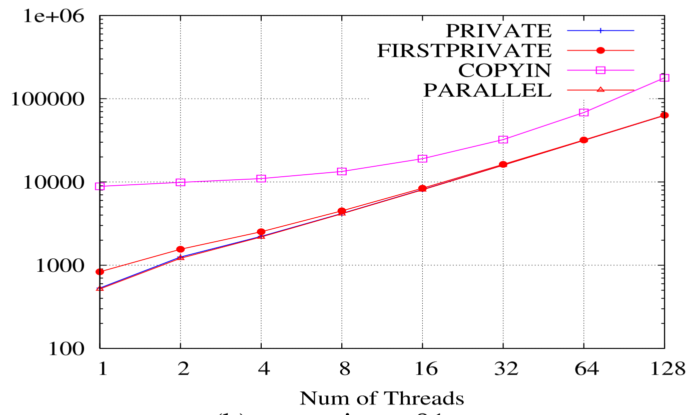


(c) array size = 9

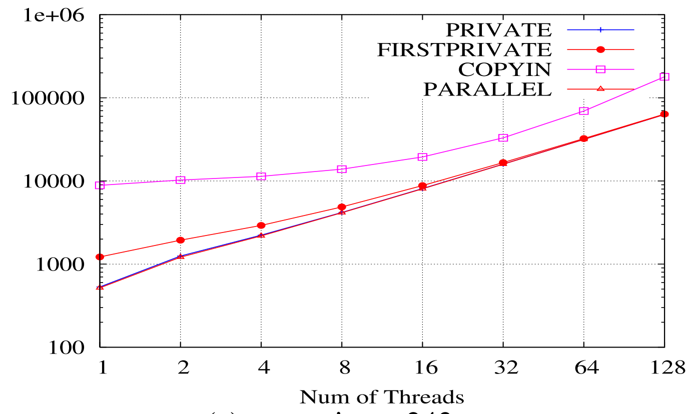
Figure 4.24: Overhead (cycles) of Data-Sharing Attribute and Data Copying Clauses: (a) array size = 1 (b) array size = 3 (c) array size = 9



(a) array size = 27



(b) array size = 81



(c) array size = 243

Figure 4.25: Overhead (cycles) of Data-Sharing Attribute and Data Copying Clauses: (a) array size = 27 (b) array size = 81 (c) array size = 243

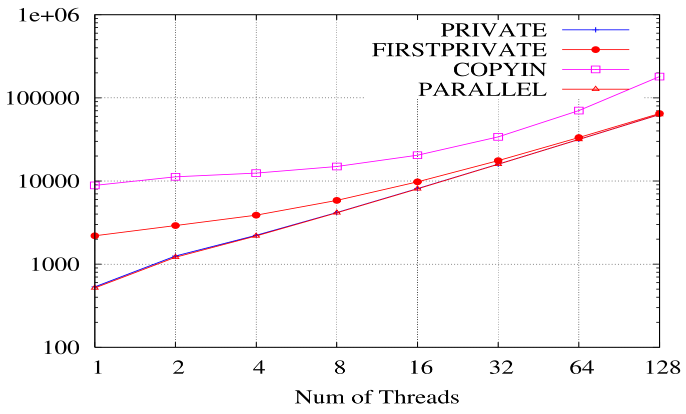


Figure 4.26: Overhead (cycles) of Data-Sharing Attribute and Data Copying Clauses: array size = 729

4.3.5 Related Work and Summary

Most of the previous work on performance characterization of OpenMP were conducted on the general purpose commercial shared memory SMP systems [28, 105, 29, 26, 136, 60]. Liao et. al. [109] evaluated OpenMP on chip multithreading platforms. However, the chip multiprocessor (UltraSPARC III) evaluated in the paper only has two cores. To the best of our knowledge, our work is the first attempt to measure and evaluate the performance characteristics of OpenMP language constructs on a C64-like (160 cores) like large-scale multi-core architecture.

In [9, 142], the authors presented the experiment results of OpenMP NAS benchmarks on an experimental Cyclops architecture. It is worth noting that this experimental architecture was a preliminary design of the Cyclops architecture and it is never to be built, while the first C64 system is planned to be delivered in 2007. Also, this experimental Cyclops architecture included data caches in the design, and the C64 system employs scratchpad memory technology instead of data cache. Neither [9] nor [142] conducted performance characterization of the OpenMP language constructs, since that was not the purpose of those two papers.

In this section, we reports and analysis the performance characteristics of OpenMP language constructs on the Cyclops-64 chip architecture. In addition, we compare our results to previous work on conventional SMP systems [60] and find remarkable differences. In many instances, the overhead on C64 is one order of magnitude lower. With this study we provide insight regarding the following aspects of software development on large-scale multi-core architectures: (1) we provides the performance characteristics of synchronization operations in the context of a high-level parallel programming model – OpenMP; (2) we provide application developers a better understanding of the behavior of OpenMP programs on a large-scale multi-core architecture; (3) we give library and compiler developers hints regarding possible optimizations and/or language extensions specific to multi-core architectures, specifically, to efficiently exploit multi-level memory hierarchies and fast intra-chip synchronization mechanisms; (4) using the OpenMP runtime library optimization as an example to understand the pros and cons of the C64 architecture, we provide software developers hints on how to write and optimize programs for this type of architecture. To the best of our knowledge, this work is the first attempt that measures and evaluates the performance characteristics of OpenMP language constructs on large-scale multi-core architecture with up to 160 cores.

4.4 Summary

Large-scale multi-core architectures (with 10s cores and beyond) are emerging. It is critical to understand the performance behavior of parallel program on such approaching large-scale multi-core architectures. The performance characteristics of synchronization mechanisms is essential for determining the performance behavior of parallel programs. Little experience has been gained for multi-core chips with more than 100 cores in previous studies and literatures.

To complement the absence of such a study, this chapter presented thorough performance measurement and evaluation of a range of widely used synchronization mechanisms on the IBM 160-core Cyclops-64 chip architecture. First, we evaluated the performance of six most widely used and cited spin-lock algorithms with two representative microbenchmarks. On C64-like large-scale multi-core architectures, not surprisingly, the simple *test_and_set* based lock achieves best performance when there is no contention; the MCS lock algorithm is suitable for high-contention scenario. We then presented a customized version of MCS on C64 as MCS-SW. Compared to MCS, MCS-SW demonstrates lower overhead, same level contention, half of the memory usage, and far less number of instructions executed for per pair of lock acquisition/release operations.

Second, we evaluated and compared the performance of three common lock-free concurrent data-structures with their lock-based counterparts. Contrary to common belief, the lock-free concurrent data structures do not always perform better than the lock-based counterpart on C64 chip architecture, not mentioning the complexity for designing and maintaining such lock-free algorithms. Based on our experimental results, for data structures with limited parallelism, the lock-based versions is a preferable choice because of the faster execution time, less memory contention, and algorithmic simplicity; for data structures with abundant fine-grain parallelism, the lock-free version can be a candidate, if the scenario of a large number of threads accessing the same data structure is expected. The operating system or runtime system developers, library developers, and application developers for C64-like multi-core architectures should choose the appropriate version of the concurrent data structure to implement by referencing to our performance evaluation results.

Third, we chose a high-level parallel programming model, OpenMP, to further exploited the performance behavior of synchronization mechanisms within the context of a real multithreading programming environment on C64. We evaluated the overhead of all common OpenMP language construct with the EPCC benchmark. In addition, we compare our results to previous work on conventional SMP systems and find remarkable differences. In many instances, the overhead on C64 is one order of magnitude lower.

Chapter 5

SSB: SYNCHRONIZATION STATE BUFFER

5.1 Motivation

The design of high-performance processor chips is rapidly moving towards the multi-core architecture that integrates a large number of tightly-coupled processing cores on a single chip [46, 27]. For instance, the IBM Cyclops-64 petaflop supercomputer project features a chip architecture that integrates more than one hundred thread units and memory banks on a single chip [52, 53]. Another example is an research prototype many-core design with 80 cores in a single chip, which is just announced by Intel [91, 160]. In order to fully utilize the on-chip parallelism provided by such large-scale multi-core chips, it is important to exploit the fine-grain parallelism that is available in applications. The granularity of parallelism that can be efficiently exploited in such processors is often restricted by the lack of effective architectural support for efficient fine-grain synchronization. Software-only solution, with very limited architectural support, often lead to high synchronization overhead, high storage cost, and poor scalability. It is often difficult or even impossible to harness fine-grain parallelism at the compilation time.

Consider the example shown in Figure 5.1. The example shows the kernel loop in Random Access HPCC benchmark [137] implemented using critical section in OpenMP

```
#pragma omp parallel for private(ran,i,idx) shared(y,N,size)
for(i = 0; i < N; i++){
   ran = rand();
   idx = ran & (size - 1)
#pragma omp critical
 {
   y[idx] = y[idx] op ran;
 }
}
```

Figure 5.1: Random Access with DOALL Loop

```
for ( i=1 ; i<n ; i++ )
   for ( k=0 ; k<i ; k++ )
      W[i] += b[k][i] * W[(i-k)-1];
```

Figure 5.2: Linear Recurrence Equations

API. The critical section requires the read-modify-write operations in the loop are performed atomically. [1] Unstructured references like the one shown in Figure 5.1 are impossible to analyze at compile time. Therefore, the compiler can only assign a single lock for the whole table `y[]`. Now if the table `y[]` is much larger than the number of threads and if one uses high quality uniform random number generator, the chance of conflicts to access the same `y[idx]` is very low. A single lock for the table introduces unnecessary serialization. Efficient run-time fine-grain synchronization mechanism is necessary to exploit such inherent parallelism by avoiding unnecessary serialization.

Now consider the Livermore Loop 6 shown in Figure 5.2, which represents a general linear recurrence equations that is widely used in linear algebra computations. As shown in Figure 5.3, the outer loop computes the array `W`, and at each iteration `i`, `W[i]`

[1] The original benchmark allows data races as long as the percentage does not exceed 1%. In the context this dissertation, we enforce the mutual exclusion to examine fine-grain synchronization mechanisms.

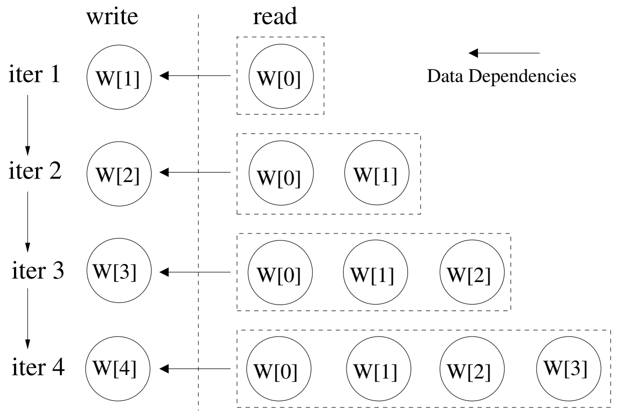


Figure 5.3: Characteristics of Linear Recurrence Equations

depends on values computed in all previous iterations, that is, `W[i]` depends on `W[0]`, `W[1]`, ... , `W[i-1]`. Such cross-iteration dependencies of array `W` makes it very difficult to parallelize this loop at compilation time [156]. Once gain, in order to exploit the fine-grain parallelism within the loop, an efficient fine-grain synchronization mechanism is required to enforce the read-after-write data dependencies among concurrent threads.

5.2 Challenges

On large-scale multi-core chips (with 16 to 100 cores and beyond), the on-chip storage (memory) available per processor core is far less (often 1-2 orders less) than traditional single core microprocessors. On the other hand, there are plenty of distributed resources (e.g. large number of thread and memory units, ample on-chip interconnection bandwidth, etc.) available to facilitate efficient fine-grain coordination between processing cores and memory. Therefore, the following new challenges are emerging with respect to fine-grain synchronization solutions in large-scale multi-core architectures, they should:

- be scalable and can fully exploit the parallelism due to the distributed on-chip resources.

- be supported with limited on-chip resources.
- incur low synchronization overhead.
- be able to support a variety of synchronization functionalities with modest hardware cost.
- be able to smoothly and efficiently handle the cases where the precise synchronization point cannot be resolved statically at compile time.

5.3 SSB: A Novel Fine-Grain Synchronization Solution

There are several design choices that one can architect to implement fine-grain synchronization. For instance, HEP [147], Tera [11], MDP [47], Sparcle [4], M-Machine [98], the MT processor in Eldorado [57], and others use hardware bits as tags (e.g., *full/empty bits*) to support word-level fine-grain synchronization. These designs tag the entire memory of the machine by associating additional access state bits with each word in memory. Dataflow model-based architectures that use the I-structure [20] and M-structure [25] like fine-grain synchronization also exploit similar designs. Given that on-chip memory is one of the most precious resources for many-core chips, one down side of such design choices is the overhead and the cost associated with tagging every word in the memory.

To address the problem of such high-cost synchronization mechanisms, we made one key observation: *at any instance during the parallel execution only a small fraction of memory locations is actively participating in synchronization.* To further elaborate on the key observation, consider the kernel loop shown in Figure 5.1. Let us assume a non-preemptive thread model. Now let T be the number of active threads and so $T \ll N$, where N is the size of the table `y[]`. In the example, we can then observe that at any instance during the parallel execution, the number of memory locations S that are *actively participating in synchronization* is less than or equal to T, that is, $S \leq T$, and therefore $S \ll N$.[2] In other words, at any instance, only a small part of the table need to be actively synchronized (i.e. locked). Therefore, rather than supporting fine-grain synchronization by tagging every word (in the table), one can focus on recording and managing synchronization states of only those actively synchronized memory words. One could make a similar observation for the example Livermore Loop 6 kernel shown in Figure 5.2. [3] Observe that the different synchronization characteristics of the two kernel loops. The Random Access loop in Figure 5.1 uses mutual exclusion synchronization, whereas Livermore Loop 6 in Figure 5.2 uses produce-consumer synchronization. For both, mutual exclusion (e.g. lock/unlock) and data synchronization (e.g. synchronized write/read) the associated states of synchronized memory location(s) can be naturally released when the synchronization is completed (e.g. via unlock or synchronized reads).

Based on the key observation, we introduce a novel synchronization architecture, with a modest hardware extension to many-core architectures, called *Synchronization State Buffer* (SSB). SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of *active synchronized data units* to support and accelerate word-level fine-grain synchronization. SSB caches the states of memory locations that are currently accessed by special SSB synchronization operations. An interesting aspect of our SSB design is that it avoids enormous memory storage cost, and still creates an illusion that each word in memory is associated with a set of states that can be used to support word-level fine-grain synchronization. We will show later in the paper that how each SSB entry can encode larger states of memory. We will use these larger states to support more synchronization functionality than the previous proposals. SSB can supports a rich set of synchronization functionalities. In our current design, SSB

[2] Even in a preemptive thread model, the number of threads is normally much less than the size of memory for a practical multithreading program. Therefore $S \ll N$ generally holds.

[3] The key observation for the Livermore loop 6 is not straightforward. Later in Section 6.5, we will discuss the details.

can be used to enforce mutual exclusion and read-after-write data dependencies between threads. For mutual exclusion, SSB supports different fine-grained locks, including word-level read, write, and recursive locks. For data synchronization, SSB allows fine-grained low-overhead synchronized read and write operations at word-level in memory. SSB supports several modes of data synchronization, including two single-writer-single-reader modes, and one single-writer-multiple-reader mode. SSB is programmable and compiler optimizations can be used to efficiently facilitates fine-grained synchronizations to help multithreading programs exploit fine-grained parallelism inherent in applications.

Although the SSB for each memory bank is small, our experiments show that an SSB with limited number of entries for each memory bank is sufficient to support common multithreading programs, even those running with a large number of threads. Moreover, in order to use fine-grain data synchronization to efficiently parallelize loops, various compiler optimization techniques have been developed to minimize the amount of fine-grain synchronization added for parallelized do-across loops [108, 124, 103, 37, 129, 15, 138] and others. Those techniques can be combined with SSB-based hardware support to efficiently parallelize do-across loops. This is especially useful when the synchronization resource requirements are greater than the number of SSB entries provided. Furthermore, some techniques can also be adapted to our framework, for instance, to group multiple data synchronizations into one.

Because of the relatively smaller storage cost, each SSB entry can afford to encode larger states – thus can support more synchronization functionality than the previous proposals. To avoid the bottleneck in a centralized organization and enhance the scalability, each memory bank is attached with its own SSB . Therefore, SSB, which is as distributed as other on-chip resources (e.g. thread units, and on-chip memory banks, etc.), can take full advantage of ample on-chip interconnection bandwidth. Also, previous studies [102, 165] have shown that fine-grain synchronization results in successful synchronization in most cases. Therefore, our SSB design ensures that the cost of a successful synchronization should be very small.

To understand the design space of SSB we implemented our solution in the context of IBM Cyclops-64 multi-core chip architecture as a case study. We extended the IBM Cyclops-64 architecture simulator with the new SSB architectural features. We then designed a hardware/software interface for SSB access and management. Using detailed simulation with microbenchmarks and application kernels, our experimental results demonstrate the effectiveness and efficiency of SSB solution for supporting fine-grain synchronization.

- For mutual exclusion: our method exploits the ample parallelism that often exists in operations on different elements of the concurrent data structures. Using distributed fine-grain locking on each memory unit, we avoid the unnecessary serialization of those operations without incurring any extra memory usage. In addition, the SSB has also resulted in considerable reduction of the overhead of each individual lock/unlock pair. Also, compared to the software-only solutions, up to 84% performance improvement has been observed for the benchmarks we tested.

- For read-after-write dependence synchronization: our method encourages the exploration of do-across style loop-level parallelism - where *loop-carried* data dependence can often be directly implemented by the application of our fine-grain solutions and the removal of barriers. Our experimental results demonstrate significant performance gain due to the use of such fine-grain synchronization. For instance, by adopting a fine-grain synchronization based parallelization scheme, we observe a 312% performance improvement over the coarse-grain based approach when solving linear recurrence equations.

- The experiments also demonstrate that 1) a small SSB for each memory bank is normally sufficient to record and manage the access states of outstanding synchronizing data units for multithreading programs, and 2) most of fine-grain synchronizations are successful (e.g. successful lock acquisition, and synchronized read).

5.4 Design Principles of Synchronization State Buffer

In this section we lay the foundation for SSB and present the principles for efficient implementation of fine-grain synchronization using SSB .

5.4.1 Large-Scale Multi-Core Architecture

Unlike traditional uniprocessor chips, where few architecture designs dominate, researchers in multi-core chips have not yet reached a consensus. Architects and researchers are actively exploring the design space of multi-core chip, which is currently in a state of flux. The design choices for efficient implementation of fine-grain synchronization solution are likely to be strongly influenced by the underlying architectural design and model. In this dissertation, we focused on a class of large-scale multi-core architectures where a large number of simple cores and memory modules are integrated on a chip and connected via an on-chip interconnection network. Examples of these multi-core/many-core chips include the recent announcement of the Intel terascale chip [160] and the Larrabee mini-cores chip [2], and the IBM Cyclops-64 [52, 53] (C64) chip architecture. In this dissertation, we have implemented SSB in the context of the C64 architecture.

One important feature of such large-scale multi-core architectures is that the amount of on-chip storage per core is far less than traditional single core processors - by up to one to two orders of magnitude. Therefore tagging every on-chip memory word for fine-grain synchronization can incur high cost. One of our design objective in SSB is to avoid such cost.

One direction in multi-core chip design is to exploit *explicitly programmable local memory store* (e.g., scratchpad memory) for each processing core rather than coherent data cache. For instance, the IBM Cell processor [72, 71], Cyclops-64 [52, 53] chip architecture, ClearSpeed CSX processor architecture [40], and other multi-core architectures, employ such local store approach. The local store approach allows denser hardware implementation and simplifies the microarchitecture by avoiding the complexity of tag-match compare and late hit-miss detection, miss recovery, and coherence management associated with cache hierarchies [72]. From the software perspective, non-deterministic memory access latencies of cache always negatively affect compiler scheduling and optimizations. On the other hand, the local store with low and deterministic access latency can offer aids to the effectiveness of many complier-based static scheduling and optimizations, such as instruction scheduling, loop unrolling, and software pipelining [55]. Unlike many synchronization mechanisms built on coherent cache architectures, SSB makes no such assumption, and thus can be naturally implemented as the fine-grain synchronization mechanism for large-scale multi-core architectures with the local store approach.

Another important feature of such large-scale multi-core architectures is that they often employ a large number of simple cores. For example, the IBM Cyclops-64 (C64) chip contains 160 cores (also called thread units). C64 system software model and associated programming and execution environment are centered around TiNy Threads [51, 50]. One feature of the TiNy Threads is the efficient support of a non-preemptive thread model: the core on which a thread is running is simply made idle when the thread is suspended. Under a large-scale multi-core architecture such as C64, thread context-switching can be particularly costly due to two reasons. First since on-chip memory is precious and limited, saving the context of a large number of threads in on-chip memory can become prohibitively expensive and impractical. Second, saving the context in off-chip memory suffers from high latency and low bandwidth. The effectiveness of the non-preemptive model has been demonstrated through the mapping of a number of applications onto C64 [86, 152, 161, 38]. An assumption for designing and implementing SSB that we make throughout this chapter and next chapter is the non-preemptive thread execution model.

5.4.2 Formalization of the Key Observation

Recall the key observation that at any instance during the parallel execution only a small set of memory locations are actively participating in synchronization. We formalize this simple observation as follows: Let T be the number of non-preemptive active threads and let $N = M \times B$ be number of memory locations, where M is the size of each memory bank and B is the number of memory banks. Observe that T is usually far less than $M \times B$, that is, $T \ll M \times B$. Now let $S(t)$ be the number of active synchronized memory locations at any instance t. In other words, $S(t)$ is the amount of synchronization in an application at any instance t, and is given by:

$$S(t) \leq \alpha(t) \times T, \tag{5.1}$$

where $\alpha(t)$ indicates the maximum number of distinct memory locations synchronized by a thread at any instance t. Therefore a large-scale multi-core architecture can take advantage of the SSB design whenever the following relation holds:

$$S(t) \leq \alpha(t) \times T \ll M \times B, \tag{5.2}$$

For the examples shown in Figure 5.1, $\alpha(t) = 1$ at any instance t. Given that B is much smaller than M, we can compute the average amount of synchronization at a memory bank as

$$S_b = S(t)/B \ll M, \tag{5.3}$$

We will use Equations 5.1, 5.2, and 5.3 in the design of SSB in the next section.

5.5 Design of Synchronization State Buffer

In this section we present the design of SSB . Recall that the design of SSB is motivated by the key observation that at any instance during the parallel execution only a small fraction of synchronized memory locations is actively in usage. SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of frequently synchronized data units to support and accelerate word-level fine-grain synchronization. We will first highlight the challenges that one faces in designing a solution for fine-grain synchronization.

SSB can be used to enforce both mutual exclusion and read-after-write data dependencies between a large number of threads. In the case of mutual exclusion, SSB allows each memory word to be individually locked with minimal overhead. SSB supports various locks: read lock (shared lock), write lock (exclusive lock), as well as recursive lock. For data synchronization that enforces the read-after-write dependencies between threads, SSB allows fine-grained low-overhead synchronized read and write operation on word in memory. SSB supports several modes of data synchronization: two single-writer-single-reader modes, and one single-writer-multiple-reader mode. By coordinating with the software, SSB efficiently facilitates fine-grained synchronizations to help multithreading programs exploit fine-grained parallelism inherent in applications. In this section we will discuss the various design parameters of SSB.

5.5.1 Buffer Size

The first design parameter is the number of entries E_b in an SSB for a memory bank b. The number of entries E_b is related to the size of memory bank M_b as follows: $E_b \leq M_b$. Now if $E_b = M_b$, SSB design is equivalent to tagging every memory location. In SSB we want to avoid tagging all memory location, and therefore we want:

$$E_b \ll M_b \tag{5.4}$$

From Equations 5.1, 5.2, and 5.3 we know that if an application can take advantage of the architectural design objective of Equation 5.4, then the following is the design requirement for the size of the buffer:

$$E_b \geq S_b \tag{5.5}$$

Let us generalize the above relation as follows:

$$E_b \simeq \beta \times S_b \tag{5.6}$$

where β is a factor that relates the amount of synchronization in an application to the hardware resource limitation. If $\beta \geq 1$ then SSB is cost effective, and if $\beta < 1$ then the performance of SSB is affected since the buffer will fill up and we have to fall back to software mechanism for synchronization. Given a particular buffer size, a compiler can optimize an application so as to reduce the amount of synchronization in the application. In practice, architects can determine the number of entries, and the level of set associativity of an SSB according to the class of applications to be supported, the transistor budget, the power consumption requirements, and other design factors.

Past research has indicated that fine-grain parallelism unleashed by some dataflow models that use the I-structure like fine-grain synchronization [20] far exceeded the capacity of a given hardware architecture to effectively exploit the parallelism [44]. Therefore, researchers have worked on solutions that somehow "throttle" the parallelism during program execution. In this dissertation, given a large number of processing cores and limited per core on-chip memory supported by underlying multi-core chip, we are using a thread model where the number of active threads is always limited by the number of available hardware thread units, which avoids the excessive parallelism in one dimension. Using SSB with limited size we throttle the parallelism in another dimension, and therefore the amount of parallelism that can exploited by active synchronization events is also limited. Our experimental results demonstrated sufficient thread-level parallelism can be effectively "throttled" (or regulated) using SSB of small size.

5.5.2 Hardware Cost

The SSB on the memory controller of each memory bank functions as a look-up table. Given the small size of each SSB, the single-cycle lookup function can be easily implemented with common hardware technology and modest cost. Another merit of SSB is its de-centralized and distributed nature, because of the independence of each SSB . Therefore, the hardware cost for implementing SSB increases only linearly proportional to the number of on-chip processing cores and memory banks, and the complexity of hardware logic remains the same for each SSB. In other words, SSB is a scalable fine-grain synchronization solution for large-scale multi-core chips.

5.5.3 Structure of SSB

state (4−bits)	counter (8−bits)	thread id	address

Figure 5.4: One SSB Entry

The overall structure of an SSB entry is shown in 5.4. Each SSB entry consists of four parts: (1) address field that is used to determine a unique location in a memory bank, (2) thread identifier, whose size is $\lceil log(T) \rceil$, where T is the number of non-preemptive threads supported by the underlying multi-core architecture, (3) an 8-bits counter, and (4) a 4-bits field that can support up-to 16 different synchronization modes. The address bits are used as a *key* to search the buffer and locate the entry of the synchronized location. The remaining three fields forms the synchronization state for that memory location. Since we assume a non-preemptive thread execution model, the “thread id” can be used to identify a processing core as well as a unique software thread running on it. The use of the counter field depends on the type of synchronization operation that is performed, which we will explain in the next section. Table 5.1 shows different synchronization modes that we support in our current design. An entry in SSB is allocated and released according to its state and the function of an SSB instruction operating on it.

An SSB instruction is treated the same way as other memory instructions by the on-chip interconnection network. All memory instructions, including SSB instructions are handled in a FIFO manner when arrive at a particular memory bank through the on-chip interconnection network.

Table 5.1: SSB State Bits

State Bits	Function	Description
0x0000	WLOCK	Write Lock
0x0001	RLOCK	Read Lock
0x0010	WRLOCK	Write-Recursive Lock
0x0011	SR1	Single-Writer-Single-Reader Mode 1
0x0100	SR2	Single-Writer-Single-Reader Mode 2
0x0101	MRF	Single-Writer-Multiple-Readers Full Mode
0x0110	MRL	Single-Writer-Multiple-Readers Lock Mode
0x0111	MRQ	Single-Writer-Multiple-Readers Queue Mode
0x1000	MRQL	Single-Writer-Multiple-Readers Queue Lock Mode
0x1001	LLSC	Load-Linked and Store-Conditional Mode

5.5.4 Memory Efficient Synchronization

Using SSB fine-grain synchronization operation is memory efficient. First, since SSB maintains the states for the synchronized memory locations in hardware, there is no need to allocate corresponding software-managed synchronization variables, which cost extra memory. Second, with one memory transaction, an SSB instruction does not only perform the synchronization on the memory location, but also brings the datum to the processor upon success. Therefore, compared to ordinary load operation, SSB synchronization operation only adds negligible overhead and saves the number of memory transactions needed.

5.6 An Architectural Model for SSB

In this section we present an architecture model for SSB , that consists of description of the various SSB states, the state transitions, and the corresponding SSB instructions.

5.6.1 Support for Fine-Grain Locking

SSB associates locking functions with memory locations dynamically. When a memory location needs to be accessed exclusively, the lock operation is issued with the address of this location. In the SSB of the corresponding memory bank, an entry for this address, if not exists, is allocated to monitor the state of the memory location. If an entry already exists, the state might be changed according to the function of the operation. The return value of the operation informs the state to the software, which then proceeds accordingly. Since an SSB instruction takes the address of a memory location to perform the locking operation, it does not require any pre-allocated synchronization variable. As a result, SSB is able to smoothly and efficiently handle the cases where the precise synchronization point cannot be resolved statically at compile time.

5.6.1.1 Implementation of Fine-Grain Lock

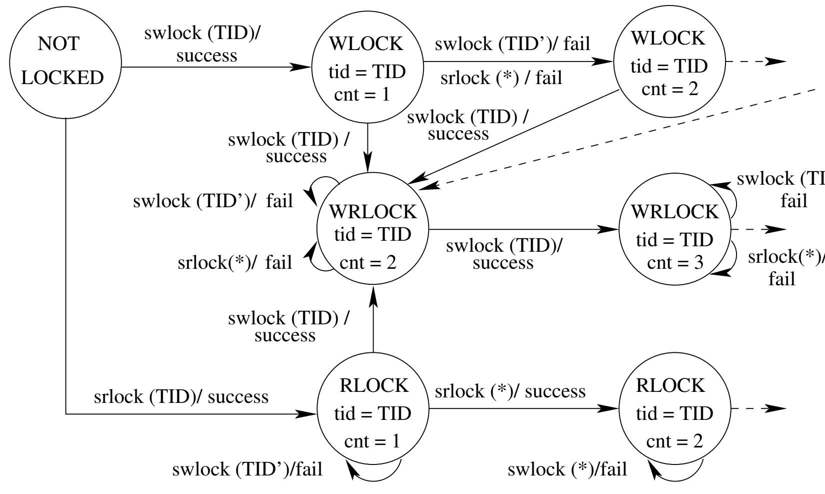
SSB provides following operations to perform the lock/unlock operations:

```
(RT, Value) = swlock_l(MemAddr);
/* swlock_l: acquire write lock for location MemAddr */
/*           and load the content                    */
/* MemAddr: the address of the memory location       */
/* RT: return value, success or failure              */
/* Value: the content of the memory location         */

(RT, Value) = srlock_l(MemAddr);
/* srlock_l: acquire read lock for location MemAddr */
/*           and load the content                   */
/* MemAddr: the address of the memory location      */
/* RT: return value, success or failure             */
/* Value: the content of the memory location        */

sunlock(MemAddr);
/* sunlock: release the lock for location MemAddr   */
/* MemAddr: the address of the memory location      */

RT = sunlock_r(MemAddr);
/* sunlock_r: release the lock for location MemAddr */
/* MemAddr: the address of the memory location      */
/* RT: return value, success or failure              */
```

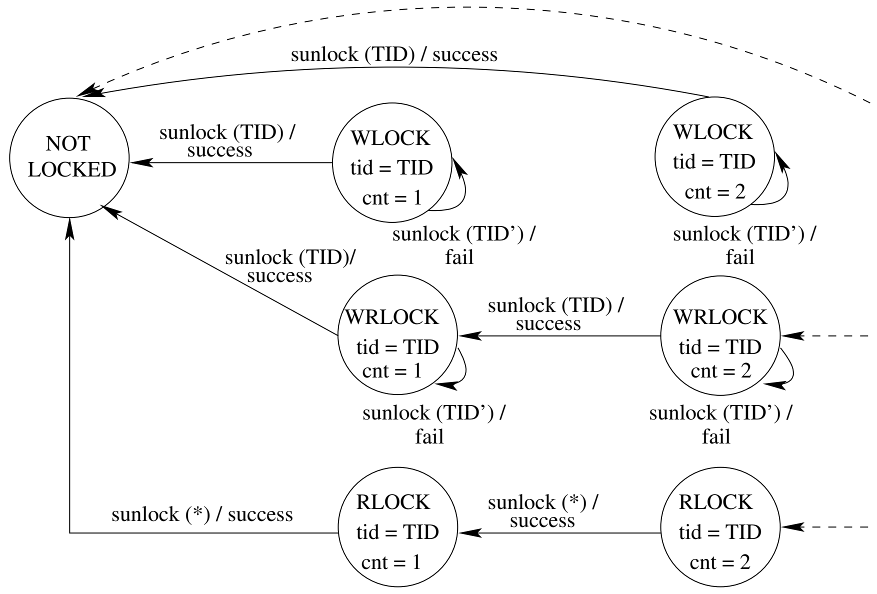
The swlock_l and srlock_l acquire the *write lock* and the *read lock* for the memory location MemAddr respectively. Upon success, they also load the content of the memory location to Value. sunlock releases the lock previously acquired. Figure 5.5 illustrates how the lock/unlock operations interact with the SSB hardware.

As shown in Figure 5.5(a), swlock_l acquires the *write lock* for memory location MemAddr. If there is no record for this location in SSB, which means it is not locked by any other thread, an entry for this location is allocated, and the state is set to WLOCK. Before this location is unlocked by the owner, write/read lock acquisition from other thread will fail, and cause the "counter (cnt)" incremented by 1. The current value of "cnt" is returned to the thread to indicate the failure. Therefore, in WLOCK mode, the return value accurately reflects the status of runtime lock contention on the memory location, i.e., how "hot" it is. Software may take advantage of this information to implement a *contention manager*, such as a backoff policy. SSB also supports recursive (or nested) lock. A thread can repeatedly acquire the write lock it already owns. If a thread is the only owner of the read lock, it can upgrade the lock to a write lock. In both cases, the state is set to WRLOCK, and the "cnt" records the number of the nested recursive locks. The software is required to perform paired lock/unlock operations, which guarantees the number of lock and unlock operations to be equal.

srlock_l acquires *read lock* for memory location MemAddr. Multiple threads can own the same read lock at the same time. The first successful acquisition allocates an entry in SSB, and sets the state to RLOCK. The "cnt" records the number of successful acquisitions. As described before, when "cnt" is equal to 1, a write lock acquisition from the same thread is able to upgrade the lock to a WRLOCK. Except for this special case, all the write lock acquisitions will fail. The behavior of sunlock operation is shown in Figure 5.5(b). When a lock is finally released, the corresponding entry in SSB will be



(a) states transition caused by swlock_l and srlock_l operations



(b) states transition caused by sunlock operation

A circle represents the state of a memory location monitored by SSB . The edge shows the transition between two states. Near the transition edge, the transition condition is described by a pair of text connected by a "/" symbol. The left side of "/" shows the operation performed to cause the transition; the right side of "/" indicates the return result of the operation. TID *in the parentheses suggests that the operation is issued by thread* TID. TID' *means a thread other than thread* TID. *The symbol "$*$" in the parentheses means that it can be "any thread".*

Figure 5.5: State transition diagram of SSB lock/unlock operations.

freed for reuse. It is worth noting that sunlock does not return the “success”/“fail” result to software. If a sunlock fails, an exception is raised. sunlock_r, on the other hand, returns the “success”/“fail” result to software. If a sunlock_r fails, it just return “fail” without generating any exception.

5.6.2 Fine-Grain Data Synchronization

SSB can help the programmer to exploit data-level parallelism by allowing the program to perform synchronized read and write at the word-level in memory. SSB provides a set of instructions to support fine-grained data synchronization that can enforce data dependencies between concurrent threads.

In the current design, two different types of data synchronization are supported: single-writer-single-reader, and single-writer-multiple-reader data synchronization.

5.6.2.1 Single-Writer-Single-Reader (SWSR) Data Synchronization

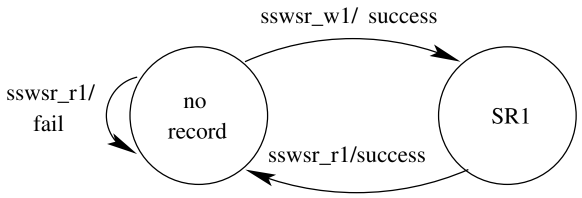
The single-writer-single-reader (SWSR) synchronization enforces ordering between a thread that produces the data and another thread that consumes the data. The following are the interfaces provided by SSB :

```
RT = sswrsr_w1(MemAddr, Value);
/* sswrsr_w1: SWSR synchronized write mode 1    */
/* MemAddr: the address of the memory location  */
/* Value: the Value to be written to MemAddr    */
/* RT: return value, success or failure         */

(RT, Value) = sswrsr_r1(MemAddr);
/* sswrsr_r1: SWSR synchronized read mode 1     */
/* MemAddr: the address of the memory location  */
/* RT: return value, success or failure         */
/* Value: the content of the memory location    */

RT = sswrsr_w2(MemAddr, Value);
/* sswsr_w2: SWSR synchronized write mode 2     */
/* MemAddr: the address of the memory location  */
/* Value: the Value to be written to MemAddr    */
/* RT: return value, success, failure or        */
/*     reader's thread id                       */

 (RT, Value) = sswrsr_r2(MemAddr);
/* sswsr_r2: SWSR synchronized read mode 2      */
/* MemAddr: the address of the memory location  */
/* RT: return value, success, failure, or wait  */
/* Value: the content of the memory location    */
```

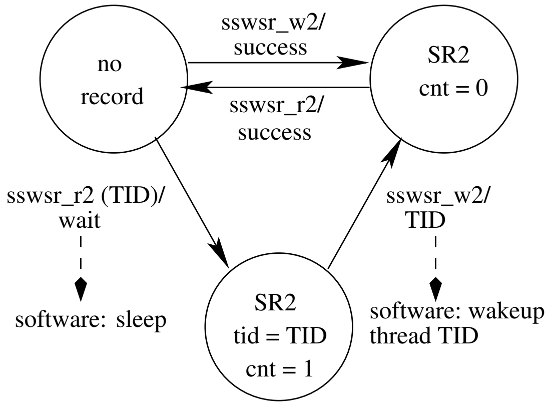
As shown in Figure 5.6(a), the sswsr_w1 and sswsr_r1 can coordinate with software to support a busy-wait approach. If the writer has not performed sswsr_w1 to the memory location addressed by MemAddr yet, the sswsr_r1 performed by the reader returns a failure. The reader needs to try again with sswsr_r1 afterwards. The reader can get the data only after the sswsr_w1 is finally performed, which allocates an entry in the SSB, sets the state to SR1, and writes the Value into MemAddr. When the sswsr_r1 is successfully executed, the entry in SSB is released, and the content of MemAddr is loaded for the reader.

Figure 5.7 shows an example using sswsr_w1 and **sswsr_r1** to synchronize between two threads in C like pseudo code. A software busy-wait approach is applied.

Other than the busy-wait approach, a blocking strategy can be implemented with the sswsr_w2 and sswsr_r2 operations, and the instruction-level sleep/wakeup support of the underlying multi-core architecture. As illustrated by Figure 5.6(b), if the reader performs sswsr_r2 before the sswsr_w2 from the writer, an entry will be allocated in SSB, the state is set to SR2, and the counter is set to 1 to represent that the data is not available yet. The thread id of the reader is also recorded. When the reader finds out that the return value is "wait", it issue a *sleep* instruction to suspend the execution and go to sleep. Later, the sswsr_w2 instruction issued by the writer will write the Value into MemAddr, and set the counter to 0 to indicate the availability of the data. The instruction also returns the thread id (TID) of the reader to the writer. Then the writer issues a



(a) Mode 1: a busy-wait approach



(b) Mode 2: a sleep-wakeup approach

A circle represents the state of a memory location monitored by SSB . The edge shows the transition between two states. Near the transition edge, the transition condition is described by a pair of text connected by a "/" symbol. The left side of "/" shows the operation performed to cause the transition; the right side of "/" indicates the return result of the operation. TID *in the parentheses suggests that the operation is issued by thread* TID*. "software:" means the operation that described by following text is performed by software.*

Figure 5.6: State transition diagram of SSB Single-Writer-Single-Reader operations.

```
/* "tmp" is a local variable */
tmp = ...
/* write "tmp" to shared variable "data",
   and mark it as available.
*/
while(sswsr_w1(&data, tmp) != SUCCESS)
```

(a) Writer

```
/* read from shared variable "data" to
   local variable "tmp", if available
*/

/* a busy-wait loop */
while(1){
   (RT, tmp) = sswsr_r1(&data);
   if(RT == SUCCESS)
     break;
}
```

(b) Reader

Figure 5.7: Example: Using sswsr_w1 and sswsr_r1 to enforce data dependence between a pair of write and read

hardware interrupt to wake up the reader. After having been awakened, the reader can now retrieve the value by sswsr_r2 and free the corresponding entry in the SSB.

Figure 5.8 shows an example using sswsr_w2 and sswsr_r2 to synchronize between two threads in pseudo code. Compared to the busy-wait approach shown in Figure 5.7, a blocking-wait approach is employed here. Therefore, by using sswsr_w2 and sswsr_r2, software can avoid generating unnecessary memory traffic.

5.6.2.2 Single-Writer-Multiple-Reader (SWMR) Data Synchronization

The single-writer-multiple-reader (SWMR) synchronization enforces ordering between a thread that produces the data and a number of other threads that consume the data. The following are the interfaces:

```
tmp = ... /* "tmp" is a local variable */
/* write "tmp" to shared variable "data",
   and mark it as available. */
while(1){
  RT = sswsr_w2(&data, tmp);
  if(RT == FAIL)
   continue;
  else if(th_id == SUCCESS) break; /* no waiter */
  else{ /* there is a waiter, RT is a thread id */
   wakeup_thread(RT);    /* wakeup the thread */
   break;
  }
}
```

(a) Writer

```
/* read from shared variable "data" to
  local variable "tmp", if available */
while(1){
  (RT, tmp) = sswsr_r2(&data);
  if(RT == WAIT){  /* data not available yet */
    /* sleep until waken up by writer */
    sleep();
  }
  else if(RT == SUCCESS)
    break;
}
```

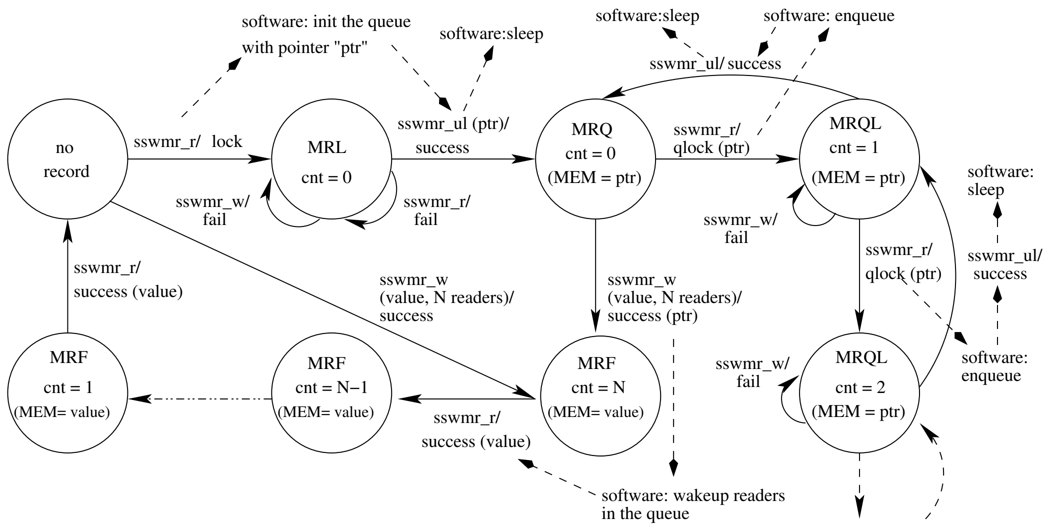
(b) Reader

Figure 5.8: Example: Using sswsr_w2 and sswsr_r2 to enforce data dependence between a pair of writer and reader

```
RT = sswmr_w(MemAddr, Value, NumOfReaders);
/* sswmr_w: SWMR synchronized write             */
/* MemAddr: the address of the memory location   */
/* Value: the Value to be written to MemAddr     */
/* NumofReaders: the number of readers           */
/* RT: return value, success, failure,           */
/*     or the pointer the wait queue             */

(RT, Value) = sswmr_r(MemAddr);
/* sswmr_r: SWMR synchronized read                */
/* MemAddr: the address of the memory location    */
/* RT: return value, success, failure, lock mode, */
/* or qlock mode                                  */
/* Value: the content of the memory location      */
/* upon success, or the pointer to the queue      */
/* if the RT is lock mode or queue mode           */

sswmr_ul(MemAddr, QueuePtr);
/* sswmr_ul: SWMR queue unlock                    */
/* MemAddr: the address of the memory location    */
/* QueuePtr: the pointer to the wait queue        */
```

Figure 5.9 shows how SSB SWMR operations interact with software to perform the data synchronization between one writer and multiple readers. In the ideal case, the sswmr_w write operation is executed before all the read operations. As a result, an entry is allocated in the SSB, the state is set to MRF (full mode), "cnt" (counter) is initialized to N, which represents the number of readers, and Value is written into the memory location addressed by MemAddr. All the following sswmr_r operations read the value from the memory and decrement the "cnt" by 1. When all the reads finish and the "cnt" reaches 0, the corresponding entry in SSB is freed.

However, it is possible that some readers issue the sswmr_r read operations before the write. The first such sswmr_r instruction allocates an entry in the SSB and sets the state to MRL (lock mode). Then the thread that issues this read will initialize a wait queue, put itself into the queue, and issue a sswmr_ul instruction with the pointer to the tail of the wait queue as a parameter. The sswmr_ul stores the pointer into the memory location, and switches the state to MRQ (queue mode). The following sswmr_r operations issued by other threads will get this pointer, with which a thread can enqueue itself. As shown in Figure 5.9, if one or more threads are performing the enqueue operation, the state of the SSB entry is MRQL (queue lock mode), which prevents the write from happening. After the enqueue operation, the thread issues a sswmr_ul operation and goes to sleep. When the state of the SSB entry is switched back to MRQ and a sswmr_w operation arrives, the write can be performed, and the state is changed to MRF. In this case, the queue pointer is returned to the writer thread, which then wakes up all the threads in the queue. Since



A circle represents the state of an memory location monitored by SSB . The "MEM =" in the parentheses indicates the content of the memory location that is monitored by this SSB entry. The edge shows the transition between two states. Near the transition edge, the transition condition is described by a pair of text connected by a "/" symbol. The left side of "/" shows the operation performed to cause the transition, with its parameters in parentheses; the right side of "/" indicates the return result of the operation, with an additional return value in parentheses. "software:" means the operation that described by following text is performed by software.

Figure 5.9: State transition diagram of SSB Single-Writer-Multiple-Reader Operations.

the state of the entry is already MRF, all the awakened threads as well as other threads can now read data from the memory.

Figure 5.10 and 5.11 demonstrate an example using **sswmr_w1**, **sswmr_r1**, and **sswmr_ul1** to synchronize between a single writer and multiple readers in pseudo code.

5.6.3 Discussions

5.6.3.1 Handle Hardware Resource Limitation

Since the (hardware) SSB is a fixed size buffer, for some applications, it can become full. In such situation we trap to a software solution. Each hardware SSB (at a memory bank), called HSSB, has its associated software SSB, called SSSB. An SSSB is

```
typedef struct qnode{
   unsigned th_id;
   struct qnode *pred;
}qnode;

typedef qnode qQueue;

void queue_init(void * addr){
   qnode I;
   qQueue *Tail;
   I.th_id = thread_id();
   I.pred = NULL;
   Tail = &I;
   /* put the address of
      "Tail" into memory
      location referenced
      by "addr" */
   sswmr_ul1(addr,&Tail);
   /* sleep until waken up
      by another thread
   */
   sleep();
   return;
}
```

```
void enqueue(tQueue **pTail,
             void * addr){
  qnode *pred;
  qnode I;
  I.th_id = thread_id();
  /* use atomic operation
     fetch_and_store */
  pred =
   fetch_and_store(pTail, &I);
  if(pred != NULL)
    I.pred = pred;
  sswmr_ul1(addr,pTail);
  /* sleep until waken up */
  sleep();
  return;
}

void wakeup_all(tQueue *Tail){
  qQueue *ptr;
  ptr = Tail;
  while(ptr != NULL){
    wakeup_thread(ptr->th_id);
    ptr = ptr->pred;
  }
  return;
}
```

Figure 5.10: Example: Using sswmr_w1, sswmr_r1, and sswmr_ul1 to enforce data dependence between a single writer and multiple readers: Queue data structure and functions

an extension to its corresponding HSSB, and to simplify our discussion we assume them to be fully associative. Each HSSB contains two bits, FBIT and SBIT. FBIT is set to ON automatically by hardware whenever the HSSB becomes full, otherwise it is OFF. The SBIT indicates whether there are software maintained entries in the SSSB. When the kernel starts, it initializes all the SSSBs. An HSSB also has a register, called SREG that is initialized during boot time by the kernel, holds a pointer to its corresponding SSSB and an associated software lock. The SSSB software structure is common across all applications on the system. An entry in the SSSB has the same structure as the HSSB entries.

```
/* "tmp" is the local,
   "data" is the shared,
   try to write "tmp"
   into "&data". */
tmp = ...
if(thread_id() == 0){
  while(1){
    num_readers =
        num_threads() - 1;
    RT = sswmr_w1(&data,
                  tmp,
                  num_readers,
                 );
    if(RT == FAIL) /* fail */
      continue;
    else if(ret == SUCCESS){
     /* success, no
      waiting reader */
      break;
    }
    else{ /* RT contains a
             pointer to the
             waiting queue */
      pTail = (tQueue **)RT;
      wakeup_all(pTail);
      break;
    }
}
```

(a) Writer

```
if(thread_id != 0){
  /* load shared "data" into
     local variable "tmp" */
   while(1){
     (RT,tmp) = sswmr_r1(&data);
     if(RT == SUCCESS)
       break; /* success, data
                 is already in
                 tmp */
     if(RT == FAIL)
       continue; /* fail */
     if(RT == QLOCK){
       /* tmp is now the
          pointer
       */
       enqueue((tQueue **)tmp);
       continue;
     }
     else if(RT == LOCK){
       /* setup the waiting
          queue
       */
       queue_init(&data);
       continue;
     }
   }
}
```

(b) Reader

Figure 5.11: Example: Using sswmr_w1, sswmr_r1, and sswmr_ul1 to enforce data dependence between a single writer and multiple readers: writer and reader

We assume that instructions that arrive at a memory bank are processed in an FIFO order. When an SSB instruction reaches and searches the HSSB, there are following possible cases:

Matching entry in HSSB?	FBIT	SBIT	Case
Yes	Any	Any	1: HW only solution
No	OFF	OFF	2: HSSB is not full, HW only solution
No	ON	OFF	3: HSSB is full, set SBIT on, trap to SW
No	Any	ON	4: Entries in SSSB, trap to SW

Accordingly, the steps that are taken by the memory controller on the memory bank is shown in Figure 5.12.

```
1.   Search the HSSB
2.   if Find a matching entry
3.       Perform normal operations
4.   else
5.       if SBIT is OFF
6.           if FBIT is OFF
7.               Create an entry in HSSB, perform operations on it
8.           else
9.               Set SBIT to ON, a software trap is raised
10.      else
11.          A software trap is raised
```

Figure 5.12: Operations of Memory Controller

The raised trap is handled using a software handler, to which the pointer in the SREG, along with the opcode and operands of the SSB instruction, are supplied as parameters. The handler is executed by the thread that issued the SSB instruction. The software lock associated with each SSSB has to be acquired by the thread before it executes the handler, thus no other threads can change the states of an SSSB simultaneously. It is possible that the state of the corresponding HSSB has changed between the duration of the raise of the trap and the acquisition of the lock. Therefore, the software handler will deal with following cases:

SBIT	Matching Entry in HSSB?	FBIT	Case
OFF	No need to check	Any	1: SSSB is empty, fall back to HW
ON	Yes	Any	2: Fall back to HW
ON	No	OFF	3: Attempt to promote the entry to HW
ON	No	ON	4: SW only solution

To check the state of SBIT, FBIT, and search the HSSB for matching entry, special instructions are used. When the thread gets the lock and begins to execute the handler, it first checks the SBIT. If SBIT is OFF, the SSSB is empty due to the operation of another thread who owned the lock previously. As suggested in case 1, the handler releases the lock and re-issues the SSB instruction. If SBIT is on, the handler issues an instruction to search the HSSB. If a matching entry is found, it is case 2, and the handler takes the same action as case 1. Otherwise, it performs the operations on SSSB, then check the FBIT. If the FBIT is OFF, which is case 3, the handler attempts to flush the entry to the HSSB, also with an instruction. If successful, the handler removes the software entry from the SSSB. The remaining step of case 3 and case 4 are the same. If the SSSB becomes empty, the handler sets the SBIT to OFF, releases the lock, and returns. The steps performed by the handler are summarized in Figure 5.13.

The software mechanism will slow down the requested synchronization operation. However, it is expected that a small SSB is normally sufficient for most multithreading programs. As we will show in Section 6.6, for many benchmarks, only one has a small percentage of synchronization operations that encounter the “full” situation.

5.6.3.2 Support Load Linked (LL), and Store Conditional (SC) Operations

Normally by extending cache protocols, current mainstream processor architectures support the Load Linked (LL), and Store Conditional (SC) instructions as atomic primitives to implement other atomic operations. However, “none allow nesting or interleaving of LL/SC pairs, and most prohibit any memory access between LL and SC” [122]. For multi-core architectures, which employ the scratchpad memory (or other format of

```
1.  Acquire the corresponding software lock
2.  Check the SBIT using a special instruction
3.  if SBIT is ON
4.      Search the HSSB using a special instruction
5.      if Find a matching entry
6.          Release the lock, re-issue the SSB instruction
7.      else
8.          Search the SSSB
9.              if Find a matching entry
10.                 Operate on the entry
11.             else
12.                 Create an entry in SSSB
13.                 Operate on the entry
14.         if The entry is not freed in SSSB
15.             Check the FBIT using a special instruction
16.             if FBIT is OFF
17.                 Flush the entry to HSSB using a special instruction
18.                 if Success
19.                     Remove the entry from SSSB
20.         if SSSB is empty
21.             Set the SBIT to OFF using a special instruction
22.     Release the lock
23. else
24.     Release the lock, re-issue the SSB instruction
```

Figure 5.13: Operations of the Software Handler

local memory) instead of data cache, it is apparent that it is straightforward to support the LL/SC instructions using SSB to monitor the states of memory location accessed by LL/SC. For the synchronization operations introduced previously, the SSB only interacts with a group of SSB instructions, it does not need to handle normal load and store. However, to support the semantics of LL/SC, the normal store operations need to be monitored by SSB as well. By implementing LL/SC with SSB, there is also no limitation in nesting or interleaving LL/SC pairs, or other memory accesses between LL and SC.

5.7 Related Work

Our SSB design provides an illusion that the entire memory is tagged at word-level, and therefore can be considered as "virtual tagged memory" design. The major difference between SSB and the classical tagged memory (e.g. full/empty bits) in HEP [147], Tera [11], MDP [47], Sparcle [4], M-Machine [98], the MT processor in Eldorado [57], and other machines, has been explained in Section 5.3. I-structure [20] memory sytem employed in some dataflow model based architectures [20, 97] exploits similar design as full/empty bits based memory system. Tagging each word of the entire memory requires modification to off-the-shelf SRAM or DRAM technologies and introduces significant storage cost. Because of such cost, the number of state bits that can be tagged to a word has to be small, which can only be used to implement limited synchronization functionalities. Because of the small storage cost, SSB can afford to form much larger states in each entry, thus can potentially support more synchronization functionality.

Hardware mechanisms such as hardware queue based QOLB [93], MAOs on SGI Origin [107], lock box [155] for SMT processor, SoC lock cache [6], AMO [166] and others, target to improve the efficiency of lock primitives. Unlike SSB or tagged memory, they are not designed to provide architectural support for word-level fine-grain synchronization in memory. The M-Machine [98] also allows fast synchronization between three on-chip processors through register-register communication. Sampson et. al. [144] proposed barrier filters, a hardware mechanism for enabling fast barrier synchronization on multi-core chips.

Transactional memory (TM) using non-blocking synchronization is proposed as a replacement to lock-based synchronization for multithreading programming [84, 140, 141, 116]. Most hardware TM designs need to extend and modify the existing cache coherence protocols and speculative execution techniques. Our current SSB design relies on blocking synchronization mechanism, and it will be interesting to see how to explore non-blocking synchronization in an SSB-like design.

Finally, various loop optimization techniques have been developed to minimize the amount of fine-grain synchronization for parallelized do-across loops [108, 124, 103, 37, 129, 138]. Those techniques can be combined with SSB-based hardware support to further improve the resulting code, especially when the synchronization resource requirements exceeds the number of SSB entries provided.

Chapter 6

EVALUATION OF SSB

Our objective in this chapter is to illustrate the characteristics of SSB and verify the efficiency and effectiveness of SSB. We also compare SSB with other synchronization mechanisms. We explore the characteristics of SSB in the context of the IBM 160-core Cyclops-64 (C64) chip architecture [52, 53], which represents a class of large-scale multi-core architectures that we discussed in Section 5.4. For more detail about the C64 architecture, please refer to Section 2.3.4.

6.1 Implementation of SSB on Simulator

We implemented the proposed SSB as an extension to the C64 ISA using FAST simulator for the C64 large-scale multi-core architecture [49]. We model the C64 chip design with the 160 cores, the three-level memory hierarchy, and the crossbar interconnection network. The simulator takes into account the main sources of pipeline delays and stalls in the processor architecture, as well as models all details in the memory hierarchy, including contention in memory and the crossbar network. The SSB extension to C64 is implemented in the simulator. SSB instructions that require return (data) values have the same latency as a load instruction, otherwise the latency is same as a store instruction. For our experiments we used a 16-entry SSB for each on-chip memory bank, and used a 1,024-entry SSB for each off-chip memory bank, both of which are 8-way set associative.

6.2 Selected Benchmarks

The set of selected benchmarks that we used for experiments are summarized in Table 6.1.

Table 6.1: Summary of Benchmarks Analyzed for SSB Behavior

Benchmark	Source	Description
Random Access	HPCC Benchmarks [137]	random updates of memory
Livermore Loop 13	Livermore Loops [56]	2-D particle-in-cell
Livermore Loop 14	Livermore Loops	1-D particle-in-cell
Loop $G1$	SSCA#2 [24]	graph problem
Ordered Integer Set	Common data structure	hash-table based
$K1$, $K2$, $K3$ $K4$, $K5$, $K6$	Kernel Loops from SPEC OMP [42]	DOAcross Loops with constant & positive dependence distances
1D Laplace Solver	scientific application kernel	partial differential equations
Livermore Loop 6	Livermore Loops	linear recurrence equations
2D Wavefront	scientific application kernel	2D wavefront computation

Benchmark	Data Set	Synchronization
Random Access	2^{17} 64-bit integers	write lock
Livermore Loop 13	4K doubles for h table, 512 iterations	write lock
Livermore Loop 14	4K doubles for rh table, 2,048 iterations	write lock
Loop $G1$	$n = 2^{13}$	write lock
Ordered Integer Set	25 buckets, average load 100	write/read lock
$K1$, $K2$, $K3$ $K4$, $K5$, $K6$	5000 iterations	SWSR data synch.
1D Laplace Solver	512,1024,2048,4096	SWSR data sync.
Livermore Loop 6	5K doubles	SWMR data sync.
2D Wavefront	$1K \times 1K$ doubles	SWSR data sync.

In the rest of the chapter we will compare SSB with the other synchronization mechanisms, and answer the following questions:

- What is the cost of a successful synchronization operation?
- How effective is SSB for fine-grain mutual exclusion synchronization?

- How effective is SSB for fine-grain data synchronization?
- How effective is SSB in exploiting fine-grain parallelism?

6.3 Cost of Successful Synchronization

Previous studies have shown that fine-grain synchronization results in successful synchronization in most cases [102, 165], and this is also true for SSB-based fine-grain synchronization (see Section 6.6). Therefore, it is important to ensure that the cost of a successful synchronization is very low.

6.3.1 Fine-grain lock

To measure the overhead of different synchronization mechanism, we wrote a simple loop that iterates 10,000 times and at each iteration a 64-bit integer value is loaded from on-chip SRAM, a simple arithmetic operation is performed on the value, and the result is stored back to the memory. A reference time is obtained by executing the loop sequentially without using any synchronization. Then the synchronization overhead is calculated by comparing the reference time with the execution time of the same code extended with synchronization operations. When using a test-and-set spin lock, a lock has to be acquired/released before/after accessing the memory location. A lock-free approach can be implemented using the *compare-and-swap (CAS)* instruction to commit the result into memory if the content of the memory location has not changed since the last load. SSB-based synchronization is similar to the spin lock in this case. The loop with synchronization is also executed on a single thread, thus all the synchronization operations (lock acquisition or CAS commitment) are successful. Figure 6.1 shows that SSB incurs the lowest cost among the three mechanisms. This can be attributed to the fact that an SSB instruction performs a successful synchronization and brings the datum to the processor in one memory transaction.

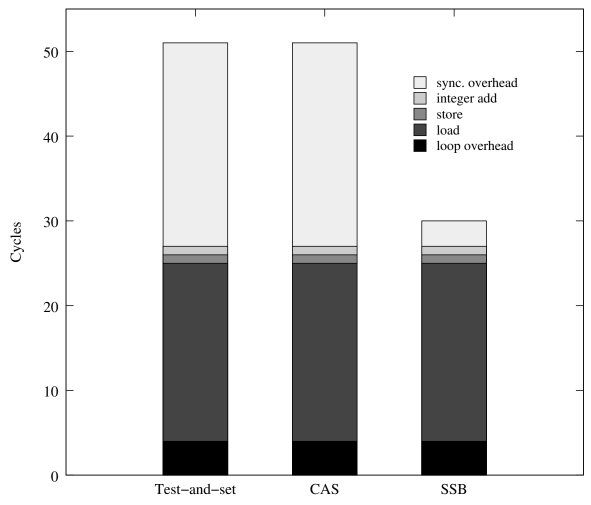


Figure 6.1: Overheads of Synchronization Mechanisms

6.3.2 Fine-grain data synchronization

To measure the overhead of data synchronization we used a simple loop. A reference time is obtained by executing the loop of 10,000 iterations with 2 threads. Each iteration contains a barrier operation. One thread performs a store before the barrier, and the other performs a load after the barrier. The overhead is computed by comparing this reference time with the execution time of the same code but replacing the store/load operation with SSB synchronized write/read operation. The barrier in the code guarantees the synchronized write happens before the synchronized read, which is always successful as a result.

Table 6.2: Overhead of successful SSB data synchronization operations

SSB Operations	Overhead (cycles)
sswsr_w1/sswsr_r1	22
sswsr_w2/sswsr_r2	24
sswmr_w/sswmr_r	26

As shown in Table 6.2, the overhead of SSB data synchronization operations are small when performed successfully. The major overhead comes from the difference between synchronized write and normal store instruction. It takes 1 cycle to issue a normal store instruction without introducing any data dependence. However, a data dependence is formed between the synchronized write instruction and the instruction that checks its return value (success, failure, etc.). Therefore, there is a latency similar to a load operation. One can hide this latency by issuing other independent instructions. Additional overhead comes from the code that checks and handles the return value of the synchronization operations.

6.4 Effectiveness of SSB for Fine-Grain Lock

In this section, we examine the effectiveness of SSB for fine-grain locking using four benchmarks, where a conventional synchronization mechanism can not easily exploit the available parallelism: Table Toy (also called Random Access) from the HPC Challenge benchmarks [137], two of the Livermore loops, and a hash-table based implementation of ordered integer set.

6.4.1 Random Access

As shown in Figure 5.1, the address of the memory location to be accessed is only known right before entering the critical section. In this case, if a conventional spin-lock is used, to enforce the mutual exclusion, the programmer/compiler normally assigns a single lock to the whole array, which however serializes the execution. One solution to exploit the parallelism is to allocate an array of locks with the exact the same size as $y[]$, so that a thread can acquire the corresponding lock in the array for a element of $y[]$ dynamically – once a thread determines the member of $y[]$ to be accessed at runtime, it can acquire the corresponding lock in the lock-array first. However, this *lock-array* approach doubles the memory usage. Using the SSB lock operations, one can simply provide the runtime calculated address as a parameter to the SSB lock interface to achieve the same effect as the lock-array approach without any overhead in memory usage.

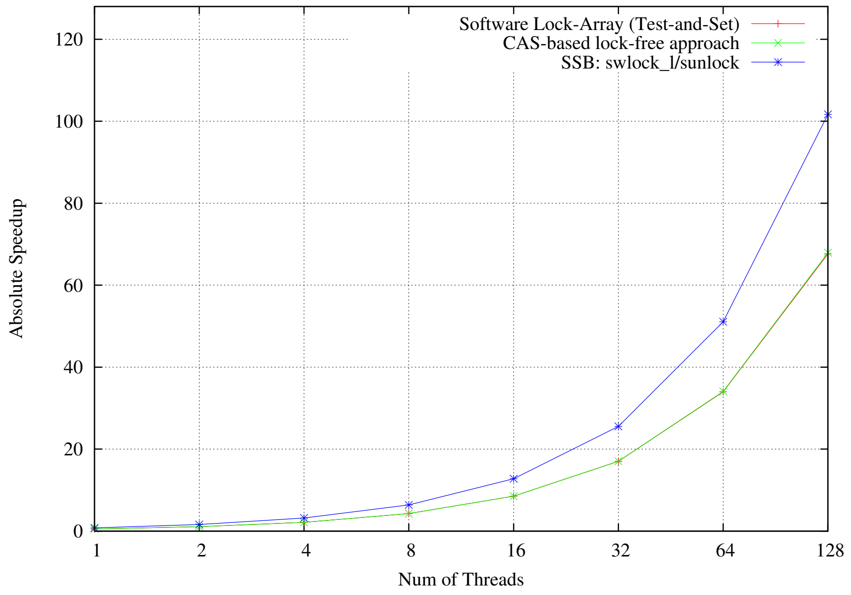


Figure 6.2: Absolute Speedup of Random Access Benchmark

Figure 6.2 compares three parallelization schemes of Random Access using different fine-grain synchronization mechanisms. The table is placed in on-chip SRAM. The software lock-array approach provides scalable performance, however, it incurs large memory usage overhead, which is not practical for real applications. The CAS-based lock-free approach achieves a similar speedup curve as the lock-array one. The SSB-based solution indicates the best performance by fully exploiting the fine-grain parallelism with low cost synchronization operations. When running on 128 threads, it yields an absolute speedup of 101, outperforming the other two approaches by 50.3% and 49.7% respectively without any extra memory usage.

6.4.2 Livermore Loop 13 and 14

Because of the cross-iteration dependencies (which cannot be determined statically), Livermore Loops 13 and 14 cannot be easily parallelized [156]. Within each iteration, a few elements of the array are updated. However, the calculation of the indices is

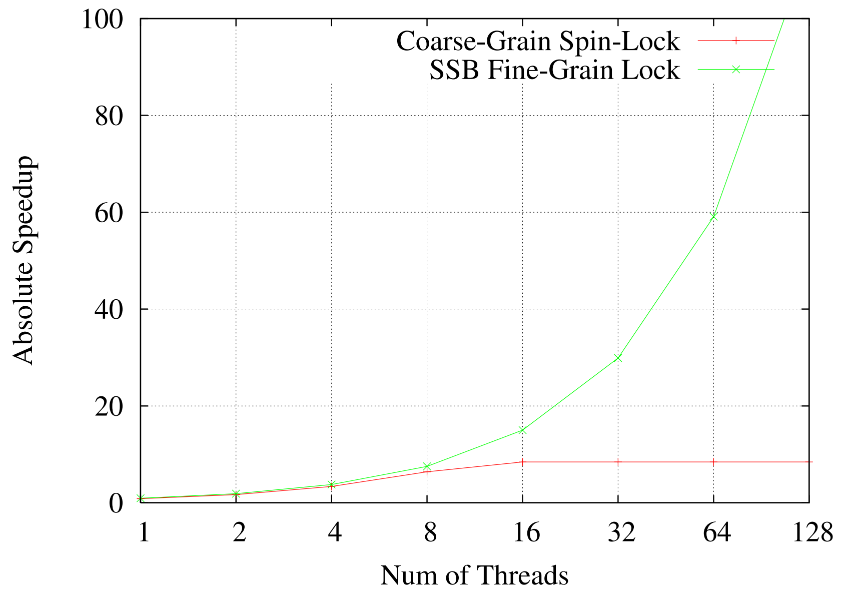


Figure 6.3: Absolute Speedup of Livermore Loop 13

unpredictable and data-dependent. Since it is not necessary to preserve the order of these updates, we use locks to guarantee mutual exclusion for updating elements of the array that can only be determined at runtime when running with multiple threads.

Figures 6.3 and 6.4 compare coarse-grain synchronization with SSB. The coarse-grain approach serializes the updates to the array using an MCS [118] spin-lock to ensure mutual exclusion. The fine-grain approach makes use of the SSB lock instructions to individually lock the locations to be updated. Therefore, the iterations that access different locations do not contend with each other. The SSB-based fine-grained synchronization exploits the inherent parallelism in the code without unnecessarily serializing the updates to non-conflicting locations of the arrays (see Figure 6.3 and 6.4). As a result, we achieve speedups of 114.3 and 72.4 on 128 threads for Loop 13 and Loop 14, respectively.

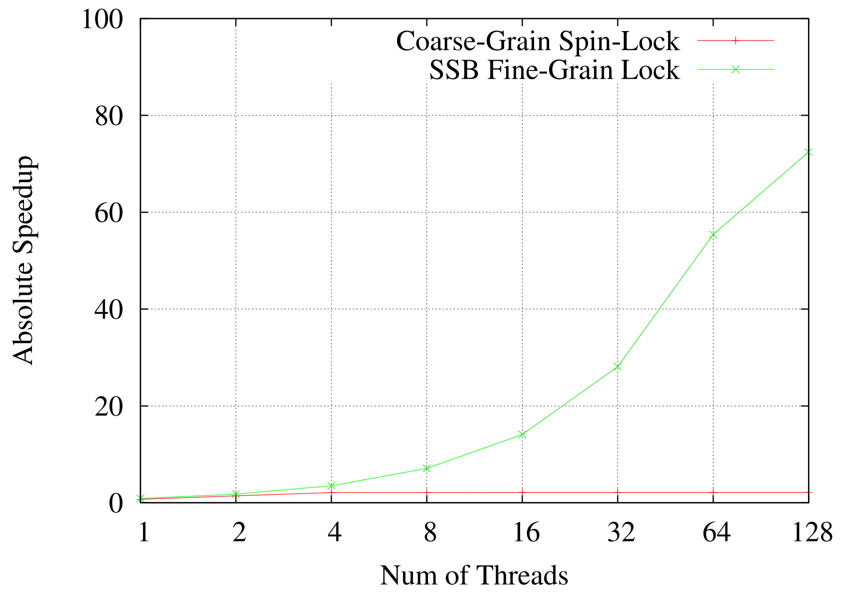


Figure 6.4: Absolute Speedup of Livermore Loop 14

6.4.3 A Kernel Loop from SSCA#2

The Scalable Synthetic Compact Applications Benchmark Suite 2 (SSCA#2) represents a graph theoretic problem which is representative of computations in the fields of national security, scientific computing, and computation biology [24]. A hallmark of the graph problem is the irregular memory accesses, which leads to poor data locality and statically unsolvable synchronization points.

Figure 6.5 shows a loop extracted from SSCA#2 version 1.1. Let us call this loop as Loop $G1$. The code is written using the Bader's SIMPLE library [23]. We briefly review the major characteristics of the code as follows:

- **Line 1:** Initialize an array of n locks. With data type defined in the SIMPLE library, an element in the $vLock$ array is actually a pthread mutex. The function **lock_init_arr** is performed in parallel. It is worth noting that n represents the number of vertices in the graph.

```
 1  lock_init_arr(&vLock, n, TH);
 2
 3  node_Barrier();
 4
 5  pardo(u, 0, n, 1) {
 6    for (j=G->outVertexIndex[u]; j<G->outVertexIndex[u+1]; j++) {
 7      v = G->outVertexList[j];
 8      if (!isEdgePresent_OutVertex(G, v, u)) {
 9        my_lock(&vLock[v]);
10        inDegree[v]++;
11        my_unlock(&vLock[v]);
12        impliedEdgeFlag[j] = 1;
13        inVertexListSize++;
14      }
15    }
16  }
17
18  node_Barrier();
19
20  lock_destroy_arr(&vLock, n, TH);
```

Figure 6.5: A Loop ($G1$) Extracted from SSCA#2

- **Line 5: pardo** is a do-all loop construct. It statically distributes iteration 0 to $n-1$ of Loop $G1$ to all threads. u is the iterator.

- **Line 9, and 11:** Using lock **vLock[v]**, **my_lock** and **my_unlock** function form a critical section for accessing **inDegree[v]** mutually exclusively. In SIMPLE library, **my_lock** and **my_unlock** are mapped to **pthread_mutex_lock** and **pthread_mutex_unlock** respectively.

- **Line 20:** Destroy the lock array **vLock**. The function **lock_destroy_arr** is performed in parallel.

In order to exploit the inherent parallelism in the code, fine-grain synchronization is required. The fine-grain synchronization approach taken in the loop shown in Figure 6.5 uses a software lock-array approach similar as the one we showed for Random Access benchmark. Given a graph problem, the number of vertices n is normally very large. Therefore, the allocation of array **vLock** costs a lot of memory space. For example, in

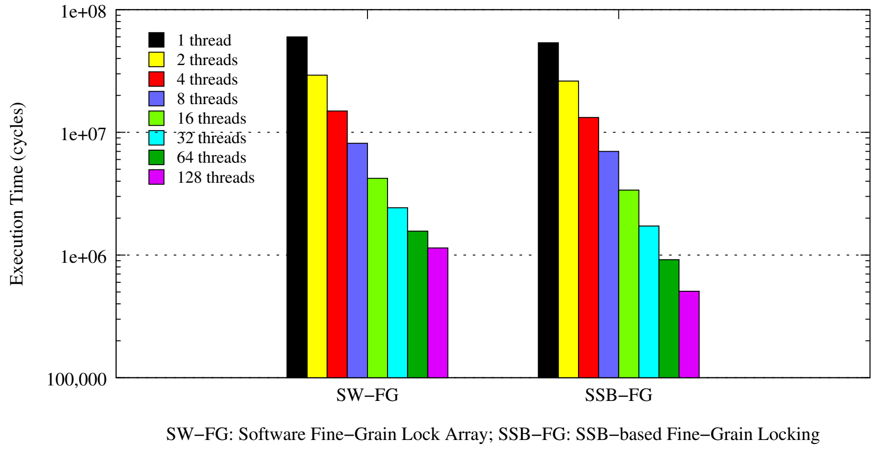
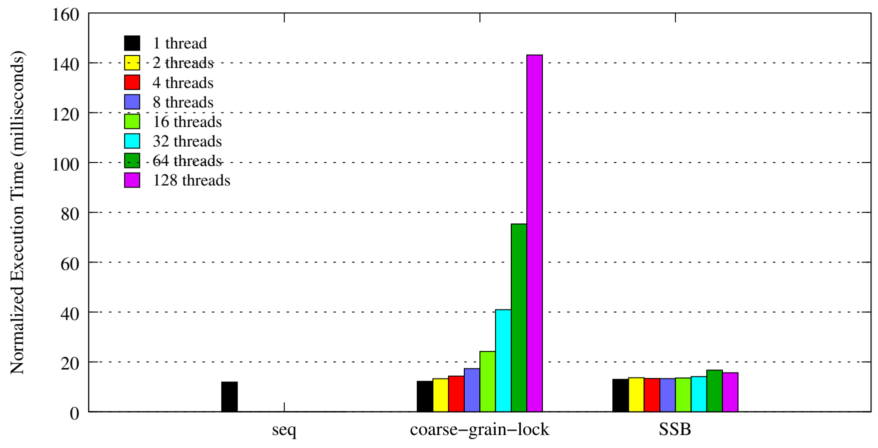


Figure 6.6: Execution Time of the Loop $G1$ Extracted from SSCA#2

our experiments, when we set $n = 2^{13}$, the size of **vLock** array is 64K bytes. Moreover, at runtime, the **if** condition at **line 8** is normally false. As a result, a large portion of the **vLock** is not actually used.

SSB-based fine-grain lock mechanism can avoid all the drawbacks of the software-based one. Using SSB , there is no need to allocate the **vLock** array, which saves memory. At runtime, given the address of a a particular element in the array $inDegree$, SSB lock/unlock instruction is used to ensure the mutual exclusion for accessing it. For this particular example, it is worthing noting that the operation $inDegree[v] + +$ can be completed atomically in memory with instruction **ADD_M** on C64. However, the set of in-memory atomic instructions provided by ISA can only cover limited data type and operations. SSB presents a general fine-grain synchronization mechanism with no limitation on data types and operations.

Given the low overhead of SSB operations, the SSB-based approach does not only avoid the memory cost for allocating the array of locks, but also improves the performance. Figure 6.6 compares the execution time of the SSB-based solution to the software-based one with $n = 2^{13}$. The execution time for the software-based version includes the



Y-axis: the normalized execution time by number of threads.

Figure 6.7: Concurrent Hash Table: SSB Fine-Grain Synchronization vs Coarse-Grain Synchronization

time spent on executing the loop, initialize, destroy the lock array. The SSB -based version does not need to allocate and free the lock array. From Figure 6.6, it is clear that the SSB-based version performs faster than the software-based one in all cases. When the number of threads increases to 128, the SSB-based one is 125% faster.

6.4.4 Hash Table Based Ordered Integer Sets

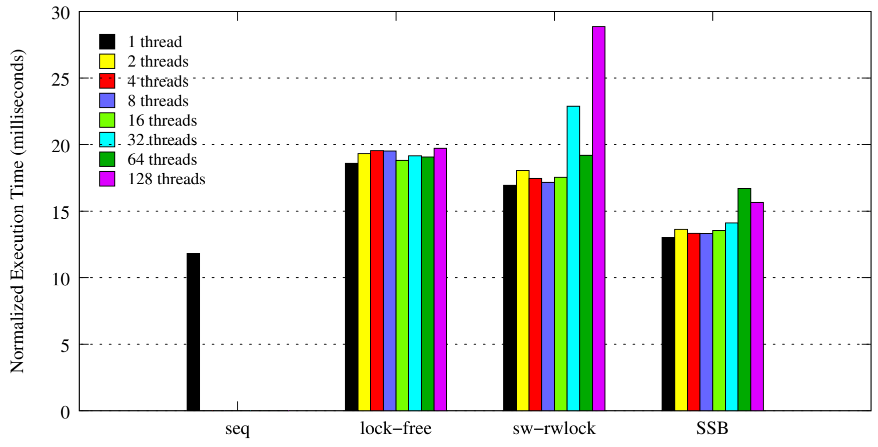
Hash table is a common data structure widely used in system programs as well as applications as a search structure. In this study, we use a hash table to implement an ordered integer set. The hash table has multiple buckets, each managing an ordered linked list. Given an integer key k, the hash function $h(k)$ determines the bucket, where the key might be inserted, deleted, or accessed. We We implemented four different versions of concurrent hash tables:

- *Coarse-grain lock based version:* each bucket is protected by a spin-lock (implemented with MCS-SW, see Section 4.1.4), which has to be acquired before the insertion, deletion, or search operation, and released afterwards.

- *Lock-free version:* uses Michael's lock-free hash table algorithm [119]. The *hazard pointers* mechanism is used to guarantee safe memory reclamation of lock-free objects as well as ABA-safety [122].
- *sw-rwlock version:* uses software based read and write locks. A lock variable is added into the data structure of the node of the hash table. Read locks are continuously acquired and released for accessed nodes, while the code travels through a selected ordered linked list to perform the search operation. When the position where the key to be inserted or deleted is found, the corresponding read locks are upgraded to write locks, and the operations are performed. This version increases the memory usage of every node by 50%.
- *SSB version:* similar as the sw-rwlock version. SSB read and write lock operations are used to replace the software-based ones. There is no need to modify the data structure of the node, thus no extra memory usage.

To evaluate the performance of these implementations, the hash table is initialized with 10 buckets and a load factor of 100, which represents the average number of items per bucket. Each thread performs 1,000 operations, of which 20% are insertions, 20% are deletions, and 60% are searches. At each iteration, the operation to be performed is randomly determined, after which a small random delay is inserted.

Figure 6.7 and Figure 6.8 shows that the SSB based version achieves the best performance when the number of threads is greater than 1. The execution time of the coarse-grain lock-based version keeps increasing with the number of threads, because of the contention when multiple threads access the same bucket concurrently (see Figure 6.7). The other three fine-grain versions show near constant execution time even when the number of threads reaches 128. With SSB instructions, the synchronization overhead is small when there is no contention. Both the lock-free and sw-rwlock version needs to check the return value of the synchronization operations (CAS, or lock acquisition). Therefore,



Y-axis: the normalized execution time by number of threads.

Figure 6.8: Concurrent Hash Table: Comparison of Three Different Fine-Grain Synchronization Solutions

even without contention, a synchronization operation incurs overhead at least equal to a load operation. In addition, the lock-free version also needs to pay certain cost for the safe memory reclamation. As shown in Figure 6.8, when running on a single thread (i.e., no contention), the lock-free version and sw-rwlock version are 56% and 42% slower than the sequential version, respectively, whereas the SSB-based version is only 9% slower. In all cases, the SSB version is at least 14% and up to 84% faster than the other two versions without any extra memory usage.

6.5 Effectiveness of SSB for Fine-Grain Data Synchronization

An important class of the target applications for large-scale multi-core architectures are scientific numerical computations, many of which are intrinsically deterministic - that is for a given input a fixed output (result) should be produced no matter how the program is parallelized. Under a shared-memory parallel programming model, it is critical that the data dependencies in such programs should be realized efficiently to best exploit parallelism.

One of the functionalities of SSB is to provide efficient fine-grain data synchronization, which ensures that a consumer thread reads a value at word-level in memory only after it has been written by a producer thread. Based on SSB, this section (1) compare SSB-based fine-grain data synchronization to three software based synchronization methods [99] using 6 DOACROSS style kernel loops extracted from SPEC OMP 2001 benchmark suite; (2) investigates the parallelization of three representative scientific computation kernels using fine-grain data synchronization.

These kernels represent three typical computation patterns in scientific applications: iterative approximation in finite difference method, linear recurrence with irregular pattern of data dependencies, and the wavefront form of computation. For each kernel, we demonstrate how it can be effectively parallelized with word-level fine-grain data synchronization, which expresses the producer-consumer relation between the computation of concurrent threads. Unlike global synchronization (i.e., barrier) based coarse-grain parallelization, where read-after-write data dependencies are enforced by making all consumers wait for all producers at a common synchronization point, the fine-grain data synchronization based parallelization takes a point-to-point synchronization approach, which allows the consumer only waits for the data it needs for proceeding the computation. Therefore, fine-grain synchronization can avoid unnecessary waiting and global communication that caused by coarse-grain barrier synchronization. Using detailed simulation, our experimental results demonstrate:

- On multithreaded large-scale multi-core architectures, fine-grain data synchronization mechanism is important and effective for exploiting fine-grain parallelism in scientific application kernels.

- For large-scale multi-core architecture, fine-grain synchronization based parallelization schemes can achieve significant performance improvement over the coarse-grain ones. For the three representative kernels we investigated, when running with 128 threads, fine-grain based implementation outperforms the coarse-grain ones by 38.1%, 312%, and 94.9% respectively.

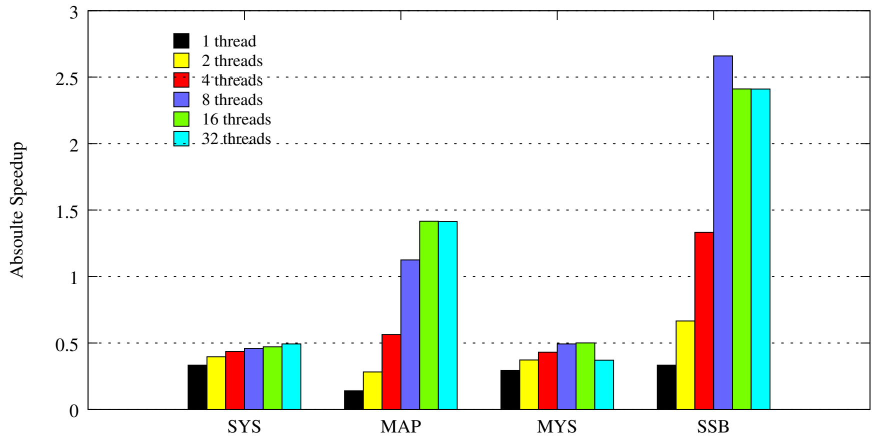
- With only modest hardware extension to multi-core architectures, SSB provides an efficient mechanism for enforcing read-after-write data dependencies at word-level in memory among concurrent threads.

6.5.1 Kernel Loops from SPEC OMP

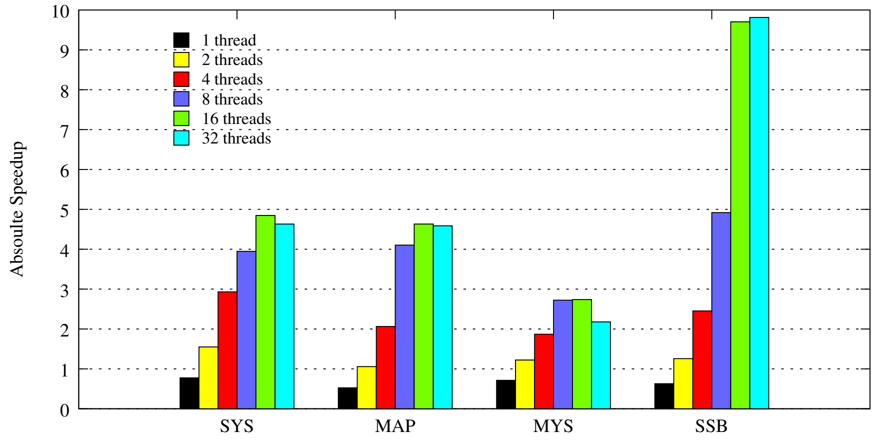
The 6 kernel loops, $K1$, $K2$, ..., $K6$, are extracted from multithreaded applications, such as *314.mgrid* and *318.galgel*. [1] The cross-iteration dependence distance of all the kernels are constant and positive. We parallelize those loops by statically assigning iterations to different threads in a round robin fashion. We compared the SSB-based approach with the three software-based synchronization methods (SYS, MAP, and MYS), which are recently proposed by Kejariwal et. al. [99]. For more details, please refer to [99]. For the SSB-based approach, we use SSB SWSR operations to enforce the data dependencies among threads.

The workloads for each iteration of $K1$, $K2$ and $K3$ are small. For instance, there is only one arithmetic operation in the loop body of $K1$. Because of the low computation to synchronization ratio, none of the methods show significant absolute speedup. However, in all cases (Figure 6.9(a), (b), (c)), SSB-based approaches show better performance than software methods. For kernel $K4$, $K5$, $K6$, all with a two-level loop nest, the workloads inside each iteration of the outer loop are large. The software methods can only exploit the parallelism of the outer loop. The SSB-based method can naturally

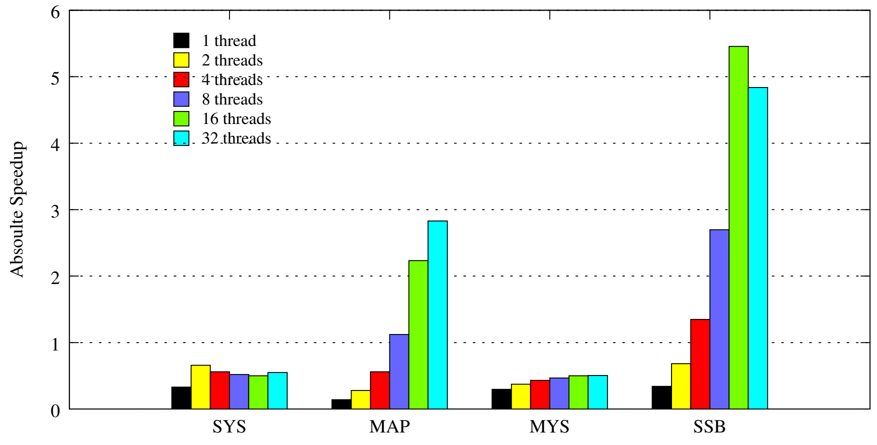
[1] These 6 kernel loops are the same ones used in the performance evaluation section of [99].



(a) $K1$ (DIST = 8)

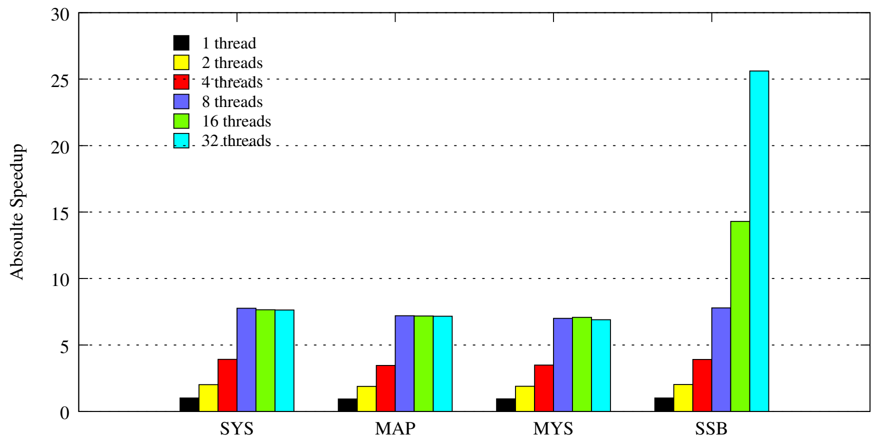


(b) $K2$ (DIST = 16)

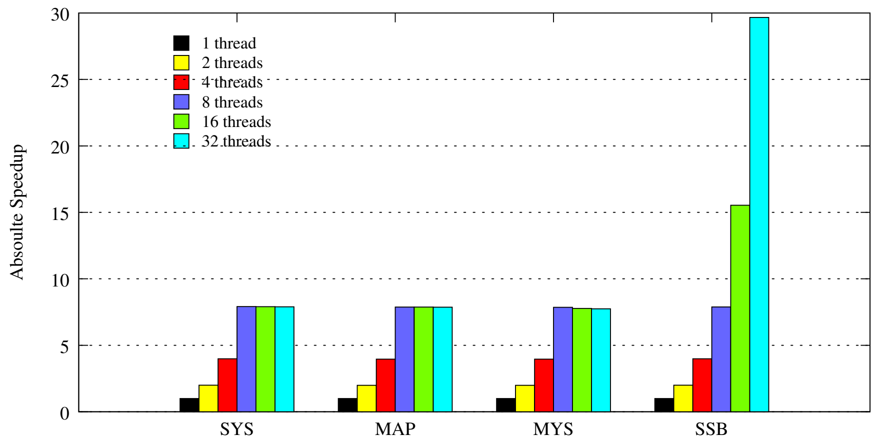


(c) $K3$ (DIST = 8)

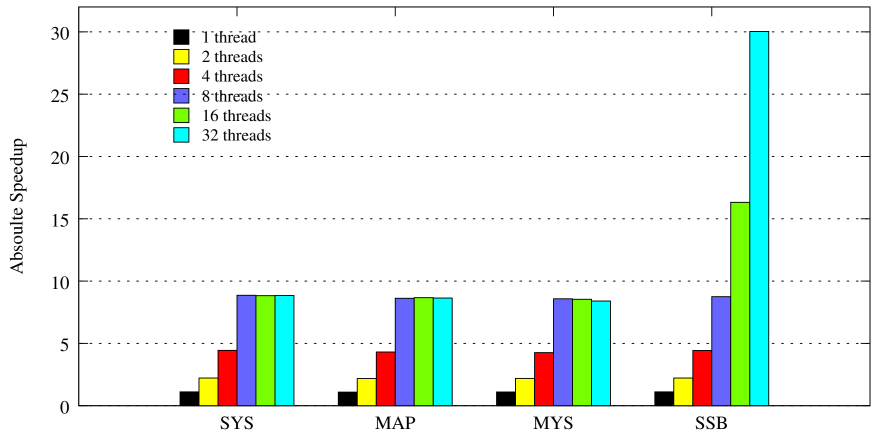
Figure 6.9: Performance of Multithreaded DOACROSS Kernel Loops ($K1$, $K2$, $K3$). (DIST: dependence distance.)



(a) $K4$ (DIST = 8)



(b) $K5$ (DIST = 8)



(c) $K6$ (DIST = 8)

Figure 6.10: Performance of Multithreaded DOACROSS Kernel Loops ($K4$, $K5$, $K6$). (DIST: dependence distance.)

```
for( i = 0; i < ITERATIONS; i++){
   for( j=1; j< TOTALSIZE-1; j++ )
      xnew[j] = 0.5*( x[(j-1)]+x[(j+1)]+b[j] );
   for( j=1; j< TOTALSIZE-1; j++){
      x[j] = xnew[j];
}
```

Figure 6.11: Sequential version of 1D Laplace Solver

exploit fine-gain parallelism in the loop nests with no overhead of memory usage. Therefore, the SSB-based approach shows much better scalability than the software-based approaches (Figure 6.10(a), (b), (c)). These 6 loops illustrate the effectiveness of SSB-based fine-grain data synchronization (compared to state-of-the-art software approaches) for DOACROSS loops with simple cross-iteration dependencies. The following two benchmarks illustrates how SSB can help in exploiting fine-grain parallelism of applications with complex data dependencies, which cannot be easily handled by software methods.

6.5.2 1D Laplace Solver

Laplace's equations is a famous partial differential equation, which is important in many fields of science, such as electromagnetism, astronomy, and fluid dynamics. The 1D Laplace solver use a finite difference method to achieve numerical approximation of the equation. We use a hypothetical 1D Laplace solver to demonstrate the effectiveness of using fine-grained data synchronization to enforce the read-after-write dependence among threads.

In the kernel of the Laplace solver, at each iteration, every position of a single-dimension array is updated with a value function of its left and right neighbors that computed from the previous iteration. All the elements of the array need to be updated before the next iteration starts (see Figure 6.11). For simplicity, within each iteration, two arrays are actually used. One stores the data computed by previous iteration, the other stores the data generated by the current iteration.

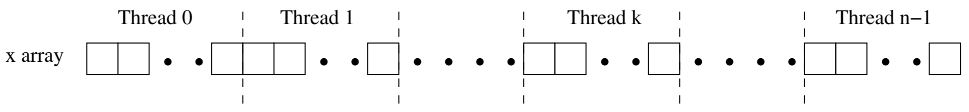
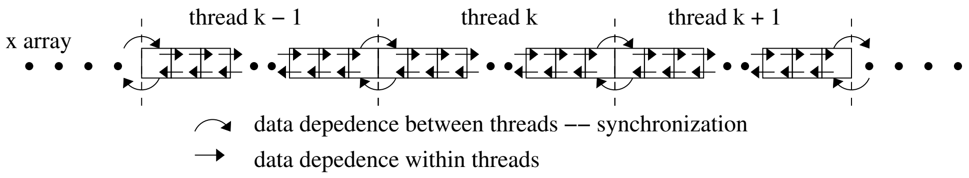


Figure 6.12: 1D Laplace Solver: Partition the Array among n Threads



thread k − 1
thread k
thread k + 1
x array
data depedence between threads –– synchronization
data depedence within threads

Figure 6.13: Data Dependencies and Synchronizations in 1D Laplace Solver

The multithreaded parallel implementation partitions the 1D array among threads, as shown in Figure 6.12. To enforce the producer-consumer relation, a barrier is performed after all $xnew$ are computed, and another barrier is executed after $xnew$ is copied to x. This *barrier* based coarse-grain synchronization scheme enforces each thread to wait for all others completing the current iteration before starting the next one.

From the point of view of a thread, however, it only needs to wait for its two neighbor threads to supply the data at the border of its partition in order to continue its own computation (see Figure 6.13). Assuming that the portion of the x array assigned to a thread is between x_{start} and x_{end}, in order to start its next iteration, this thread only needs to read two elements from its two neighbors. For instance, for starting the computation of $xnew_{start}$ and $xnew_{end}$ at iteration i, the thread only needs its two neighbors to write their results into $x_{start-1}$, and x_{end+1} at iteration $i-1$.

Using this scheme, we can implement another parallel version of the solver using the SSB single-writer-single-reader operations to perform the fine-grain data synchronization between threads. The coarse-grain barriers are removed, the data synchronization is used to enforce each thread to wait for the data that is exactly necessary for starting the

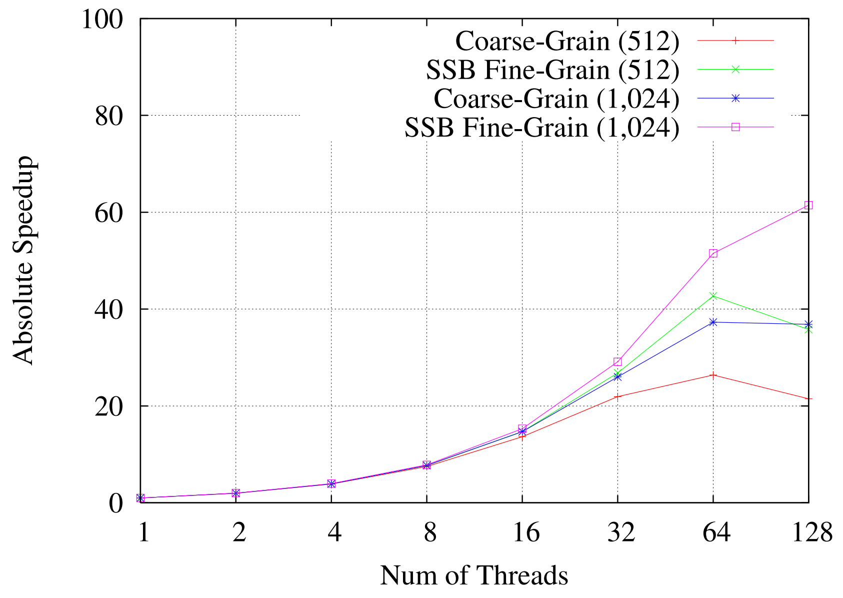


Figure 6.14: Barrier-based Coarse-Grain Synchronization vs. SSB-based Fine-Grain Synchronization for 1D Laplace Solver. (Problem Size: 512, and 1,024)

new iteration.

Figure 6.14 and 6.15 demonstrates the effectiveness of the SSB-based fine-grain synchronization, which naturally expresses the data dependencies in the original 1D Laplace solver problem. The "one-to-one wait" data synchronization strategy avoids the unnecessary "all-to-all wait" scenario due to the use of barrier as well as the overhead of barrier. As a result, the SSB-based fine-grain synchronization approach beats the barrier based coarse-grain counterpart in all cases, even the C64 hardware-based barrier is very efficient. For example, when the solver runs on 128 threads with a problem size of 4,096, the SSB-based version can achieve a speed up of 109, and outperform the coarse-grain version by 38.1%.

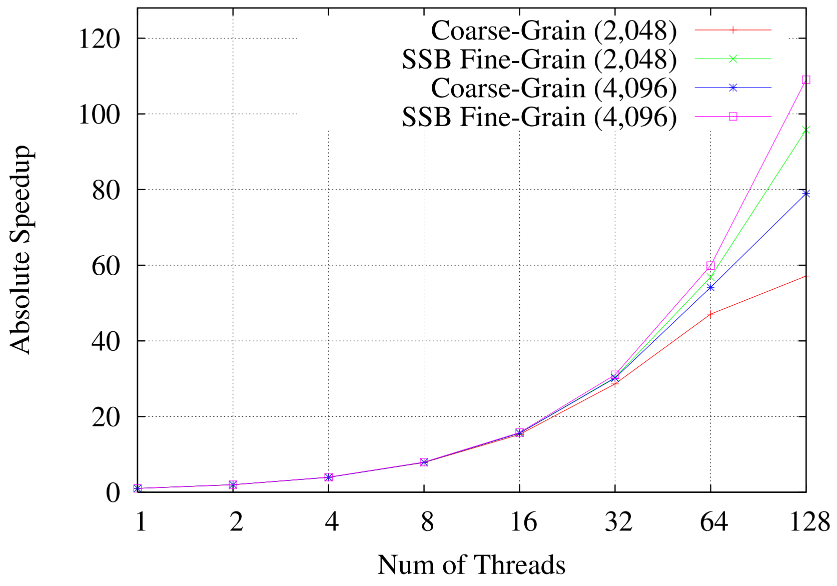


Figure 6.15: Barrier-based Coarse-Grain Synchronization vs. SSB-based Fine-Grain Synchronization for 1D Laplace Solver. (Problem Size: 2,048, and 4,096)

```
for ( i=1 ; i<n ; i++ )
    for ( k=0 ; k<i ; k++ )
        W[i] += b[k][i] * W[(i-k)-1];
```

Figure 6.16: Livermore Loop 6: Linear Recurrence Equations

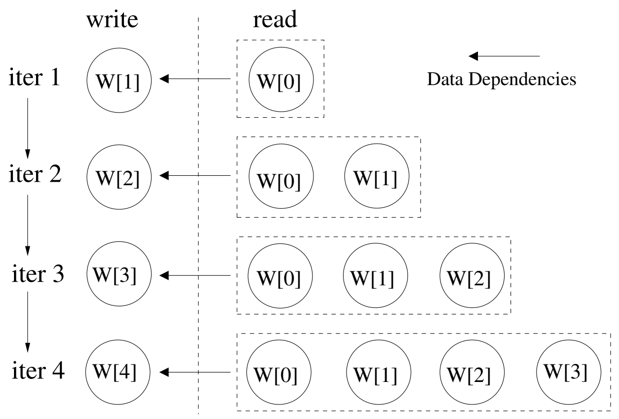


Figure 6.17: Characteristics of Linear Recurrence Equations

6.5.3 Linear Recurrence Equations (Livermore Loop 6)

Linear recurrence equations are widely used in scientific linear algebra computations. Livermore Loop 6 [56], which is shown in Figure 5.2, represents a general form of linear recurrence equations. (To ease the reading, the code of the loop is reintroduced in this section as Figure 6.16). As shown in Figure 6.17, the outer loop computes the array `W`, and at each iteration `i`, `W[i]` depends on values computed in all previous iteration, that is, `W[i]` depends on `W[0]`, `W[1]`, ... , `W[i-1]`. Such cross-iteration dependencies of array `W` makes it very difficult to parallelize this loop at compilation time [155].

A coarse-grain based parallelization approach is introduced in [56]. Observe that an element $W[i]$ appears in the sum of all elements whose indices are higher than its, i.e., $W[i+1]$, $W[i+2]$, ... , and $W[n]$. Therefore, at iteration i instead of summing elements $W[1]$ through $W[i-1]$ to compute $W[i]$, one can add $W[i-1]$ multiplied by the corresponding elements of b to $W[i]$ through $W[n]$ [56]. To achieve the new scheme, the original loop is transformed to a new one as shown in Figure 6.18.

An iteration of the outer loop of the original Loop 6 (see Figure 6.16) is an inner product of know values. Given unlimited number of threads, the time complexity is $O(Logn)$. In the new loop (see Figure 6.18), at an iteration of the outer loop , all the

```
for ( k=0 ; k< n - 1 ; k++ )
   for ( i= k + 1 ; i < n ; i++ )
      W[i] += b[i-1][i-k] * W[k];
```

Figure 6.18: Livermore Loop 6 after Loop Transformation

addition in the inner loop can be done in parallel, thus the time complexity is $O(1)$. Our coarse-grain implementation is based on the new loop. At each iteration of the outer loop, the computation of the inner loop is partitioned to different threads. After the computation, all threads join a barrier, then start next iteration.

A recent Micro paper claims that the irregular pattern of synchronizations in this loop nests does not make them amenable to point-to-point synchronization, and a global barrier is a natural choice in this code [144]. Against their belief, we regard the barrier-based coarse-grain approach has several drawbacks:

- **Decreasing parallelism:** In this approach, the parallelism available each iteration is continuously decreasing with the increase of the iterator k of the outer loop (see Figure 6.18). During the epilog of the loop, threads do not have enough computation task to do to justify the cost of parallelization. Moreover, since the number of iterations of the inner loop, which keeps changing, can not always be divided evenly among threads, the computation is normally unbalanced.

- **Poor Locality:** Notice that each iteration of the inner loop computes a different $W[i]$. When an iteration of the inner loop is assigned to a thread, a corresponding $W[i]$ is loaded, a new value is computed, and stored back to the global memory. Therefore, there is no locality exploited for $W[i]$.

- **Barrier:** The barrier at the end of each iteration creates an "all-to-all wait" scenario, which requires global communication among threads.

To address above issues, we attempt to use SSB-based fine-grain synchronization to parallelize the loop. The parallelization is based on the original loop code shown in Figure 6.16. For simplicity, we assign the iterations of the outer loop to different threads in a round-robin fashion. Using this scheduling policy, the computation for a $W[i]$ is performed by a single thread. Therefore, the partial result of $W[i]$ can be kept in the register of the thread to exploit the locality. Only when the final result is computed, the $W[i]$ is written back to global memory.

The SSB single-reader-multiple-writer data synchronization mechanism is used to enforce the read-after-write dependencies among iterations. A straightforward synchronization scheme can be designed following the data dependencies characteristics of the loop, as shown in Figure 6.17. For computing $W[i]$, a thread uses synchronized read to load $W[1]$, $W[2]$, ..., and $W[i-1]$. After $W[i]$ is calculated, the result is written back to memory via a synchronized write, which should set the number of readers as $n-i$. However, this approach generates excessive synchronization, which is not efficient and can cause the SSB to become full.

By analyzing the loop nests carefully, our actual parallelization and synchronization scheme is shown in Figure 6.19, which illustrates the case where 8 iterations are concurrently executed by 4 threads, and the chunk size of round-robin scheduling is 1 iteration. When thread 1 completes iteration 1, it notifies threads 2, 3, and 4 about the availability of $W[1]$. Thread 1 then executes iteration 5 according to the round-robin work distribution policy. Although the computation of iteration 5 depends on $W[1]$ to $W[4]$, it does not have to explicitly wait for $W[1]$, since thread 1 itself computed $W[1]$ previously. Similarly, when thread 2 moves to iteration 6, it does not need to check the availability of $W[1]$,or $W[2]$, because $W[2]$ is computed by itself previously, and when $W[2]$ is available, $W[1]$ is ensured to be available. By taking this synchronization strategy, after the computation of an iteration, a thread performs a synchronized write to the memory to notify num_threads -1 readers. When a thread begins a new iteration i to compute

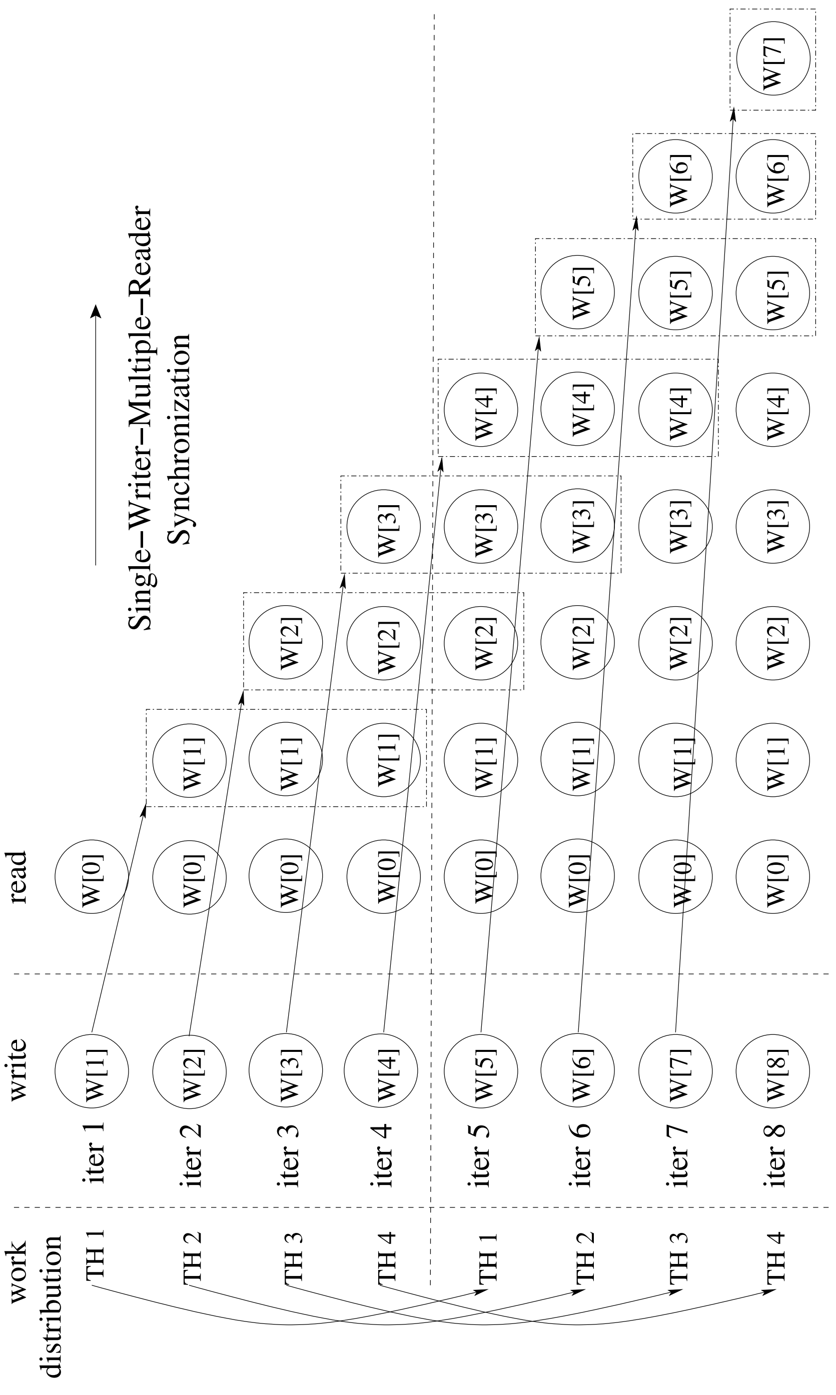


Figure 6.19: Parallelization and Synchronization of Livermore Loop 6 (*4 threads, round-robin scheduling, chunk size = 1*)

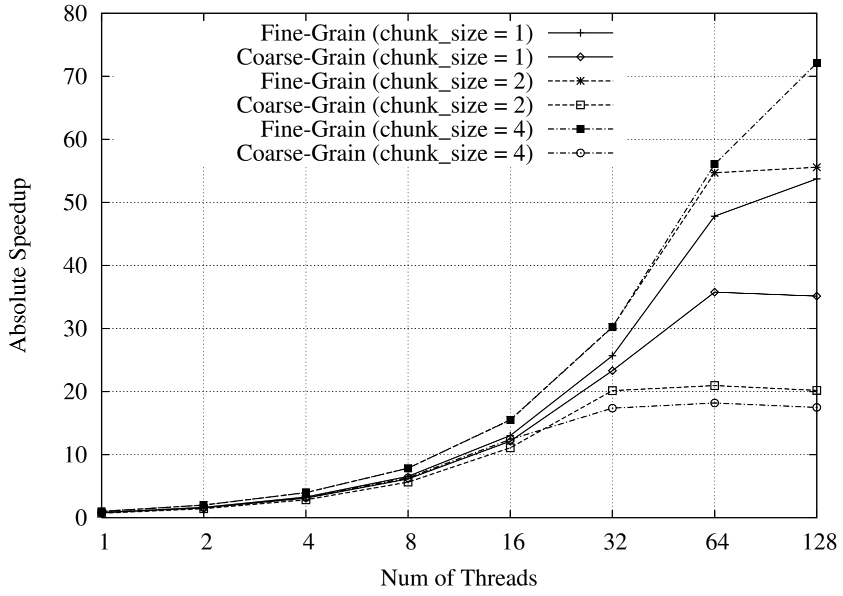


Figure 6.20: Speedup of Parallelized Livermore Loop 6

$W[i]$, it uses normal load operations to read from $W[0]$ to $W[(i-1)-(\text{num\_threads}-1)]$, and uses synchronized read to load the remaining num_threads $-$ 1 elements of W. As a result, no matter how large the problem size, the number of synchronization reads and writes required only depends on the number of threads.

Compared to the barrier-based coarse-grain approach, our fine-grain solution 1) exploits the inherent fine-grain parallelism in the computation, thus can achieve better workload balancing at runtime; 2) achieves much better locality as explained before; and 3) eliminates the use of barrier, thus avoids the overhead of the barrier as well as the unnecessary "all-to-all wait" scenario.

Our parallelization and synchronization scheme is shown in Figure 6.19, which illustrates the case where 8 iterations are concurrently executed by 4 threads, and the chunk size of round-robin scheduling is 1 iteration. When thread 1 completes iteration 1, it notifies threads 2, 3, and 4 about the availability of $W[1]$. Thread 1 then executes iteration 5 according to the round-robin work distribution policy. Although the computation of iteration 5 depends on $W[1]$ to $W[4]$, it does not have to explicitly wait for $W[1]$, since thread 1 itself computed $W[1]$ previously. Similarly, when thread 2 moves to iteration 6, it does not need to check the availability of $W[1]$,or $W[2]$, because $W[2]$ is computed by itself previously, and when $W[2]$ is available, $W[1]$ is ensured to be available. By taking this synchronization strategy, after the computation of an iteration, a thread performs a synchronized write sswmr_w to the memory to notify num_threads $-$ 1 readers. When a thread begins a new iteration i to compute $W[i]$, it uses normal load operations to read from $W[0]$ to $W[(i-1)-(\text{num\_threads}-1)]$, and uses synchronized read (sswmr_r) to load the remaining num_threads $-$ 1 elements of W. As a result, no matter how large the problem size, the number of synchronization reads and writes required only depends on the number of threads. It is now obvious that this application kernel also satisfy the Equation 5.2 ($S(t) \ll M \times B$) introduced in Section 5.4.

Given a W array with size 5,120, Figure 6.20 compares the fine-grain data synchronization based approach with the coarse-grain based implementation. For the fine-grain approach, the *chunk_size*, as explained before, is the number of iterations to be scheduled per time by the round-robin algorithm. For the coarse-grain approach, the parallel version is based on a sequential version that has been loop unrolled certain times specified by the *chunk_size*. In Figure 6.20, when *chunk_size* equals to 2 or 4, the speedups are calculated against the sequential versions, which have been loop unrolled twice and 4 times respectively. Therefore, the comparison of two curves will be meaningful, only if the *chunk_size* is the same. As shown in Figure 6.20, by exploiting fine-grain parallelism, the single-writer-multiple-reader fine-grain data synchronization based approach always performs better when running on a large number of threads.

Figure 6.21 shows the performance improvement of the SSB-based fine-grain approach over the coarse-grain one (calculated as $(Speedup_{fine-grain} - Speedup_{coarse-grain})/Speedup_{coarse-grain}$). From Figure 6.21, we can observe that the performance improvement increases significantly when the number of threads is large.

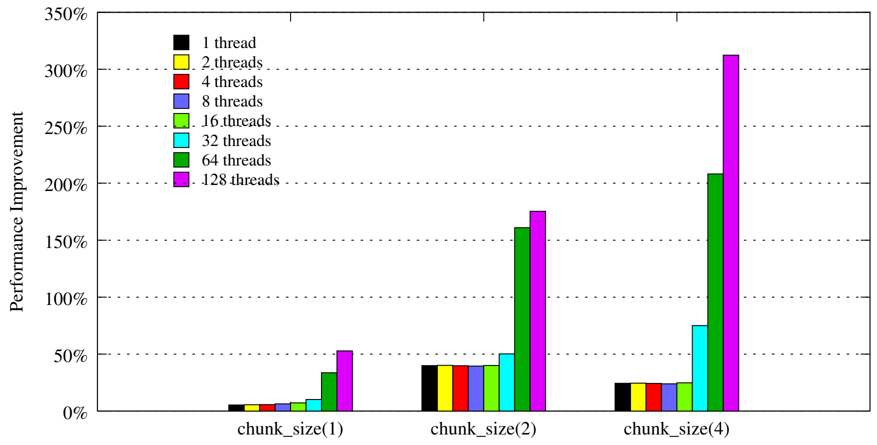


Figure 6.21: Livermore Loop 6: Fine-Grain vs Coarse-Grain

For example, when 128 threads are used, the fine-grained approach with a chunk size of 4 achieves an absolute speedup of 72, which demonstrates a 312% improvement over the corresponding coarse-grained parallelization scheme. This proves the effectiveness of the SSB-based fine-grain synchronization for exploiting massive on-chip parallelism in the large-scale multi-core chips. It can also be noticed that the fine-grain approach can take better advantage of the commonly used loop optimization techniques, such as loop unrolling. The performance advantage of the SSB-based fine-grain approach is attributed to 1) fine-grain parallelism exploited by SSB-based synchronization solution; 2) better data locality; and 3) the removal of the barriers.

6.5.4 Wavefront Computation

Wavefront computations are common in scientific applications. Given a matrix (see Figure 6.22), the left and top edges of which are all 1, the computation of each remaining element depends on its neighbors to the left, above, and above-left. If the solution is computed in parallel, the computation at any instant form a wavefront propagating toward in the solution space. Therefore, this form of computation get its name as wavefront.

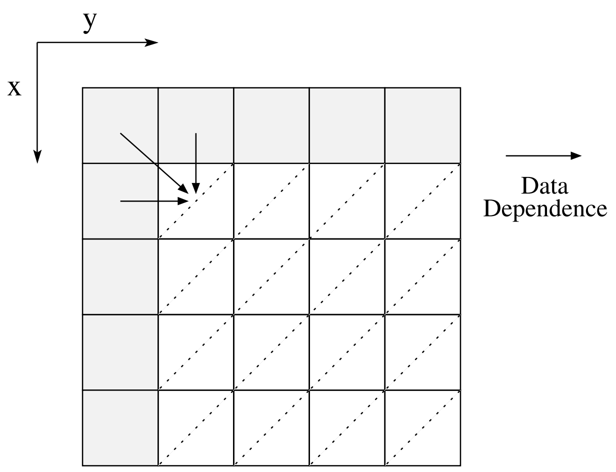
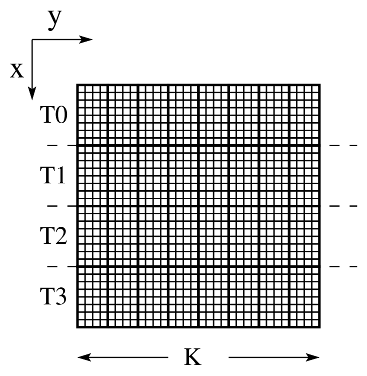


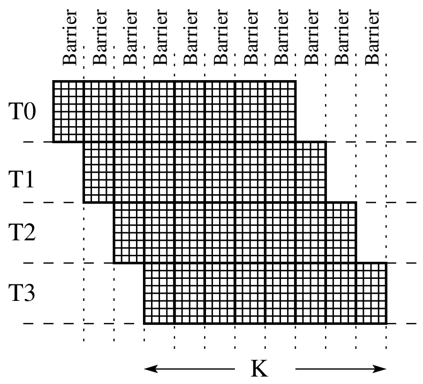
Figure 6.22: Wavefront Computation

Figure 6.23 illustrate the coarse-grain parallelization scheme inspired by [164]. The solution space is partitioned along the x dimension. Let T be the number of threads, X be the number of rows. Each thread is assigned with the computation of X/T contiguous rows. In order to gain parallelism, the solution space is further partitioned to K blocks along the y dimensions. Each thread completes all the computation in a block, joins a barrier, and start the computation of the next block. As shown in Figure 6.23(b), the computation is performed as a pipeline, and the data dependencies between blocks are enforced by the barrier. The parameter K determines the degree of the parallelism. With the increase of K, the granularity of data associated with each barrier synchronization is decreasing, and the number of global synchronizations (barriers) required is increasing. Therefore, the level of parallelism can be exploited is determined by the efficiency of the barrier synchronization. However, the cost of the barrier normally increases with the number of threads.

In our fine-grain implementation, the rows of the matrix are assigned to threads in a round-robin fashion (modulo T, see Figure 6.24). With this static scheduling policy, to compute an element, only the availability of its above neighbor needs to be checked.



(a) Partition the Solution Space



(b) Computation Scheme in Parallel

Figure 6.23: Coarse-Grain Parallelization of Wavefront Computation: Number of Threads $T = 4$, Number of Computation Blocks $K = 2 * T = 8$.

SSB fine-grain single-writer-single-reader synchronization can be used to force the data dependencies. Again, a straightforward synchronization scheme is to allow synchronized read/write on each data elements.

To improve the efficiency, avoid excessive synchronization, and reduce the chance of a SSB becoming full, we takes a blocking approach. To reduce the amount of the synchronization, we group 8 consecutive elements in a row as a block. Once a thread completes the computation for a block, it writes the first element of the block to the memory with a synchronized write, and the other elements in the block are written with normal store instruction. Afterwards the thread moves to the next block. Before the computation of a block, a thread performs a synchronized read to get the first element of the block, the remaining elements of the block are read with normal load instruction. With the fine-grain solution, although the computation is still in a wavefront form, a thread can be kept busy as soon as the block, which it is waiting for, becomes available. Except the prolog and epilog stage of the computation, all threads can be kept usefully busy in a pipelined fashion. Unlike the global synchronization with barrier, threads never wait unnecessarily using point-to-point data synchronization.

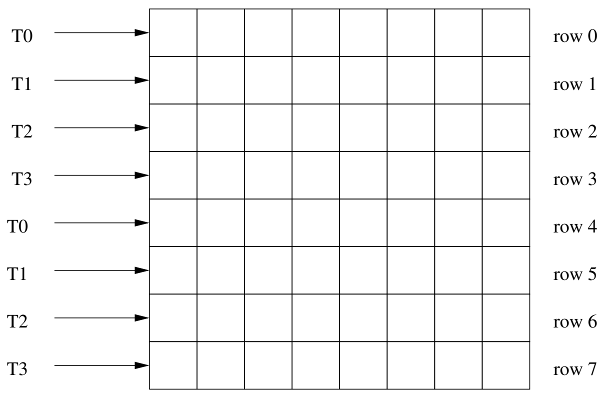


Figure 6.24: Fine-Grain Parallelization of Wavefront Computation: Number of Threads $T = 4$, Solution Space 8×8.

Our experiments are conducted with a 1024×1024 matrix [2]. Figure 6.25 compares the speedups of the fine-grain approach to the coarse-grain ones.

Recall that we group 8 consecutive elements in a row as a block in our fine-grain synchronization based parallelization scheme. In the code, the calculation of the 8 elements in a block is written in a way that is similar as loop unrolling. To make a fair comparison, the inner most loop of the coarse-grain based implementation is also unrolled 8 times. The absolute speedups shown in Figure 6.25 is also calculated against the sequential version, whose inner loop has been unrolled 8 times. For the coarse-grain approach, we examined three different versions by experimenting different values of K.

From Figure 6.25, it is apparent that the SSB-based fine-grain approach outperforms the coarse-grain ones when running with multiple concurrent threads. For the coarse-grain versions, it can also be observed that a larger K can improve performance only if the number of threads is moderate. When number of threads is large, the cost of barrier cancels out the performance benefit brought by associating finer grain of data (due

[2] A 1024×1024 matrix of doubles exceeds the capacity of on-chip SRAM memory of current C64 chip design. For the purpose of our experiments, we extend the on-chip SRAM memory to 10M in the simulator.

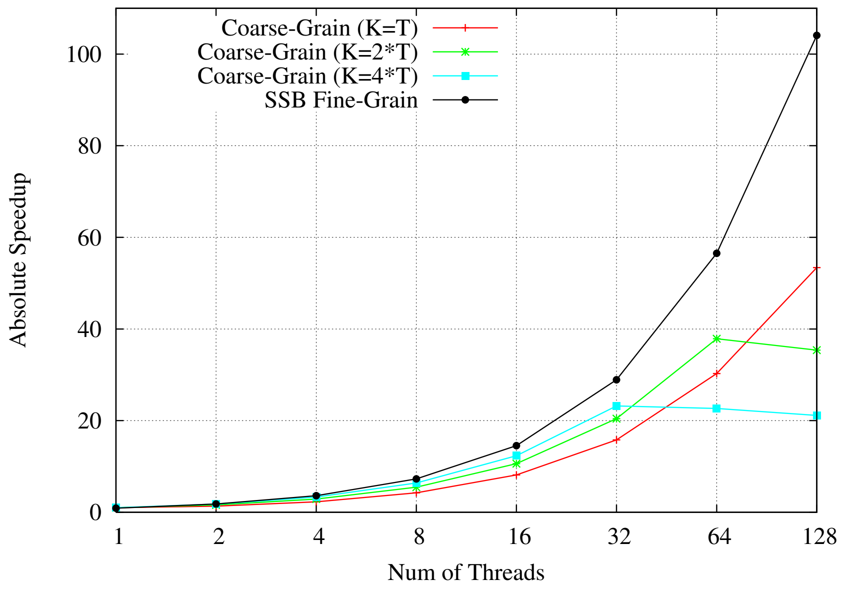


Figure 6.25: Absolute Speedup of Parallelized Wavefront Computation. T denotes the number of threads, and K denotes the number of computation blocks

to larger K) with each barrier synchronization.

Although the data dependencies in wavefront computation implies serialization, the multithreaded implementation with fine-grain data synchronization demonstrates the capability to exploit the inherent parallelism within such computation form. When running with 128 threads, the SSB-based implementation shows an absolute speedup of 104, which outperforms the three coarse-grain synchronization based implementation by 94.9%, 194.2%, and 392.7%, respectively.

6.6 Effectiveness of SSB for Fine-Grain Synchronization

We measured the percentage of successful synchronization among all synchronizations issued for the 8 benchmarks shown in Table 6.3. We can see that even for large number of threads, most fine-grain synchronization operations are successful, which in turn ensures low cost of synchronization (see Section 6.3).

Table 6.3: Synchronization Success Rates and SSB Full Rates

Benchmark	64 threads		128 threads	
	Success Rate (%)	SSB Full Rate (%)	Success Rate (%)	SSB Full Rate (%)
Random Access	99.98%	0	99.96%	0
Livermore Loop 13	99.11%	0	98.42%	0
Livermore Loop 14	99.72%	0	99.59%	0
Loop $G1$ from SSCA#2	99.98%	0	99.97%	0
Ordered Integer Set	99.97%	0	99.93%	0.0004%
1D Laplace Solver (4,096)	88.20%	0	84.29%	0
Livermore Loop 6 (chunk size = 4)	87.52%	0	72.13%	0
Wavefront	99.86%	0	99.83%	0

The Livermore Loop 6 has relatively low successful rate compared to others. This is because certain portions of synchronized reads happen before the corresponding synchronized writes. We do not show kernel loops $K1$, $K2$, ..., $K6$ in Table 6.3. For those loops, when the number of threads are smaller than or equal to the dependence distance (shown as DIST in Figures 6.9 and 6.10), the synchronization successful rates are also very high. Otherwise, the rates are not high, since certain portion of synchronized reads are destined to fail at first attempt in such cases. For example, when dependence distance is 8 and 16 threads start computation at the same time, the first attempt of synchronized read from thread 9 to 16 will fail, because the corresponding synchronized writes from thread 1 to 8 have not yet finished.

We also observed that only one benchmark encounters the situation where the SSB happens to be full. The percentage is only 0.0004% among all synchronization operations issued. In all other benchmarks, the buffer is never filled up. This analysis shows that a small SSB for each memory bank is normally sufficient to cache the access states of outstanding synchronizing data units for multithreading programs. Using modest hardware cost, SSB achieves the same effect as if each word of the entire memory is tagged.

Chapter 7

CONCLUSION

7.1 Summary

The design of high-performance processor chips is now rapidly moving towards large-scale multi-core architectures that integrates 10s (or beyond) of tightly-coupled processing cores on a single chip. This emerging technology trend on multi-core chip architecture is primarily driven by the following factors:

- With the advances in IC processing technology, the number of transistors that can be fabricated into a single die is now reaching one billion. It becomes possible to put a complete multiprocessor, including both CPUs and memory, on a single chip.
- The delivered performance versus number of transistors integrated in a chip for conventional single-core superscalar or VLIW architecture, keeps declining over time.
- Power consumption and heat dissipation limits have emerged as major obstacles in the design of micro-architecture with traditional architecture technology.
- The pervasiveness of thread-level and process-level parallelism in applications.

With this new generation high-performance large-scale multi-core architectures, parallel processing, especially multithreading, becomes a must for taking advantage of the massive intra-chip parallelism presented. It has been long realized that synchronization is one of the most critical technique to facilitate multithreading in terms of both correctness and performance. Moreover, in order to fully utilized the massive intra-chip parallelism provided by such large-scale multi-core chips, it is important to exploit the fine-grain parallelism inherent in the applications. The granularity of parallelism that can be efficiently exploited in such processors is often restricted by the lack of effective architectural support for efficient fine-grain synchronization.

In this thesis, by taking the IBM 160-core Cyclops-64 chip architecture as a case study. we evaluated and analyzed a wide range of synchronization mechanisms on a state-of-the-art large-scale multi-core architecture. Based on the performance evaluation, we also proposed customized algorithms/implementations of chosen synchronization mechanisms by taking advantage of underlying hardware features of large-scale multi-core chip architectures.

In order to facilitate efficient and effective fine-grain synchronization, we proposed a novel synchronization architecture with a modest hardware extension to large-scale multi-core architectures, called *Synchronization State Buffer* (SSB). SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of frequently synchronized data units to support and accelerate word-level fine-grain synchronization. The SSB design avoids enormous on-chip memory storage cost, and yet creates an illusion that each word in memory is associated with a set of states by only attaching a small hardware buffer to the memory controller of each memory bank. We also presented an architectural model for SSB. Based on this model, SSB can be used to provide a rich set of fine-grain synchronization functionalities, such as enforcing mutual exclusion and read-after-write data dependencies between a large number of threads.

We implemented SSB on the simulator of IBM Cyclops-64 architecture. With detailed simulation, the experimental results demonstrate the effectiveness and efficiency of our solution by showing significant performance improvement for several representative benchmarks due to the use of SSB fine-grain synchronization mechanism. To the best of our knowledge, this thesis is the first work that explores hardware support of word-level fine-grain synchronization for large-scale multi-core architectures, such as C64.

7.2 Future Work

This thesis demonstrated how fine-grain synchronization can be efficiently supported on large-scale multi-core architectures with only modest hardware extension. In also inspires several interesting future works. They are detailed as follows.

Our current design assumes non-preemptive thread model, which provides a good starting point to implement the idea of SSB. To explore preemptive threads, virtualization and other more elaborate hardware mechanism will be necessary for implementing SSB design. The virtualization of SSB is beyond the scope of the current dissertation, and we regard this as an important future work.

The proposed SSB-based fine-grain synchronization solution is a hardware-based mechanism. The interface between software and hardware is a set of add-on instructions to the original ISA of multi-core architectures. For the benchmarking in this thesis, we hard-coded the SSB instructions into the source code of the benchmarks through GCC intrinsics or embedded assembly. We also hand-optimized the assembly code in some cases. However, for application developers, these methods are hard to use and not productive. It is important to investigate language extensions that can help compiler map the high-level programming constructs to the SSB synchronization operations. It will be interesting to leverage the compilation techniques from Tera [11] and other multiprocessor machines with tagged memory.

Recently, Zhang et. al. presents a compilation framework such that compiler can automatically assign locks to critical sections [167, 151]. While the programmer assumes a single global lock for all critical sections, which eases parallel programming, the compiler finds the minimum number of locks that can be assigned to critical sections in a parallel program without reducing its parallelism. The limitation of this work is that it only handles the statically analyzable synchronization. Therefore, the lock assignment is completed at a coarse-grain level. It will be interesting to integrate SSB-based dynamic conflict resolving techniques into this framework to further enhance the performance by exploiting fine-grain parallelism.

Since the synchronization resource provided by SSB is limited, it is important to avoid excessive synchronization that may cause the SSB resource to be used up. When an SSB for a memory bank is full, the synchronization operation traps to software method, which is far less efficient than the hardware method. Various compiler optimization techniques have been developed to minimize the amount of fine-grain synchronization added for parallel programs [108, 124, 103, 37, 129, 15, 138]. Those techniques try to minimize/reduce the amount of fine-grain synchronization operations inserted, but still preserve the parallelism that can be extracted from the code. Based on these synchronization optimization techniques and the specialty of the SSB, it is important to investigate compiler techniques that can optimize the allocation and scheduling of the SSB resources. The research on high-level language extension and compiler optimization should be integrated together to achieve productive and efficient use of SSB.

The key idea of the SSB solution is to use a small hardware buffer to cache the synchronization states of the synchronized memory locations. To put this technique in a broader context, SSB liked state bookkeeping hardware can be used to facilitate parallel program debugging, runtime performance monitoring, and the other techniques that can take advantage of efficient state recording and managing by hardware. Therefore, it will be interesting to conduct research in this direction that can lead to new hardware innovations on large-scale multi-core architectures.

BIBLIOGRAPHY

[1] Cray MTA-2 System.

[2] Meet Larrabee, Intel's answer to a GPU. `http://www.theinquirer.net/default.aspx?article=37548`.

[3] Yehuda Afek, Dalia Dauber, and Dan Touitou. Wait-free made fast. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 538–547, New York, NY, USA, 1995. ACM Press.

[4] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeoung, Geoffrey D'Souza, and M. Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[5] Ole Agesen, David L. Detlefs, Christine Flood, Alexander T. Garthwaite, Paul A. Martin, Nir N. Shavit, and Guy L. Steele Jr. DCAS-based concurrent deques. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architekture (SPAA-00)*, pages 137–146, NY, July 9–12 2000. ACM Press.

[6] B. Akgul and V. Mooney. The system-on-a-chip lock cache. *International Journal of Design Automation for Embedded Systems*, 7(1-2):139–174, September 2002.

[7] F. Allen, G. Almási, W. Andreoni, and D. Beece et. al. Blue Gene: A vision for protein science using a petaflop supercomputer. *IBM Systems Journal*, 40(2), November 2001.

[8] G. Almasi, C. Cascaval, J.G. Castanos, M. Denneau, D. Lieber, J.E. Moreira, and H.S. Warren Jr. Dissecting cyclops: A detailed analysis of a multithreaded architecture. In *MEDEA Workshop On Chip Multiprocessor: Processor Architecture and Memory Hierarchy Related Issues held in conjunction with PACT 2002 Conference*, pages 393–406, Charlottesville, Virginia, September 22- 25, 2002.

[9] George Almasi, Eduard Ayguadé, Calin Cascaval, Jesus Labarta José Castanos, Francisco Martínez, Xavier Martorell, and José Moreira. Evaluation of OpenMP for the Cyclops multithreaded architecture. *Lecture Notes in Computer Science*, 2716:69–83, Jan 2003.

[10] G.S. Almasi, C. Cascaval, J.G. Castanos, M. Denneau, W. Donath, M. Eleftheriou, M.Giampapa, H. Ho, D. Lieber, J.E. Moreira, D. Newns, M. Snir, and H.S. Warren Jr. Demonstrating the scalability of a molecular dynamics application on a petaflops computer. In *Proceedings of the 2001 International Conference on Supercomputing*, pages 393–406, Sorrento, Napoli, Italy, June 16-21, 2001.

[11] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. *SIGARCH Comput. Archit. News*, 18(3b):1–6, 1990.

[12] AMD. AMD Multi-Core. `http://multicore.amd.com/`.

[13] AMD. AMD demonstrates dual core leadership, 2004.

[14] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327, Feb 2005.

[15] James H. Anderson and Mark Moir. Universal constructions for large objects. In *International Workshop on Distributed Algorithms*, pages 168–182, 1995.

[16] T. Anderson. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[17] W. Anderson, P. Briggs, C.S. Hellberg, D.W. Hess, A. Khokhlov, M. Lanzagorta, and R. Rosenberg. Early experience with scientific programs on the Cray MTA-2. *Proceedings of the Proceedings of the ACM/IEEE SC2003 Conference-Volume 00*, 2003.

[18] ARM. ARM11 MPCore Processor. `http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html`.

[19] ARM. ARM announces first integrated multiprocessor core, 2004.

[20] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst.*, 11(4):598–632, 1989.

[21] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer Parry Husbands, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[22] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[23] D. A. Bader and J. JáJá. SIMPLE: A methodology for programming high performance algorithms on clusters of symmetric multiprocessors (SMPs). *Journal of Parallel and Distributed Computing*, 58(1):92–108, 1999.

[24] D. A. Bader and K. Madduri. Design and implementaton of the hpcs graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing (HiPC 2005)*, pages 465–476, Goa, India, Dec. 2005.

[25] P. S. Barth, R. S. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict, Functional Language with State. In *in Proc. of Conf. on 1991 Functional Programming Languages and Computer Architectures*, pages 538–568, 1991.

[26] Rudolf Berrendorf and Guido Nieken. Performance characteristics for OpenMP constructs on different parallel computer architectures. *Concurrency - Practice and Experience*, 12(12):1261–1273, 2000.

[27] Shekhar Y. Borkar, Hans Mulder, Pradeep Dubey, Stephen S. Pawlowski, Kevin C. Kahn, Justin R. Rattner, and David J. Kuck. Platform 2015: Intel processor and platform evolution for the next decade, 2005.

[28] J. Mark Bull. Measuring synchronization and scheduling overheads in OpenMP. In *Proceedings of First European Workshop on OpenMP*, Lund, Sweden, sep 30 - oct 1 1999.

[29] J. Mark Bull and Darragh O'Neill. A microbenchmark suite for openmp 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48, 2001.

[30] Doug Burger and James R. Goodman. Billion-transistor architectures - guest editors' introduction. *IEEE Computer*, 30(9):46–49, 1997.

[31] Douglas C. Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Madison, WI, June 1997.

[32] L. Carter, J. Feo, and A. Snavely. Performance and programming experience on the Tera MTA, 1999.

[33] Calin Casçaval, José G. Castaños, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E. Moreira, Karin Strauss, and Henry S. Warren Jr. Evaluation of a multithreaded architecture for cellular computing. In *Eighth International Symposium on High-Performance Computer Architecture (HPCA)*, page 311, February 2002.

[34] R. P. Case and A. Padges. Architecture of the IBM system 370. *Communications of the ACM*, 21(1):73–96, January 1978.

[35] Rohit Chandra, Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2000.

[36] Lin Chao. Preface: Meet the Intel Core Duo processor. *Intel Technology Journal*, 10(2):iii–iv, May 2006.

[37] Ding-Kai Chen. *Compiler Optimizations for Parallel Loops with Fine-Grained Synchronization*. PhD thesis, UIUC, 1994.

[38] Long Chen, Ziang Hu, Junmin Lin, and Guang R. Gao. Optimizing fast fourier transform on a multi-core architecture. In *Procs. of Workshop on Performance Optimization for High-Level Languages and Libraries*, Mar. 2007.

[39] ClearSpeed. ClearSpeed CSX600. `http://www.clearspeed.com/products/si.php`.

[40] ClearSpeed Technology. CSX processor architecture whitepaper, 2006.

[41] Kins Collins. Intel designs vs the PowerPC family. `http://www.bayarea.net/~kins/AboutMe/CPUs.html`.

[42] Standard Performance Evaluation Corporation. Spec omp (openmp benchmark suite). URL http://www.spec.org/omp/.

[43] Cray Research, Inc. SHMEM library, ready reference. Technical Report 007-3688-002, Cray Research, Inc., Mendota Heights, MN, February 1999.

[44] David E. Culler, Klaus E. Schauser, and Thorsten von Eicken. Two fundamental limits on dataflow multiprocessing. In *the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 153–164, 1993.

[45] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, inc., San Francisco, CA, 1999.

[46] Bill Dally. Computer architecture in the many-core era. In *Key note in the 24th Intl. Conf. on Comput. Design (ICCD 2006)*, 2006.

[47] W. J. Dally and et. al. The message-driven processor. *IEEE Micro.*, 12(2):23–39, 1992.

[48] Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Landing OpenMP on Cyclops-64: An efficient mapping of OpenMP to a many-core system-on-a-chip. In *Proceedings of the 3rd ACM International Conference on Computing Frontiers*, Ischia, Italy, May 2006.

[49] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *Workshop on Modeling, Benchmarking, and Simulation (MoBS2005), in conjuction with the 32nd Annual International Symposium on Computer Architecture (ISCA2005)*, Madison, Wisconsin, June 2005.

[50] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture. In *Fifth Workshop on Massively Parallel Processing, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, page 265, Denver, Colorado, USA, April 2005.

[51] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the Cyclops-64 cellular architecture. In *Proceedings of 20th International Symposium on High Performance Computing Systems and Applications*, St. John's, Newfoundland and Labrador, Canada, may 14–17 2006.

[52] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops principles of operation part I. Technical report, IBM Watson Research Center, Yorktown Heights, NY, April 2005. IBM Confidential.

[53] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops principles of operation part II. memory organization, the A-switch, and SPRs. Technical report, IBM Watson Research Center, Yorktown Heights, NY, April 2005. IBM Confidential.

[54] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul A. Martin, Nir Shavit, and Guy L. Steele Jr. Even better DCAS-based concurrent deques. In *the 14th International Symposium on Distributed Computing*, pages 59–73, 2000.

[55] Alexandre E. Eichenberger, Kathryn O'Brien, and et. al. Optimizing compiler for the Cell processor. In *in Proc. of 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005.

[56] John Feo. An analysis of the computational and parallel complexity of the livermore loops. *Parallel Computing*, 7(2):163–185, 1988.

[57] John Feo, David Harper, Simon Kahan, and Petr Konecny. Eldorado. In *Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, 2005.

[58] K. Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, University of Cambridge, September 2003.

[59] Keir Fraser and Tim Harris. Concurrent programming without locks. 2004 2004. Submitted for publications.

[60] Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian. Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 140–149, New York, NY, USA, 2003. ACM Press.

[61] Guang R. Gao. Programming and compiling for TiNy threads (TNT) – experience with Cyclops-64 architecture, Dec 2006. Invited talk at High Performance Computing Workshop on Programming Languages, Sandia National Labs.

[62] Geek.com. ChipGeek processor specs. `http://www.geek.com/procspec/procspec.htm`.

[63] GNU. GCC: the GNU compiler collection. `http://gcc.gnu.org`.

[64] GNU. The GNU Binutils. `http://sources.redhat.com/binutils/`.

[65] Simcha Gochman, Ronny Ronen, Ittai Anati, Ariel Berkovits, Tsvika Kurts, Alon Naveh, Ali Saeed, Zeev Sperber, and Robert C. Valentine. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–59, May 2003.

[66] James Goodman, M. K. Vernon, and P. J. Woest. Efficent synchronization primitives for large-scale cache-coherent multiprocessors. In *Proccedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS-III)*, pages 64–75, April 1989.

[67] Brian Gough. *An introduction to GCC: for the GNU compilers* `gcc` *and* `g++`. Network Theory Ltd., Bristol, UK, 2004. Foreword by Richard M. Stallman.

[68] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23:60–69, June 1990.

[69] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[70] Michael B. Greenwald. *Non-blocking synchronization and system design.* PhD thesis, Stanford University, 1999.

[71] Michael Gschwind. Chip multiprocessing and the Cell broadband engine. In *in Proc. of the 3rd Conf. on Computing frontiers*, 2006.

[72] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[73] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *Mico's Top Picks, IEEE Micro*, 24(6), nov/dec 2004.

[74] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13. ACM Press, Oct 2004.

[75] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[76] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[77] T. Harris and K. Fraser. Language support for lightweight transactions. In *18 th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, CA, Oct 2003.

[78] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC: International Symposium on Distributed Computing*. LNCS, 2001.

[79] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM Press.

[80] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (Third Edition)*. Morgan Kaufmann, San Mateo, CA, USA, 2002.

[81] M. Herlihy. Wait-free synchronization. *ACM Transaction on Programming Languages and Systems*, 13(1):124–149, January 1991.

[82] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer III. Software transactional memory for dynamic-sized data structures. In *the 22nd Annual ACM Symbosium*

on Principles of Distributed Computing (PODC'03), pages 92–101, Boston, MA, July 2003.

[83] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Trans. Comput. Syst.*, 23(2):146–196, 2005.

[84] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press.

[85] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[86] Ziang Hu, Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Optimization of dense matrix multiplication on IBM Cyclops-64: Challenges and experiences. In *the 12nd International European Conference on Parallel Processing (Euro-Par2006)*, August 29 - September 1 2006.

[87] Ziang Hu, Geoff Gerfin, Brice Dobry, and Guang R. Gao. Programming experience on cyclops-64 multi-core chip architecture. In *First Workshop on Software Tools for Multi-Core Systems (STMCS06), in conjunction with the IEEE/ACM International Symposium on Code Generation and Optimization (CGO2006)*, New York, NY, March 2006.

[88] Christopher J. Hughes, Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, February 2002.

[89] IBM. The CELL project at IBM research. `http://www.research.ibm.com/cell/`.

[90] IBM. IBM system/370 extended architecture, principle of operation. 1983. Publication No. SA22-7085.

[91] Intel News Release. Intel develops tera-scale research chips, September 2006.

[92] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *the 13th annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, pages 151–160, Los Angeles, CA, 1994.

[93] Alain Kägi and Doug Burger James R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, pages 170–180, 1997.

[94] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589 – 604, 2005.

[95] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March/April 2004.

[96] S. Kapil. UltraSPARC Gemini: Dual cpu processor. Hot Chips 15, 2003.

[97] Krishna M. Kavi, A. R. Hurson, Phenil Patadia, Elizabeth Abraham, and Ponnarasu Shanmugam. Design of cache memories for multi-threaded dataflow architecture. In *Proceedings of the 22nd International Symposium on Computer architecture*, 1995.

[98] Stephen W. Keckler, William J. Dally, Daniel Maskit, Nicholas P. Carter, Andrew Chang, and Whay S. Lee. Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor. In *Proceedings of the 25th annual international symposium on Computer architecture*, pages 306–317, Washington, DC, USA, 1998.

[99] Arun Kejariwal, Hideki Saito, Ximin Tian, Milind Gikar, Wel Li, Utpal Banerjee, Alexandru Nicolau, and Constantine D. Polychronopoulos. Lightweight lock-free synchronization methods for multithreading. In *Proceedings of the 20th International Conference on Supercomputing*, Cairns, Australia, June 2006.

[100] Peter M. Kogge. Past predictions, the present, and the future trends, Dec 2006. Invited talk at High Performance Computing Workshop on Programming Languages, Sandia National Labs.

[101] P. Kongetira. A 32-way multithreaded SPARC processor. Hot Chips 16, 2004.

[102] D. Kranz, B. H. Lim, and A. Agarwal. Low-cost support for fine-grain synchronization in multiprocessors. Technical Report MIT/LCS/TM-470, 1992.

[103] V. P. Krothapalli and P. Sadayappan. Removal of redundant dependences in doacross loops with constant dependencies. In *Proceedings of the 1991 Conference on the Principle and Practice of Parallel Programming*, April 1991.

[104] Sanjeev Kumar, Dongming Jiang, Rohit Chandra, and Jaswinder Pal Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):23–34, June 1999.

[105] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance evaluation of the Omni OpenMP compiler. In *the Third International Symposium on High Performance Computing*, pages 403–414, Tokyo, Japan, Octorber 2000.

[106] Vladimir Lanin and Dennis Shasha. Concurrent set manipulation without locking. In *the 7th ACm Symposium on Principles of Database Systems*, pages 211–220, March 1988.

[107] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer architecture*, 1997.

[108] Zhiyuan Li and Walid Abu-Sufah. A technique for reducing synchronization overhead in large scale multiprocessors. In *Proceedings of the 12th Annual International Symposium on Computer Architectures*, pages 284–291, May 1985.

[109] Chunhua Liao, Zhenying Liu, Lei Huang, and Barbara Chapman. Evaluating OpenMP on chip multithreading platform. In *First International Workshop on OpenMP*, Eugene, Oregon USA, June 2005.

[110] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 25–35, New York, NY, USA, 1994. ACM Press.

[111] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive software transactional memory. In *the 19th International Symposium on Distributed Computing (DISC'05)*, Cracow, Poland, September 2005.

[112] Virendra J. Marathe and Michael L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report 839, Department of Computer Science, University of Rochester, Rochester, NY 14627-0226, June 2004.

[113] T. Maruyama. SPARC64 VI: Fujitsu's next generation processor. in Mircroprocessor Forum 2003, 2003.

[114] Henry Massalin and Carlton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

[115] Doug Matzke. Will physical scalability sabotage performance gains? *IEEE Computer*, 30(9):37–39, 1997.

[116] Austen McDonald, JaeWoong Chung, Brian D. Carlstrom, Chi Cao Minh, Hassan Chafi, Christos Kozyrakis, and Kunle Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[117] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of tcc on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Sept 2005.

[118] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization onshared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[119] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.

[120] Maged M. Michael. CAS-based lock-free algorithm for shared deques. In *the 9th Euro-Par Conference on Parallel Processing*, pages 651–660, August 2003.

[121] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC23089 (W0401-136), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, January 2004.

[122] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst*, 15(6):491–504, 2004.

[123] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.

[124] S. P. Midkiff and D.A. Padua. Compiler algorithms for synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, Dec 1987.

[125] Mark Moir. Transparent support for wait-free transactions. In *the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997.

[126] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(18), April 19, 1965.

[127] J.H. Moreno, M. Moudgill, J.D. Wellman, P. Bose, and Louise Trevillyan. Trace-driven performance exploration of a PowerPC 601 OLTP workload on wide superscalar processors. In *First Workshop of Computer Architecture Evaluation using Commercial Workloads*, Las Vegas, NY, Feb. 1 1998.

[128] Newlib. Newlib. `http://sources.redhat.com/newlib/`.

[129] M. F. P. O'Boyle, L. Kervella, and F. Bodin. Synchronization minimization in a SPMD execution model. *J. Parallel Distrib. Comput.*, 29(2):196–210, 1995.

[130] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Kenneth G. Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, pages 2–11, 1996.

[131] OpenMP Architecture Review Board. OpenMP FORTRAN application program interface. Technical Report 2.0, OpenMP Architecture Review Board, November 2000. In http://www.openmp.org/specs.

[132] OpenMP Architecture Review Board. OpenMP C and C++ application program interface. Technical Report 2.0, OpenMP Architecture Review Board, March 2002. In http://www.openmp.org/specs.

[133] OpenMP Architecture Review Board. OpenMP C and C++ application program interface. Technical Report 2.5, OpenMP Architecture Review Board, May 2005. In http://www.openmp.org/specs.

[134] Krzysztof Parzyszek, Jarek Nieplocha, and Ricky A. Kendall. General portable SHMEM library for high performance computing. In ACM, editor, *SC2000: High Performance Networking and Computing*, pages 148–149, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, November 2000. ACM Press and IEEE Computer Society Press.

[135] Daniel Gracia Pérez, Gilles Mouchard, and Olivier Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 43–54, Portland, OR, Dec. 4–8, 2004. IEEE-CS TC-MARCH and ACM SIGMICRO.

[136] Achal Prabhakar, Vladimir Getov, and Barbara Chapman. Performance comparisons of basic OpenMP constructs. In Hans P. Zima, Kazuki Joe, Mitsuhisa Sato, Yoshiki Seo, and Masaaki Shimasaki, editors, *Proceedings of the 4th International Symposium on High Performance Computing*, number 2327, pages 413–424, Kansai Science City, Japan, may 15–17, 2002.

[137] The DARPA High Productivity Computing Systems (HPCS) Program. HPC challenge benchmark. `http://icl.cs.utk.edu/hpcc/`.

[138] Ramakrishnan Rajamony and Alan L. Cox. Optimally synchronizaing DOACROSS loops on shared memory multiprocessors. In *Proceedings of 1997 International Conference on Parallel Architectures and Compiliation Techniques*, 1997.

[139] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *the 34th International Symposium on Microarchitecture*, pages 294–305, Dec 2001.

[140] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Tenth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 5–17. Oct 2002.

[141] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505. IEEE Computer Society, Jun 2005.

[142] David Ródenas, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, George Almási, Călin Caşcaval, José Castaños, and José Moreira. Optimizing NANOS openMP for the IBM cyclops multithreaded architecture. In *19th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, USA, April 2005.

[143] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *International Symposium on Computer Architecture*, pages 340–347, 1984.

[144] Jack Sampson, Ruben Gonzalez, Jean-Francois Collard, Norm Jouppi, Mike Schlansker, and Brad Calder. Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In *Proceedings of the International Symposium on Microarchitecture (Micro 2006)*, 2006.

[145] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *the 24th ACM Symposium onf Principles of Distributed Computing (PODC'05)*, Las Vegas, Nevada, July 2005.

[146] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[147] Burton Smith. The architecture of HEP. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, Scientific Computation Series, pages 41–55. MIT Press, Cambridge, MA, 1985.

[148] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the tera mta. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–8, Washington, DC, USA, 1998. IEEE Computer Society.

[149] SPEC. All SPEC CINT2000 results. `http://www.spec.org/cpu2000/results/cint2000.html`.

[150] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *the 11th International Symposium on High-Performance Computer Architecture (HPCA'05)*, 2005.

[151] Vugranam C. Sreedhar, Yuan Zhang, and Guang R. Gao. A new framework for analysis and optimization of shared memory parallel programs. *CAPSL Technical Memo 63*, July 18th 2005.

[152] Guangming Tan, Ninghui Sun, and Guang R. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *Proceedings of 19th ACM Symposium on Parallelism in Algorithms and Architectures*, Jun. 2007.

[153] Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. Performance evaluation of OpenMP applications with nested parallelism. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, 2000.

[154] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apirl 1986.

[155] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *the 5th Intl. Symp. on High-Performance Computer Architecture*, pages 54–58, Orlando, Florida, January 9–13, 1999.

[156] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *the 5th Intl. Symp. on High-Performance Computer Architecture*, pages 54–58, Orlando, Florida, January 9–13, 1999.

[157] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, Istanbul, Turkey, Nov. 18–22 2002. IEEE-CS TC-MARCH and ACM SIGMICRO.

[158] J. D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[159] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposiumon Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, 2–23 August 1995.

[160] S. Vangal, J. Howard, G. Ruhl, and et. al. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of 2007 International Solid-State Circuits Conference*, Feb. 2007.

[161] I. E. Venetis and G. R. Gao. Optimizing the LU Benchmark for the Cyclops-64 Architecture. CAPSL Technical Memo 75, University of Delaware, February 2007.

[162] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

[163] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[164] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in mimd machines for preconditioned conjugate gradient. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–192, New York, NY, USA, 1993. ACM Press.

[165] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–192, 1993.

[166] L. Zhang, Z. Fang, and J. B. Carter. Highly efficient synchronization based on active memory operations. In *Proceedings of 18th International Parallel and Distributed Processing Symposium*, 2004.

[167] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao. Optimized lock assignment and allocation for productivity: A method for exploiting concurrency among critical sections. *CAPSL Technical Memo 65*, May 10th 2006.

[168] Yuan Zhang, Weirong Zhu, Fei Chen, Ziang Hu, and Guang R. Gao. Sequential consistency revisit: the sufficient condition and method to reason the consistency model of a multiprocessor-on-a-chip architecture. In *International Conference of Parallel and Distributed Computing and Networks (PDCN2005)*, Innsbruck, Austria, February 2005.