University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Efficient Fine-Grain Synchronization on a Multi-Core Chip Architecture: A Fresh Look

Weirong Zhu†
Ziang Hu
Guang R. Gao

CAPSL Technical Memo 67
July 17, 2006

†Email: weirong@capsl.udel.edu

Abstract

Multi-core chip architectures are becoming mainstream, permitting increasing on-chip parallelism through hardware support for multithreading. Fine-grain synchronization is essential to the effective utilization of the capacity provided by future high-performance multi-core architectures. However, there are also new challenges realizing such fine-grain synchronization in large-scale multi-core chip architectures – such as the IBM Cyclops-64 chip that contains more than 100 processing cores and employs a memory organization with explicitly addressable memory segments instead of data cache.

This paper presents a fresh look at the challenges and proposes a scalable solution for fine-grain synchronization that efficiently enforces *mutual exclusion* and *read-after-write data-dependencies* between concurrent threads. Using the Cyclops-64 chip architecture as a case study, we illustrate how to use a small *Synchronization State Buffer* (SSB) associated with each memory bank to accelerate the fine-grain synchronization by recording and managing the states of frequently synchronized data units with modest hardware extensions. We demonstrate the effectiveness and efficiency of the proposed solution.

- For *mutual exclusion*: Using distributed fine-grain locking at each of the memory units, we avoid the unnecessary serialization of operations on different elements of the same concurrent data structure and achieve this goal efficiently.
- For *read-after-write data-dependencies synchronization*: our method encourages the exploration of do-across style of loop-level parallelism - where *loop-carried* data dependencies can often be directly implemented by the application of the fine-grain synchronization operations and the removal of useless barriers.

The experimental results demonstrate significant performance gain due to the use of the above fine-grain synchronization solutions.

Contents

List of Figures

List of Tables

1 Introduction

As an alternative to the conventional single-thread wide-issue superscalar processor, the design of high-performance chip architectures is now rapidly moving towards the multi-core approach that integrates an increasing number of tightly-coupled processing cores on a single chip. As advances in IC processing technology have led to hundreds of millions (now reaching 1 billion and beyond) of transistors being fabricated on a single silicon die, it becomes possible to construct large-scale multi-core chip architectures. 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 [17].

In order to fully utilize the on-chip parallelism provided by large-scale multi-core chip architectures, it is important to exploit the fine-grain parallelism that is available in applications. It is well understood that the granularity of parallelism that can be exploited on a multiprocessor machine is determined by the synchronization mechanisms provided [13], which is also applicable to multi-core architectures.

It now appears that the parallel system community converges towards the opinion that fine-grain synchronization is essential to the effective exploitation of fine-grain parallelism on utilization of the capacity provided by future high-performance multi-core architectures. On large-scale multi-core chips - such as the IBM Cyclops-64 chip that contains well over 100 hardware thread units - the on-chip storage available per processor is far less (often 1-2 orders less) than traditional single core microprocessors. Therefore, a new memory organization may be employed that will provide explicitly addressable memory segments (e.g. Scratchpad memory, on-chip SRAM, etc.) instead of data cache. 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 multi-core architectures:

- Such solutions should be scalable and can fully exploit the parallelism due to the distributed on-chip resources.
- Such solutions should be supported with limited on-chip resources – unlike previous proposals (e.g. full/empty bits in Tera memory systems [6], or I-Structure [9] support in dataflow architectures, where in-memory synchronization is usually realized by enhancing the entire memory of the machine).
- Such solutions should incur low synchronization overhead.
- Such solutions should be able to support a variety of synchronization functionalities.
- Such solutions should be able to smoothly and efficiently handle the cases where the precise synchronization point cannot be resolved statically at compile time.

This paper presents a fresh look at those challenges and proposes a scalable solution for supporting fine-grain synchronization on large-scale multi-core chip architectures. Using the IBM Cyclops-64 chip architecture as a case study, we illustrate a novel architectural extension called *synchronization state*

buffer (SSB), that efficiently enforces *mutual exclusion* and *read-after-write data-dependencies* between concurrent threads with fine-grain synchronizations.

Hardware support for fine-grain synchronization has been explored in several architectures built or proposed before. HEP [39], Tera [6], MDP [14], Alewife [2, 25], M-Machine [24], the MT processor in Eldorado [18], 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. For large-scale multi-core chips, on-chip memory is one of the most precious resources. The approach that tags each word of the entire memory requires modification to off-the-shelf SRAM or DRAM technologies and introduces significant on-chip and off-chip storage cost. Because of such cost, the number of state bits that can be tagged to a memory word has to be small, which can only be used to implement limited synchronization functionalities.

In this paper, we explore an alternative solution, which is motivated by the observation that the number of fine-grain synchronized memory locations that are active (in use) at any moment is far less than the number of words in the entire memory. Furthermore, for both mutual exclusion (e.g. lock/unlock) and data synchronization (e.g. synchronized write/read), which are of interest in this paper, the associated states of synchronized memory location(s) can be naturally released when the synchronization is completed (e.g. via unlock or synchronized reads).

To this end, we introduce a novel hardware extension to C64-like large-scale multi-core architectures. To simplify the hardware design and avoid enormous on-chip memory storage cost but still create an illusion that each word in memory is associated with a set of states, our approach only attaches a small *synchronization state buffer* (SSB) to the memory controller of each memory bank. This small SSB caches the access states of memory locations that are currently accessed by SSB synchronization operations. 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 [8, 12, 26, 27, 34, 36, 37] 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 SRAM banks, etc.), can take full advantage of ample on-chip interconnection bandwidth. Also, previous studies [25, 41] 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.

In order to evaluate the efficiency of this method, we (1) extend the IBM Cyclops-64 architecture simulator with the new SSB architectural features, (2) design 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 the proposed fine-grain synchronization method.

- 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 65% performance improvement has been observed for the benchmarks we tested.

- For read-after-write dependence synchronization: our method encourages the exploration of doacross 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 449% 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 multitheading programs, and 2) most of fine-grain synchronizations are successful.

2 Cyclops-64 Large-Scale Multi-Core Chip Architecture

The Cyclops-64 (C64) [17] is a petaflop supercomputer project under development at IBM T.J. Watson Laboratory. It is designed to serve as a dedicated petaflop compute engine for running high performance scientific and engineering applications, such as molecular dynamics to study protein folding [5], or image processing to support real-time medical procedures.

2.1 C64 Chip Architecture

The C64 chip architecture (see Figure 1) employs a large-scale multi-core-on-a-chip design by integrating 160 hardware threads units, and the same amount of embedded SRAM memory banks in a single silicon chip. A C64 chip has 80 processors, each with two thread units (TU), a floating-point unit (FP) and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The basic unit of memory, a word, in C64 is 8 bytes long. The C64 chip architecture represents a major departure from mainstream microprocessor design in several aspects:

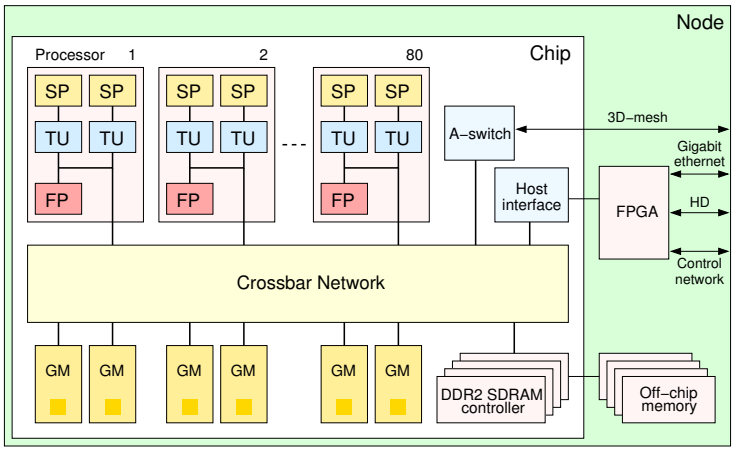


Figure 1: Cyclops-64 Chip Architecture

1. The C64 chip integrates a large number of (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 incorporates efficient support for thread level execution. For instance, a thread can 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. All the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers.

4. 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 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. There are 4 off-chip memory controllers connected to 4 off-chip DRAM banks. The current design size for DRAM is 1GB.

5. C64 also employs the Network-on-Chip (NoC) concept. All on-chip resources are connected to an on-chip crossbar network, which sustains a 4GB/s bandwidth per port per direction, which results in 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 [42].

6. C64 provides no resource virtualization mechanisms: the thread execution is *non-preemptive* and there is no hardware virtual memory manager. The former means the OS will not interrupt the

user thread running on a thread unit unless the user explicitly specifies termination or an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is *visible* to the programmer.

A maximum configuration of a C64 system consisting of 13,824 C64 chips, connected by a 3D mesh network, is expected to achieve over 1 petaflop peak performance. To the best of our knowledge, the C64 project is one of the most ambitious petaflop supercomputer projects currently under active development. A first C64 system is planned to be installed in 2007.

2.2 Synchronization Mechanisms in Current C64 Design

Several synchronization mechanisms have been implemented for current C64 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 [30]. 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) [11], *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 [15].

3 Efficient Fine-Grain Synchronization on Cyclops-64

In this paper, we propose a novel architectural extension called *synchronization state buffer* (SSB) to large-scale multi-core chip architectures, like C64. 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.

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. The design of SSB will be elaborated in following subsections.

3.1 Structure of SSB

We now take the C64 chip architecture as an example to explain the design. SSB, a special hardware buffer, is embedded into the memory controller of each memory bank, which caches the states for memory locations accessed by SSB instructions. This buffer realizes the lookup-table function in a single clock cycle, which can be implemented with the common cache technology. As an experimental design, we associate each SRAM bank with an 8-entry SSB, and each DRAM bank with a 1,024-entry SSB, both of which are 8-way set associative. In practice, the number of entries, and the level of set associativity can be adjusted according to the transistor budget, and the power consumption requirements, etc.

state	counter	thread id	address
4 bits	8 bits	8 bits	N bits

Figure 2: One SSB Entry

Each entry of the SSB is used to record and manage the states of a memory word that is accessed by SSB instructions at runtime. Because of the small storage cost, each SSB entry can afford to use a number of bits (for example, 20 bits in our current design) to encode the synchronization states. Thus, SSB can effectively support a variety of synchronization functionalities. The structure of one entry is shown in Figure 2. The address is the *key* used for the search in the SSB. The rightmost N-bits holds the address of a memory location. The number N is determined by the number of bits needed to identify a unique memory address in the corresponding memory bank. In our design for current C64 chip, 13 bits and 25 bits is chosen for SSB on SRAM bank and on DRAM bank respectively. The remaining fields compose the states of the corresponding memory location. The next 8-bits is the "thread id" field, which is used to record the ownership information of the entry. An 8-bit field supports up to 256 threads, while C64 currently has 160 thread units. Since the thread execution in C64 is non-preemptive, the "thread id" can be used to identify a hardware thread unit as well as a unique software thread running on it. The next 8-bits is a "counter" field. Its usage differs for various groups of SSB operations, which will be explained later. The leftmost 4-bits can keep up to 16 types of modes for the entry to support different SSB operations. Table 1 shows the meaning of the modes currently proposed in the design. An entry in SSB is allocated and released according to its state and the function of the SSB instruction operating on it.

An SSB instruction is treated the same way as other memory instructions by the C64 on-chip crossbar network, which delivers all memory requests to the destined memory bank. Upon arriving at a particular memory bank, SSB instructions as well as other memory instructions, are served based on a FIFO discipline.

Table 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	Linked-Load and Store-Conditional Mode

3.2 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, a 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.

3.3 Support for Fine-Grain Locking

In order to achieve fine-grained synchronization, SSB provides highly flexible dynamic locking functionality. 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.

3.3.1 A Motivating Example

Figure 3 shows how the kernel loop in Table Toy (also known as Random Access) [1] can be implemented with critical section in OpenMP. Although it is not strictly required by the original benchmark, for the purpose of illustration, we use critical section to guarantee the read-modify-write operations

```
#pragma omp parallel for private(idx,i,tmp) shared(x,y,N)
for(i = 0; i < N; i++){
   idx = rand();
   tmp = x[i];
#pragma omp critical
   {
     y[idx] = y[idx] op tmp;
   }
}
```

Figure 3: Example: Table Toy

in the loop to be performed atomically. Unstructured references, subscripted subscripts like those in Figure 3, are the hallmark of irregular applications [28]. Unstructured references like the one in the above critical section are impossible to analyze at compile time. As a consequence, the compiler can only assign a single lock in this case. Given a large enough table $y[]$ (much larger than the number of threads) and a high quality uniform random number generator, the chance of conflicts to access the same $y[idx]$ is very low. That is, the application itself inherently has enough parallelism, which can not be exploited because of the serialized execution of instances of the critical section. The dynamic locking functionality of SSB helps exploit the parallelism by avoiding the serialization.

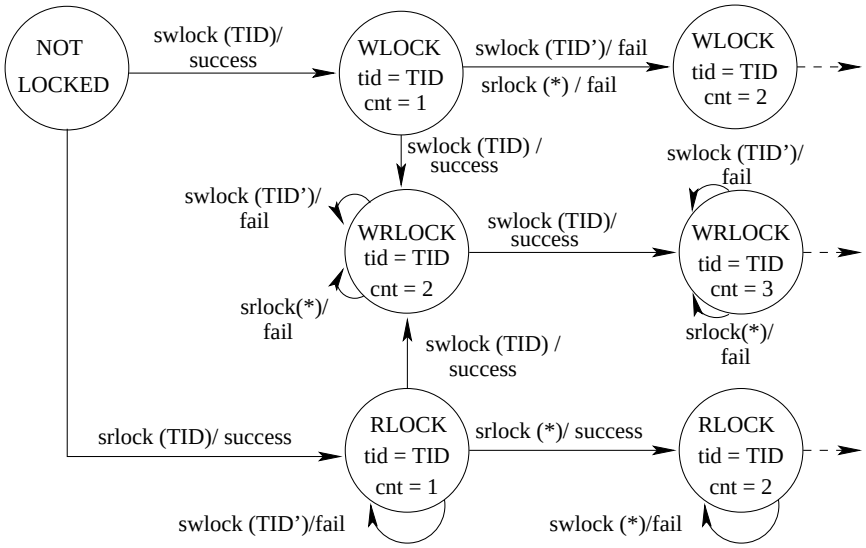
3.3.2 Implementation of Fine-Grain Lock

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

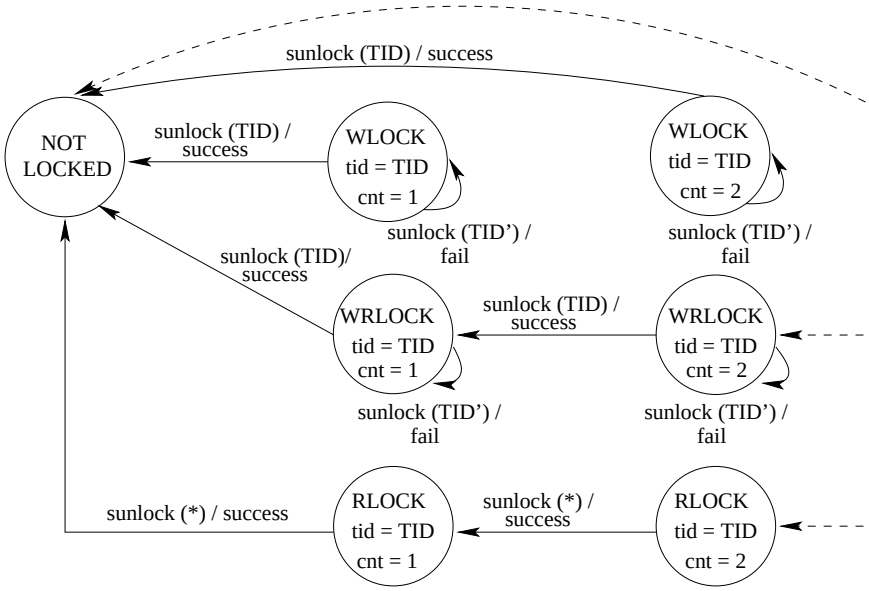
```
(RT, Value) = swlock_l(MemAddr);
/* swlock_l: acquire write lock for location MemAddr */
/*           and load the content                    */
/* MemAddr: the address of the memory location       */
/* RT: return value, success or failure              */
/* Value: the content of the memory location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

(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      */
```

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 4 illustrates how the lock/unlock operations interact with the SSB hardware.

As shown in Figure 4(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



(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 4: State transition diagram of SSB lock/unlock operations.

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 4(b). When a lock is finally released, the corresponding entry in SSB will be freed for reuse.

3.4 Support for Fine-Grain Data Synchronization

In C64, 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.

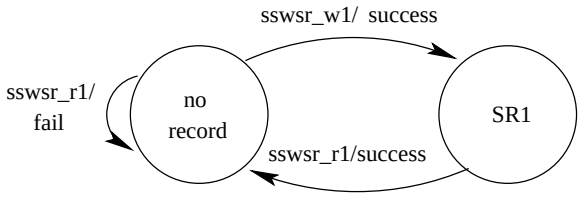
3.4.1 Single-Writer-Single-Reader (SWSR) Data Synchronization

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 :

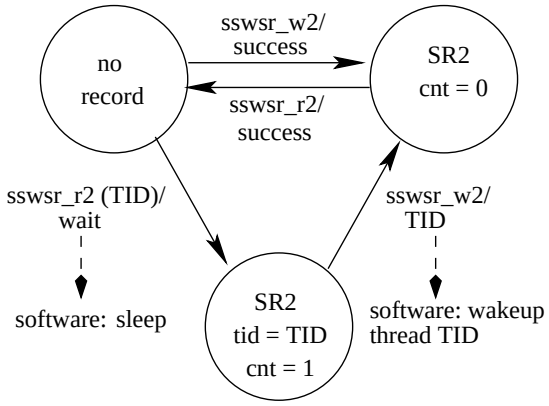
```
RT = sswrsr_w1(MemAddr, Value);
/* sswrsr_w1: SWSR synchronized write mode 1    */
/* MemAddr: the address of the memory location  */
/* Value: the Value to be written to MemAddr    */
/* RT: return value, success or failure         */

(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 write 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(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.



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

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 C64. As illustrated by Figure 5(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 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.

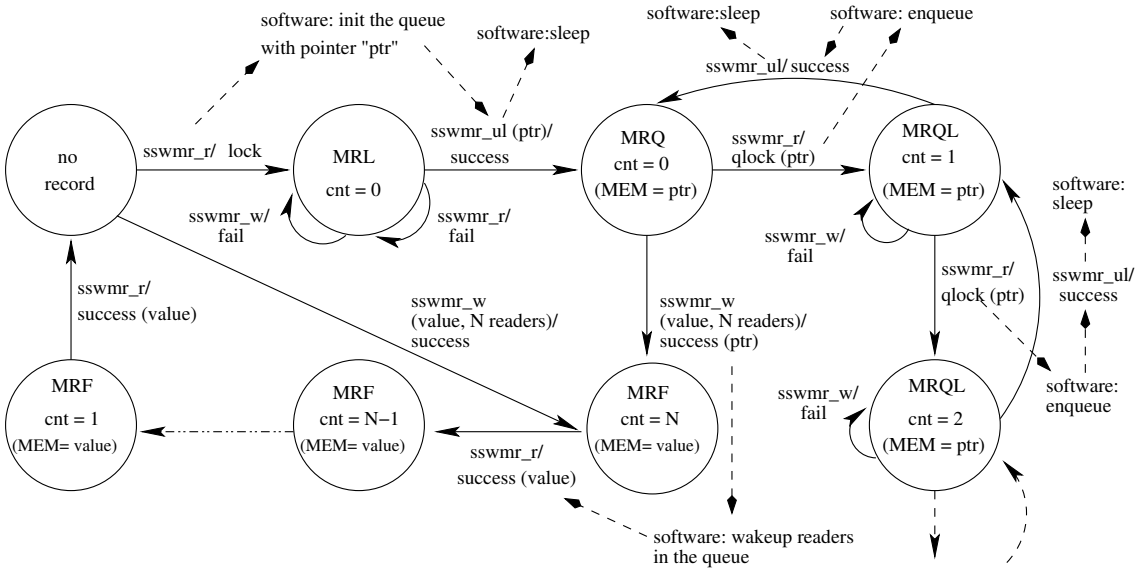
3.4.2 Single-Writer-Multiple-Reader (SWMR) Data Synchronization

enforces ordering between a thread that produces the data and a number of other threads that consume the data. The following are the interfaces:

```
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 6 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



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

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

3.5 Other Design Issues

3.5.1 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” [33]. It is apparent that it is straightforward to support the LL/SC instructions on C64 using SSB to monitor the states of memory location accessed by LL/SC. For the synchronization operations introduced in 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.

3.5.2 Handling Hardware Resource Limitation

Given a memory bank, the size of the corresponding SSB is limited. It is possible that the destination set in the SSB is full when an SSB instruction is executed. In such a case, two different mechanisms are provided. For the first one, an indication of failure is returned to the calling thread, which may decide to retry afterwards. The other mechanism is based on exception handling to resolve this limitation of hardware resource, if the trap bit in the opcode of the instruction is set. Exception handler manages a software table for each set of SSB as an extension to the hardware SSB. There is a hardware bit associated with each set in the SSB to indicate whether there are software maintained entries or not. When the corresponding set in SSB is full, or there is no matching entry in hardware SSB but the bit is on, the exception will be triggered. It is apparent that the exception handling will slow down the requested synchronization operation. However, it is expected that the hardware synchronization resource provided by SSB is normally sufficient for most of multithreading programs. As to be shown in Section 4.3, for all benchmarks we tested, only one benchmark has 0.001% synchronization operations that encounter the “full” situation.

4 Experimental Results

The experiments are conducted on the C64 FAST simulator [15], which is an execution-driven, binary-compatible simulator of a multi-chip C64 system. It accurately models the functional behavior of hardware components in a C64 system. In addition, it generates timing information that accounts for the main sources of pipeline delays and stalls such as contention in memory, the crossbar, and/or other functional units. FAST has been extensively used by the C64 architecture design team at IBM for the purpose of chip design verification, and dozens of system software developer and application scientists for early application development. More details of the simulator are given in [15]. The SSB extension to C64 is implemented in the simulator. SSB instructions that require return values have the same latency as a load instruction, otherwise as a store instruction. Currently, multithreading programs for C64 can be coded with either the Pthread-like TiNy Threads (TNT) API [16] or OpenMP.

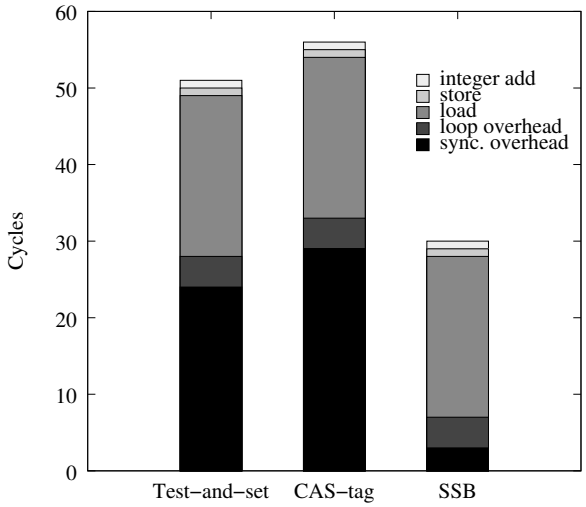


Figure 7: Overheads of Synchronization Mechanisms

4.1 Characterization and Performance of Fine-Grain Locks

4.1.1 Synchronization Overhead

One of the design criteria of SSB is that the cost of a successful synchronization operation should be very small. To demonstrate this, we measure the overhead of different synchronization mechanisms with a microbenchmark. In the microbenchmark, a reference time is obtained by executing a 10,000 iteration loop sequentially without using any synchronization. Each iteration of the loop loads a 64-bit long integer from the on-chip SRAM, performs a simple arithmetic operation (add), and stores it back to the memory. Then the overhead is calculated by comparing this reference time with the execution time of the same code extended with synchronization operations. For using a test-and-set spinlock, a lock has to be acquired before accessing the memory location. After the operation on the location finishes, the lock is released. 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 is not changed since the last load. IBM tag methodology [22] is used for ABA-prevention [22, 32], which adds extra overhead and complicates the data structure. The microbenchmark is executed on a single thread. Therefore, there is no contention and all the synchronization operations (lock acquisition or CAS commitment) are always successful. Figure 7 shows the results of our measurements. Upon successful synchronization, SSB-based operations incur the lowest overhead among all the mechanisms.

4.1.2 Exploit Fine-Grain Parallelism of Application Kernels

Efficient fine-grain locking mechanisms can help to exploit the inherent parallelism within applications, especially when the precise synchronization point cannot be resolved statically at programming/compiling time. In this subsection, we examine 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 [1], two of the Livermore loops, and a hash-table based implementation of ordered integer set.

Table Toy. As shown in Figure 3, the address of the memory location to be mutually exclusively accessed is only known right before entering the critical section. In this case, if a conventional spin-lock is used, the programmer or the compiler normally assigns a single lock to the whole array, which serializes the execution. One possible solution is to allocate an array of locks, whose size is exactly the same as the $y[]$ array. Therefore, 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 at least doubles the memory usage, which is normally not acceptable. By using the SSB lock operations, the programmer/compiler 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 to memory usage.

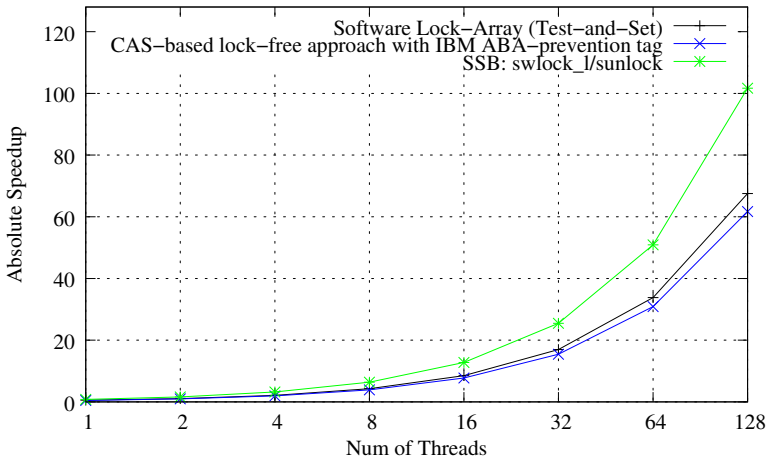


Figure 8: Speedup of Table Toy parallelized with different synchronization mechanisms

Figure 8 compares three parallelization scheme of Table Toy using different 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 makes use of IBM tag methodology to prevent the ABA problem and presents the worst performance among the three. 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 50.6% and 64.9% respectively without any extra memory usage.

Livermore Loops. Because of the cross-iteration dependencies (and the dependence distance can not be determined statically), Livermore Loops 13, and 14 can not be easily parallelized. Within each iteration, certain members of an array are updated. However, the calculation of the indices is unpredictable and data-dependent. Since it is not necessary to preserve the order of those updates, we use locks to guarantee that the runtime-determined member of the array is updated mutually exclusively.

Figure 9 and 10 compares two approaches that attempt to parallelize the two loops. The coarse-grain approach serializes the updates to the array using a spin-lock (the MCS lock [30] used here) to ensure

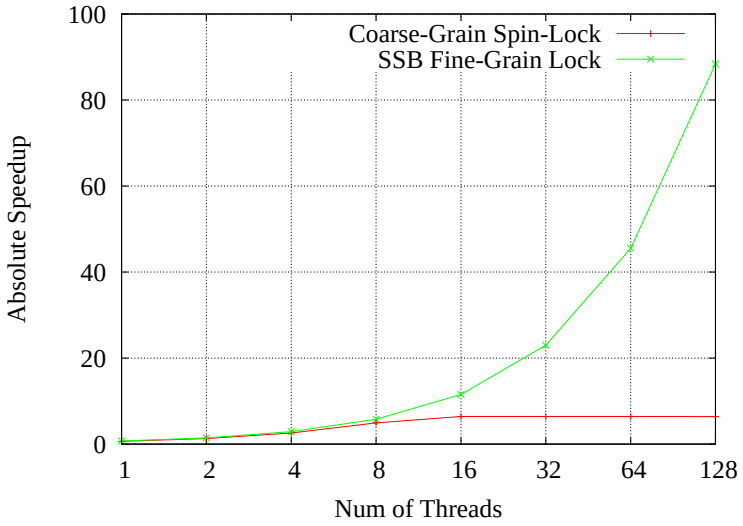


Figure 9: Speedup of Livermore Loop 13 parallelized with different synchronization mechanisms

mutual exclusion. The fine-grain approach makes use of the SSB lock instructions to lock the location to be updated individually. With SSB, the iterations that do not access the same location do not contend with each other. Figure 9 and 10 shows that the coarse-grain approach does not scale well because of the serialization of the updates to the arrays. 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. Figure 9 and 10 demonstrates that the fine-grain approach can achieve a speedup of 88.4 and 75.2 on 128 threads for Loop 13 and Loop 14 respectively.

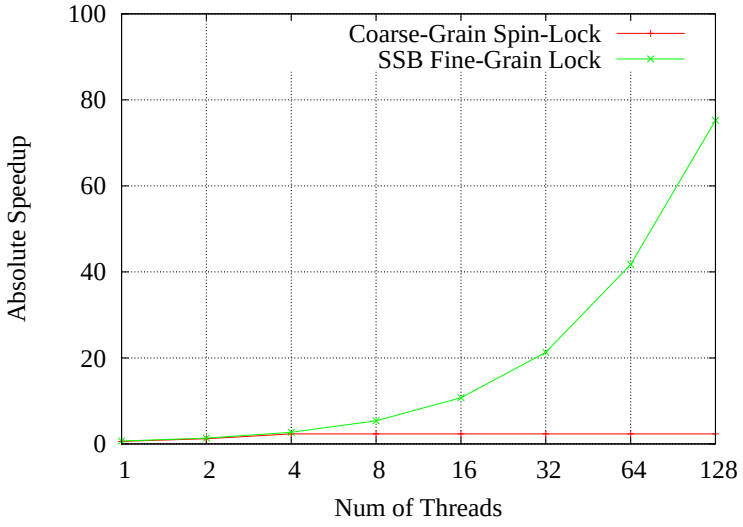


Figure 10: Speedup of Livermore Loop 14 parallelized with different synchronization mechanisms

Hash Table Based Ordered Integer Sets. Hash table is a common data structure widely used in system

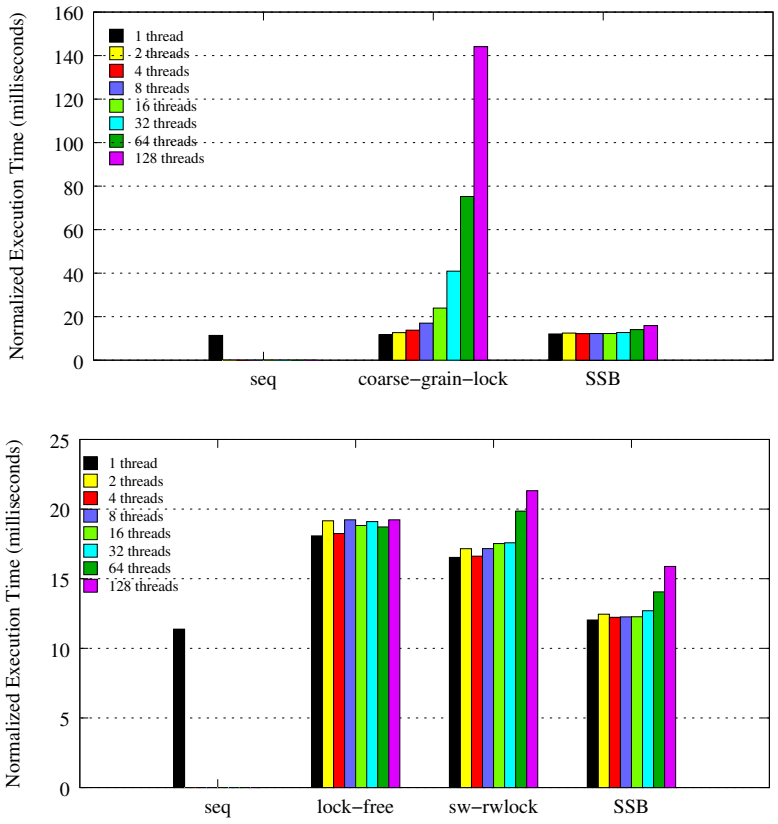
programs as well as applications as a search structure. In this study, the hash table is used to implement an ordered integer set. The hash table has multiple buckets, each managing an ordered linked list. Given an integer key k, the hash function $h(k)$ determines the bucket, where the key might be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

- *Coarse-grain lock based version:* each bucket is protected by a spin-lock.
- *Lock-free version:* uses Michael's lock-free hash table algorithm [31]. The *hazard pointers* mechanism is used to guarantee safe memory reclamation of lock-free objects as well as ABA-safe [33].
- *sw-rwlock version:* uses software based read and write locks. A lock variable is added into the data structure of the node in 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 there is no extra memory usage.

To evaluate the performance of these implementations, a microbenchmark is used. 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 11 shows that the SSB based version achieves best performances 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. The other three fine-grain versions show near constant execution time even when the number of threads reaches 128.

With SSB instructions, there is no synchronization overhead when there is no contention. The lock-free version, however, always needs to check the return value of the CAS instruction, when committing the result to the memory. Therefore, even without contention, the CAS based lock-free implementation incurs synchronization overhead. The lock-free version also needs to pay certain cost for the safe memory reclamation. The acquisition of the software-based lock of the sw-rwlock version can not avoid synchronization overhead either. As shown in Figure 11, when running on a single thread (i.e., there is no contention), the lock-free version and sw-rwlock version are 58% and 45% slower than the sequential version respectively, while the SSB -based version is only 6% slower. In all cases, the SSB version is at least 17% and up to 36% faster than the other two versions without any extra memory usage.



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

Figure 11: Implement Hash Table based integer set with different synchronization mechanisms.

4.2 Characterization and Performance of Fine-Grain Data Synchronization

4.2.1 Synchronization Overhead

Just like the SSB locking operations, the SSB data synchronization operations incur very low overhead upon successful synchronized write and read. We also use a microbenchmark to measure the overhead of the SSB data synchronization operations. In the microbenchmark, a reference time is obtained by executing a loop of 10,000 iterations with 2 threads. Each iteration contains a barrier operation. One thread performs a store operation before the barrier, and the other one performs a load operation after the barrier. Then 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. As shown in Table 2, the overhead of SSB data synchronization operations are very small when performed successfully. The overhead mainly comes from the code that checks and handles the return value of the synchronization operations.

Table 2: Overhead of successful SSB data synchronization operations

SSB Operations	Overhead (cycles)
sswsr_w1/sswsr_r1	3
sswsr_w2/sswsr_r2	5
sswmr_w/sswmr_r	6

4.2.2 Exploit Fine-Grain Parallelism of Application Kernels

To evaluate the performance of fine-grain data synchronization with the proposed architectural support, we monitored the performance of two representative application kernels: linear recurrence equations, and wavefront computation. We demonstrate how these kernels can be parallelized to exploit fine-grain parallelism, with co-operation between hardware and software.

Linear Recurrence Equations (Livermore Loop6).

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

Figure 12: Livermore Loop 6

Livermore loop 6 (Figure 12) represents the general linear recurrence equations, which are widely used in linear algebra computations. As shown in Figure 13, the outer loop computes an array W. Iteration i computes $W[i]$, which depends on $W[0]$,$W[1]$, ... , $W[i-1]$. As a result, each iteration depends on all previous iterations. The cross-iteration dependencies of array W makes it difficult to parallelize this loop.

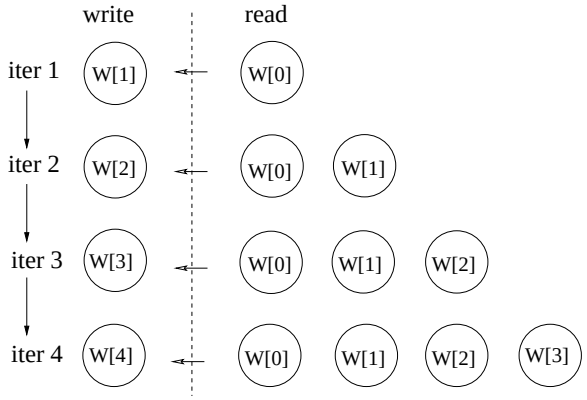
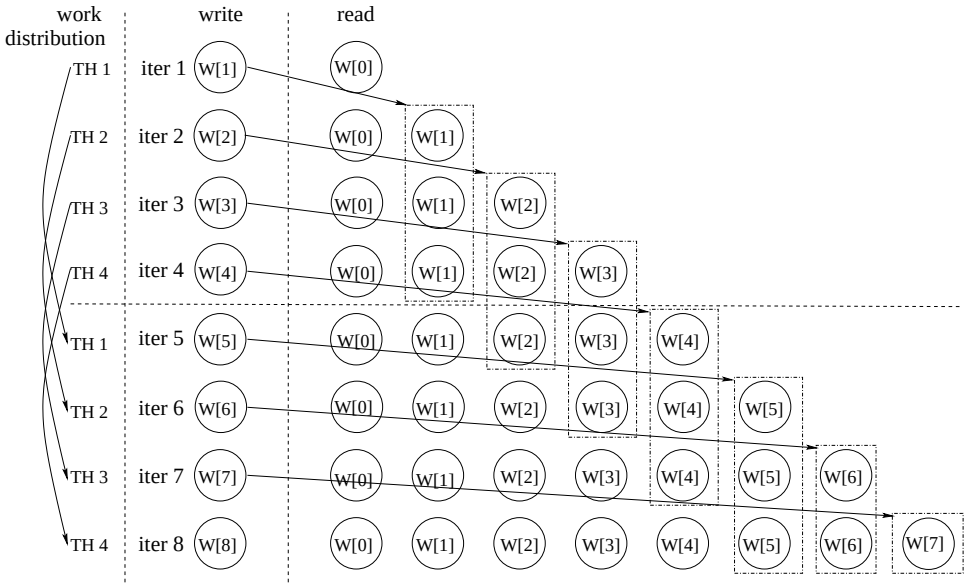


Figure 13: Characteristics of Livermore Loop 6

We parallelize the loop by assigning the iterations to different threads through a round-robin fashion.

The SSB data single-writer-multiple-reader data synchronization mechanism is used to enforce the read-after-write dependencies among iterations.

Our parallelization and synchronization strategy is shown in Figure 14, which illustrates the case that 8 iterations are concurrently executed by 4 threads, and the chunk size of round-robin scheduling is 1 iteration. During the computation, when thread 1 completes iteration 1, it notifies threads 2, 3, and 4 about the availability of $W[1]$ such that they can perform their computation for iteration 2, 3, 4. Then thread 1 moves to 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 actually need to wait for $W[1]$, because $W[1]$ is computed by thread 1 itself earlier. 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 indicating num_threads $-$ 1 readers. When a thread begins a new iteration i to compute $W[i]$, it uses a normal load operation 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 depend on the number of threads.



4 threads, round-robin scheduling, chunk size = 1.

Figure 14: Parallelization and Synchronization of Livermore Loop 6

Figure 15 compares the fine-grain data synchronization based approach with a coarse-grain based one for computing a W array with size 5,120. Instead of the outer i loop, the coarse-grain synchronization based scheme only parallelizes the inner k loop. Since there is no cross-iteration dependence for the inner loop, all iterations are independent. For iteration i of the outer loop, we distribute the computation of the inner loop to multiple threads. Each thread completes its task, reduces its local sum

to the $W[i]$, then waits on a barrier [1], which ensures that all threads completes the task before starting the next iteration $i + 1$. As shown in Figure 15, by exploiting fine-grain parallelism, the fine-grain data synchronization based approaches are always better than the coarse-grain based one when running on a large number of threads. For example, when 128 threads are used, the fine-grained approach with a chunk size of two iterations for the round-robin scheduling achieves an absolute speedup of 68, which demonstrates a 449% improvement over the coarse-grained parallelization scheme.

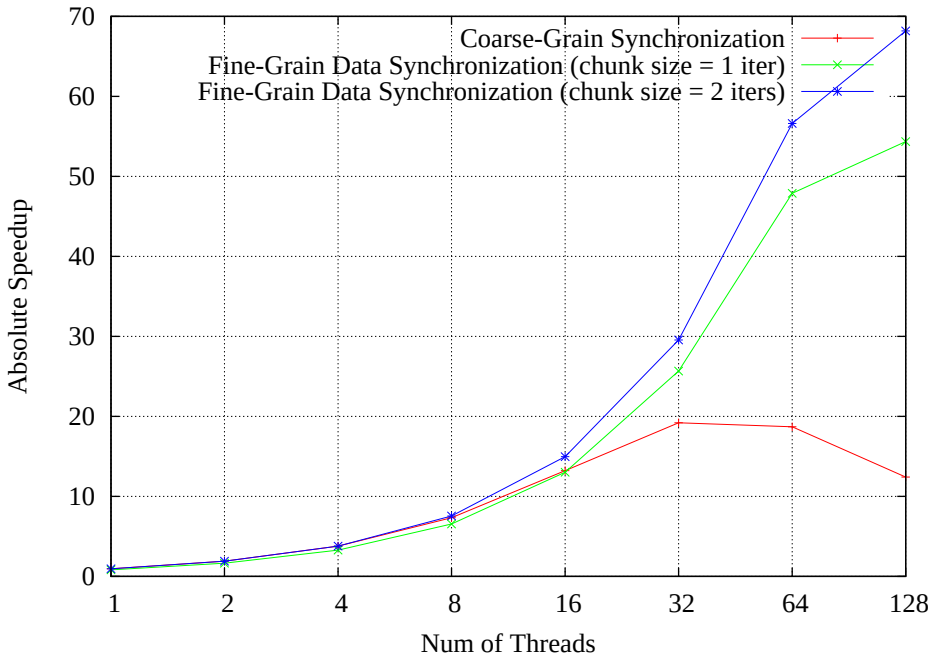


Figure 15: Speedup of Parallelized Livermore Loop 6

Wavefront Computation. Wavefront computations are common in scientific applications. As shown in Figure 16, given a matrix, 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. For a parallel computation, it is natural to use data synchronization to enforce the data dependencies between threads.

In our implementation, the rows of the matrix are assigned to threads in a round-robin fashion. In this parallelization strategy, to compute an element, only the availability of its above neighbor needs to be checked. 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 (sswsr_w2), 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 (sswsr_r2) to get the first element of the block, the remaining elements of the block are read with normal load instruction. The usage of sswsr_w2 and sswsr_r2 instructions are described in Section 3.4.1. Figure 17 shows the speedup of our

[1] It is worth noting that hardware-based barrier on C64 is very efficient. The measurement from our microbenchmark shows that it only takes 500 cycles for 128 threads to join and leave a barrier.

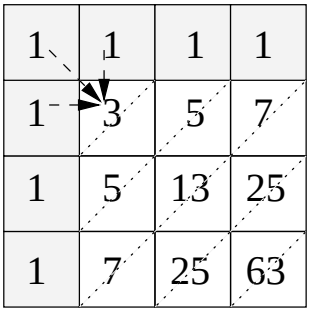


Figure 16: Characteristics of Wavefront Computation

parallelization of the wavefront computation on a 4096×4096 [2] matrix. Although the data dependencies in wavefront computation implies serialization, the multithreaded implementation with fine-grain data synchronization demonstrates the capability to exploit the parallelism. When running with 128 threads, the SSB -based implementation shows an absolute speedup of 104.8.

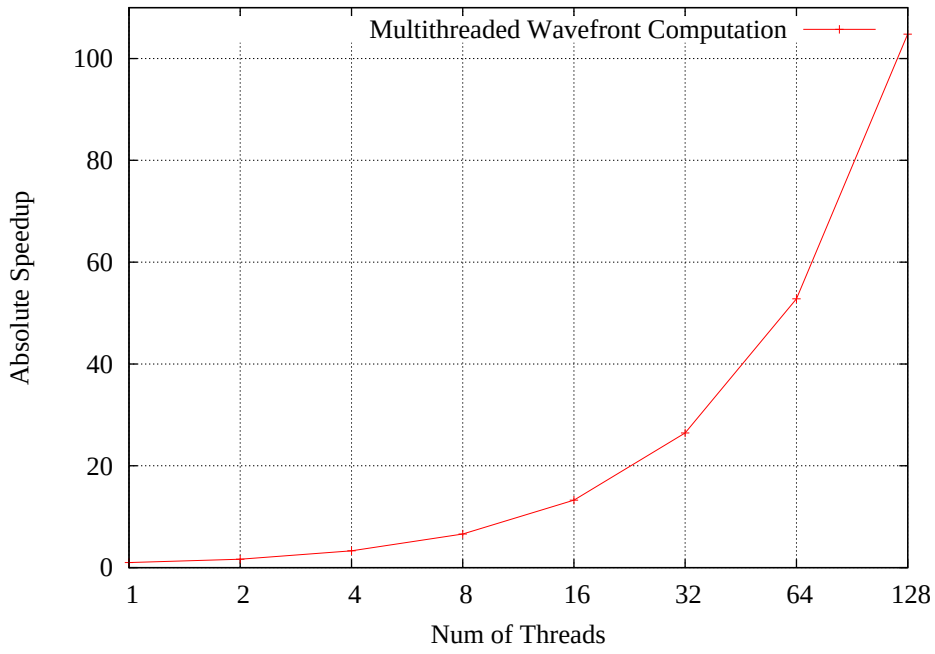


Figure 17: Speedup of Parallelized Wavefront Computation

Table 3: Synchronization Success Rates and SSB Full Rates

Benchmark	64 threads		128 threads	
	Success Rate (%)	SSB Full Rate (%)	Success Rate (%)	SSB Full Rate (%)
Table Toy	99.98%	0	99.96%	0
Livermore Loop 13	99.03%	0	99.24%	0
Livermore Loop 14	99.58%	0	99.14%	0
Hash Table	99.96%	0.0011%	99.92%	0.0013%
Livermore Loop 6 (chunk size = 1)	95.97%	0	90.96%	0
Livermore Loop 6 (chunk size = 2)	96.95%	0	92.66%	0
Wavefront	99.71%	0	99.63%	0

4.3 Synchronization Success Rates

For the six benchmarks used in the experiments, we also report the synchronization success rates, i.e, the percentage of successful synchronizations. As shown in Table 3, even when running with a large number of threads, most of the fine-grain synchronization operations are successful, which shows the righteous of our philosophy to ensure the low overhead of successful fine-grain synchronizations. Also shown in the table, for all experiments, only 0.001% of synchronization operations used by the Hash Table benchmark encounter the situation that the SSB happens to be full. In all other benchmarks, this situation never happens. This evidence shows that a small SSB for each memory bank is normally sufficient to cache the access states of outstanding synchronizing data units for multitheading programs. Using modest hardware cost, SSB achieves the same effect as if each word of the entire memory is tagged.

5 Related Work

The Cyclops-64 [17] is evolved from a preliminary design of Cyclops architecture [10]. However, there are significant differences between the two. The original Cyclops chip integrates 128 *32-bit* processing cores (thread units), each four of which share a floating point unit. In the current C64 design, there are 160 *64-bit* thread units and 80 floating point units, each of which is shared by two thread units. For the memory hierarchy, in the original Cyclops design, all thread units share 16 on-chip 512KB DRAM banks, and each four of the thread units share a 16-KB data cache. The current C64 design employs scratchpad memory instead of data cache, and 160 on-chip SRAM banks that are shared between all thread units.

The difference between SSB and tagged memory (e.g. full/empty bits) in other machines [2, 3, 6, 14, 18, 24, 25, 39] has been explained in the section of introduction. The M-Machine [24] does not

[2] For large matrix, which has to be stored in the off-chip DRAM, we partition it into smaller matrices, each of which can fit into on-chip SRAM. Using the same technique in [21], the computation is performed on a small matrix that is already loaded into SRAM. At the same time, certain number of helper threads prefetch the next small matrix to be computed. The same synchronization mechanism is used for the computation of the small matrix.

only tag every memory location with a single synchronization bit, but also allows fast synchronization between three on-chip processors through register-register communication. However, there is no study showing that the shared register approach can scale to large number of processing cores on a chip. Proposals of hardware support of locking, such as hardware queue based QOLB [23], lock box [40] for SMT processor, SoC lock cache [4], and others, target to improve the efficiency of locking synchronization primitives. However, unlike SSB or tagged memory, none of them support word-level fine-grain synchronization in memory.

Recently hardware transactional memory (TM) [7,19,20,29,35,38], a non-blocking synchronization mechanism, has been proposed as a replacement for the lock-based synchronization. A transaction is a sequence of memory reads and writes executed by a single thread, which is guaranteed to be atomic and serializable. TM systems provide great potential to facilitate multithreading programming, however, those proposals require far from modest hardware modifications. Most TM systems need to extend and modify the existing cache coherence protocols and speculative execution techniques, which are not employed in C64-like large-scale multi-core chip architectures.

6 Summary

Using IBM Cyclops-64 as a case study, this paper shows how fine-grain synchronization can be effectively and efficiently supported with the proposed *synchronization state buffer* (SSB) on the emerging large-scale multi-core chip architectures. The proposed solution makes use of only modest hardware extension to support word-level fine-grain synchronization in memory. 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 paper is the first work that explores hardware support of word-level fine-grain synchronization for large-scale multi-core architectures, such as C64. The future work includes: 1) investigate language extension to map high-level constructs to the SSB synchronization mechanism; 2) study compiler techniques that can optimize the allocation and scheduling of the SSB resources, especially for important scientific and engineering applications; and 3) explore potential extensions of SSB mechanisms to facilitate parallel program debugging, runtime performance monitoring, and other techniques that may take advantage of states bookkeeping by hardware.

Acknowledgment

We would like to acknowledge the support from IBM, in particular, Monty Denneau, who is the architect of the IBM Cyclops-64 architecture, ETI, the Department of Defense, the Department of Energy (DE-FC02-01ER25503), the National Science Foundation (CNS-0509332), and other government sponsors. We would also like to acknowledge other members of the CAPSL group at University of Delaware, who provide a stimulus environment for scientific discussions and collaborations, in particular Ioannis Venetis, Juan del Cuvillo, Yuan Zhang, and Geoff Gerfin. We would also like to thank Vugranam Sreedhar for the useful discussions.

References

[1] HPC chanllenge benchmark.

[2] 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.

[3] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: a processor architecture for multiprocessing. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 104–114, 1990.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] C. Cascaval, J.G. Castanos, L. Ceze, M. Denneau, M. Gupta, J.E. Moreira D. Lieber, K. Strauss, and Jr. H.S. Warren. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, Boston, Massachusetts.

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

[12] Ding-Kai Chen. *Compiler Optimizations for Parallel Loops with Fine-Grained Synchronization*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[13] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The impact of synchronization and granularity on parallel systems. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 239–248, 1990.

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

[15] 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 ISCA2005*, Madison, Wisconsin, June 2005.

[16] 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 IPDPS2005*, page 265, Denver, Colorado, USA, April 2005.

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

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

[19] 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. Jun 2004.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] Collin McCurdy and Charles Fischer. User-controllable coherence for high performance shared memory multiprocessors. pages 73–82, San Diego, CA, 2003.

[29] 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*, pages 53–65, Washington, DC, USA, 2006.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, February 2006.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 54–58, Orlando, Florida, January 9–13, 1999.

[41] 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.

[42] 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.