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

Synchronization State Buffer: Supporting Efficient Fine-Grain Synchronization on Many-Core Architectures

Weirong Zhu
Vugranam C. Sreedhar†
Ziang Hu
Guang R. Gao

CAPSL Technical Memo 67 Revised
November 20th, 2006

†IBM T.J.Waston Research Center, Hawthorne, NY 10532. Email: vugranam@us.ibm.com

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Efficient fine-grain synchronization is extremely important to effectively harness the computational power of large-scale multi-core (or many-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. This paper proposes the *Synchronization State Buffer* (SSB), a scalable architectural design for fine-grain synchronization that efficiently performs synchronizations 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 the SSB design in the context of the 160-core IBM Cyclops-64 architecture. Using detailed simulation, we present our experience for a set of benchmarks with different workload characteristics. We demonstrate the effectiveness and efficiency of the SSB solution: (i) for *mutual exclusion*, our solution uses fine-grain locking at each of the memory units to efficiently avoid unnecessary serialization of the operations on different elements of the same concurrent data structure; (ii) 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 using fine-grain synchronization operations.

Contents

List of Figures

List of Tables

1 Introduction

The design of high-performance processor chips is rapidly moving towards many-core architectures that integrate 10s (or beyond) of tightly-coupled processing cores on a single chip [12, 18]. Intel recently announced its research prototype many-core design with 80 cores on a single die [55]. IBM Cyclops-64 will support 160 hardware thread units in one chip [23]. In order to fully utilize the on-chip parallelism provided by such many-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 many-core processors is often limited by the lack of effective architectural support for efficient fine-grain synchronization. Software-only solutions (with very limited architectural support) can 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 compilation time. Consider the example shown in Figure 1, which shows the kernel doall loop in the Random Access HPCC benchmark [1] implemented using OpenMP API. The critical section ensures the read-modify-write operations in the loop to be performed atomically.[1] Unstructured references like the one shown in Figure 1 are impossible to analyze at compilation time. Therefore, the compiler can only assign a single lock for the whole table `y[]`, which introduces unnecessary serialization. An efficient run-time fine-grain synchronization mechanism is necessary to exploit such inherent fine-grain parallelism.

```
#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 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 2: Livermore Loop 6: Linear Recurrence Equations

Now consider the Livermore Loop 6 shown in Figure 2, which represents widely used linear recurrence equations [25]. As shown in Figure 3, 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[1]`, `W[2]`, ... , `W[i-1]`. Such cross-iteration dependencies of array `W` makes it difficult to parallelize this loop at compilation time [54]. Again, a fine-grain synchronization mechanism is essential to enforce the data dependencies among concurrent threads.

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

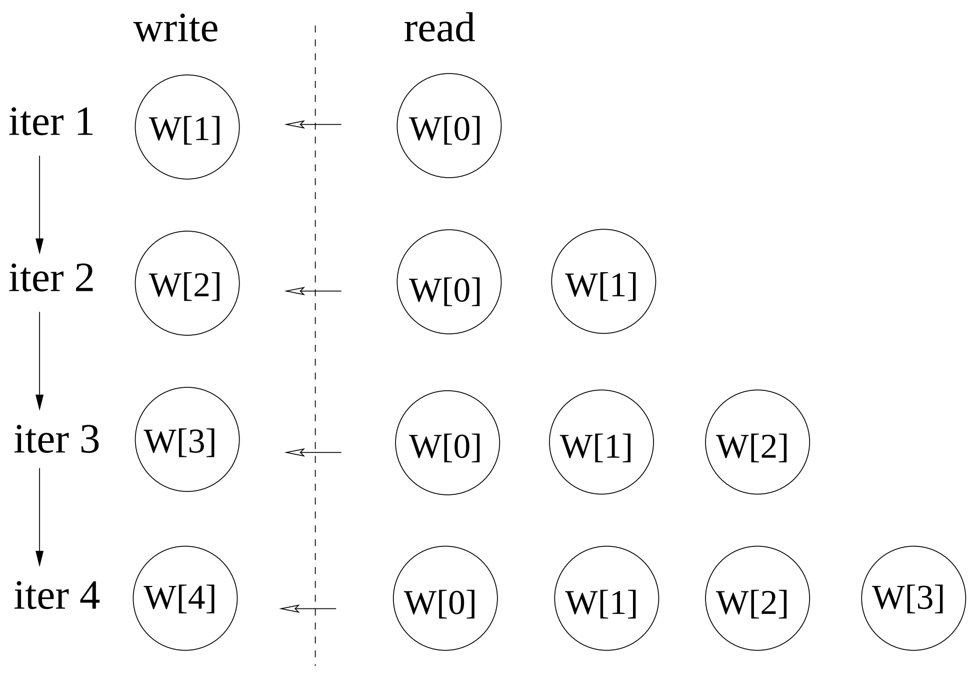


Figure 3: Characteristics of Livermore Loop 6

There are several design choices that one can employ to implement fine-grain synchronization. For instance, HEP [51], Tera [5], MDP [19], Sparcle [3], M-Machine [34], the MT processor in Eldorado [26], 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 [8] and M-structure [11] 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 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, 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 2. [3]

[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 5.4, we will discuss the details.

Based on this 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. SSB can supports a rich set of synchronization functionalities. In our current design, SSB 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.

To understand the design space of SSB, we implemented our solution in the context of the 160-core IBM Cyclops-64 (C64) chip architecture as a case study. We extended the C64 architecture simulator with the new SSB architectural features to explore the design space using detailed simulation.

For mutual exclusion, SSB supports different fine-grained locks, including word-level read, write, and recursive locks. Our approach exploits the ample parallelism that often exists in operations on different elements of concurrent data structures. Using SSB-based fine-grain locking on each memory unit, we avoid the unnecessary serialization of those operations. For the set of benchmarks that we tested, we have observed up to 84% performance improvement using SSB when compared to software only solutions.

For read-after-write data dependence synchronization, SSB allows fine-grain 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. Our SSB design can efficiently exploit the do-across style loop-level parallelism, where one can directly implement loop-carried data dependences using SSB fine-grain synchronization and eliminate the use of unnecessary barriers in the loop. Our experimental results demonstrate significant performance gain using such fine-grain data synchronization. For instance, using SSB, we observe a 312% performance improvement over the coarse-grain based approach when solving linear recurrence equations.

Finally, our experiments also demonstrate that 1) a small SSB for each memory bank is normally sufficient to record the access states of outstanding synchronizing data units for multithreading programs, and 2) most of fine-grain synchronization operations are successful.

2 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 .

2.1 Many-Core Architecture

Architects are actively exploring the design space of many-core chip, which is currently in a state of flux. The design choices for efficient implementation of a fine-grain synchronization solution are likely to be strongly influenced by the underlying architectural design and model. In this paper, we focus on a class of many-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 [55] and the Larrabee mini-cores chip [2], and the IBM Cyclops-64 (C64) chip architecture [23]. In this paper, we have implemented SSB in the context of the C64 architecture.

One important feature of such many-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 word in on-chip memory for fine-grain synchronization incurs high cost. One of our design objectives in SSB is to avoid such cost.

A few multi-core chip designs (such as the IBM Cell processor [27, 28], the Cyclops-64 [23] chip, and the ClearSpeed CSX architecture [16]) employ *explicitly programmable local memory store* for each processing core rather than coherent data cache. 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 [28]. 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 [24]. 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 many-core architectures with the local store approach.

Another important feature of such many-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 the associated programming and execution environment are centered around TiNy Threads [22]. 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 many-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 [15, 31, 53, 56]. An assumption for designing and implementing SSB that we make throughout the paper is the non-preemptive thread execution model.

2.2 Formalization of the Key Observation

Recall the key observation that at any instance 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:

$$S(t) \leq \alpha(t) \times T, \tag{1}$$

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

$$S(t) \leq \alpha(t) \times T \ll M \times B, \tag{2}$$

For the examples shown in Figure 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

$$S_b = S(t)/B \ll M, \tag{3}$$

We will use Equations 1, 2, and 3 in the design of SSB in the next section.

3 Design of Synchronization State Buffer

SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of actively synchronized data units to support and accelerate word-level fine-grain synchronization. In this section we will discuss the various design parameters of SSB.

3.1 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:

$$E_b \ll M_b \tag{4}$$

From Equations 1, 2, and 3 we know that if an application can take advantage of the architectural design objective of Equation 4, then the following is the design requirement for the size of the buffer:

$$E_b \geq S_b \tag{5}$$

Let us generalize the above relation as follows:

$$E_b \simeq \beta \times S_b \tag{6}$$

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.

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 many-core chips.

3.2 Structure of SSB

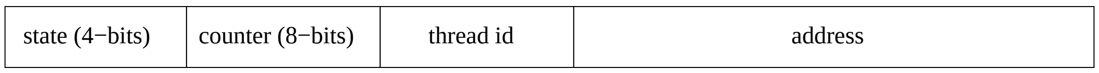


Figure 4: One SSB Entry

The overall structure of an SSB entry is shown in 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 many-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 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.

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. Since SSB maintains the states for synchronized memory locations in hardware, we avoid explicit software-managed synchronization variables that cost extra memory. Also, with one memory transaction, an SSB instruction not only

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

perform the synchronization on the memory location, but also brings the datum to the processor on success.

4 An Architectural Model for SSB

4.1 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 it does 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 synchronization state to the software, and the software can 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.

SSB provides the 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 operations swlock_l and srlock_l acquire a *write* or *read lock* for the memory location MemAddr respectively. Upon success, they also load the content of the memory location to Value. The operation sunlock releases the lock previously acquired. Figure 5 illustrates how the lock/unlock operations interact with the SSB hardware.

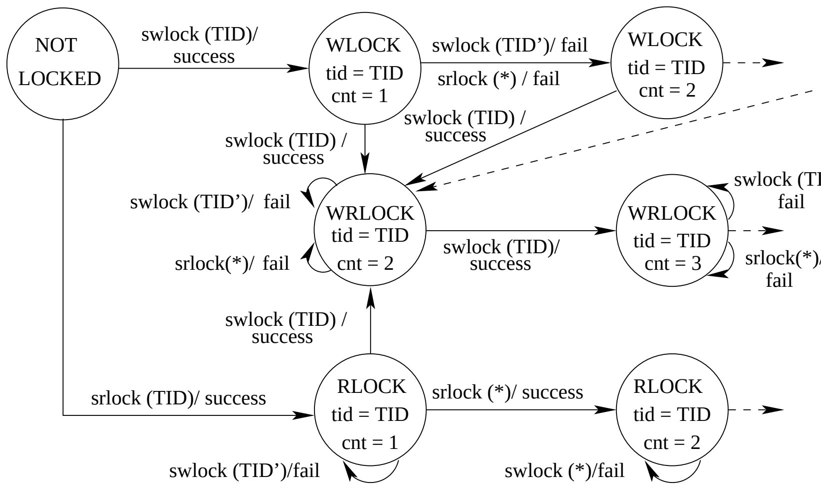
As shown in Figure 5(a), swlock_l acquires the *write lock* for memory location MemAddr. If there is no record for this location in SSB, which means it is not locked by any other thread, an entry for this location is allocated, and the state is set to WLOCK. Before this location is unlocked by the owner, write/read lock acquisition from other threads will fail, and cause the "counter (cnt)" to be 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.

The operation srlock_l acquires a *read lock* for the 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(b). When a lock is finally released, the corresponding entry in SSB will be 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.

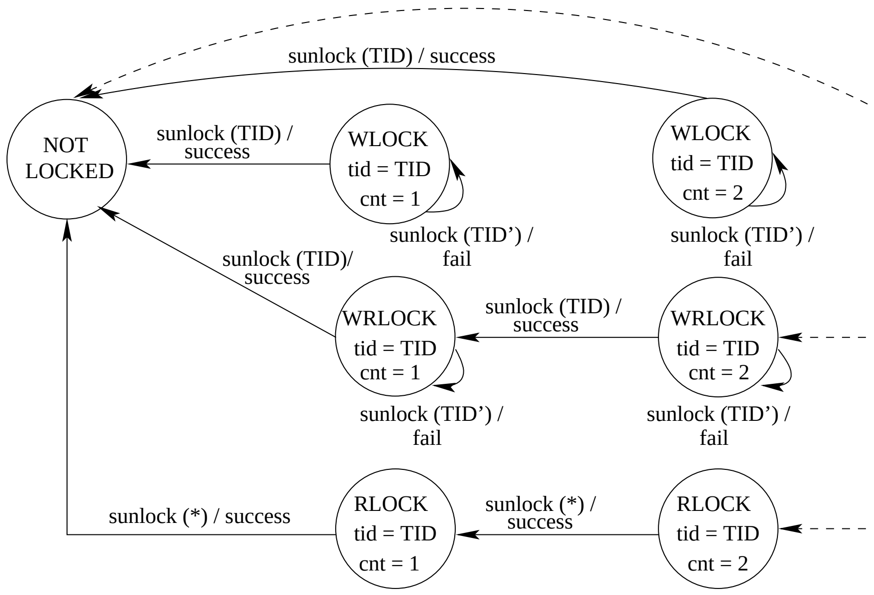
4.2 Fine-Grain Data Synchronization

SSB can help the programmer to exploit data-level parallelism by allowing a program to perform synchronized reads and writes 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.



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

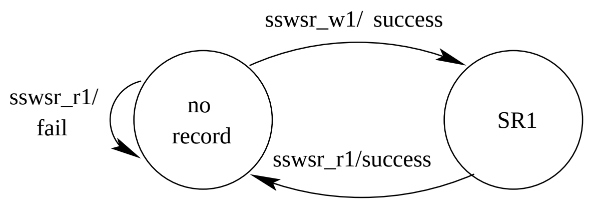
4.2.1 Single-Writer-Single-Reader (SWSR) Data Synchronization

SSB can help the programmer to exploit data-level parallelism by allowing a program to perform synchronized reads and writes at the word-level in memory. The single-writer-single-reader (SWSR) synchronization enforces order between a thread that produces the data and another thread that consumes the data. The following are the interfaces:

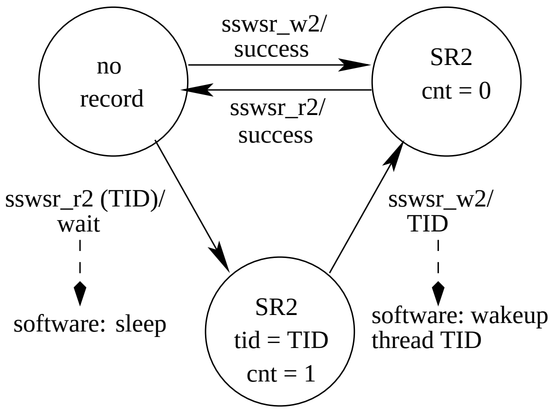
```
RT = sswrsr_w1(MemAddr, Value);
/* sswrsr_w1: SWSR synchronized write mode 1    */
/* MemAddr: the address of the memory location  */
/* Value: the Value to be written to MemAddr    */
/* RT: return value, success or failure         */

(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 6(a), 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 sswsr_r1 is successfully executed, the entry in SSB is released, and the content of MemAddr is loaded for the reader.

A *blocking* strategy can be implemented with sswsr_w2 and sswsr_r2. As illustrated by Figure 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 voluntarily suspends its execution and goes to sleep. Later, sswsr_w2 instruction issued by the writer will write Value into MemAddr, and set the counter to 0 to indicate the availability of the data. The instruction



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

also returns the thread id (TID) of the reader to the writer. The write can wake up the sleeping reader. The reader can now retrieve the value by sswsr_r2 and free the corresponding entry in the SSB.

4.2.2 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:

```
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             */
```

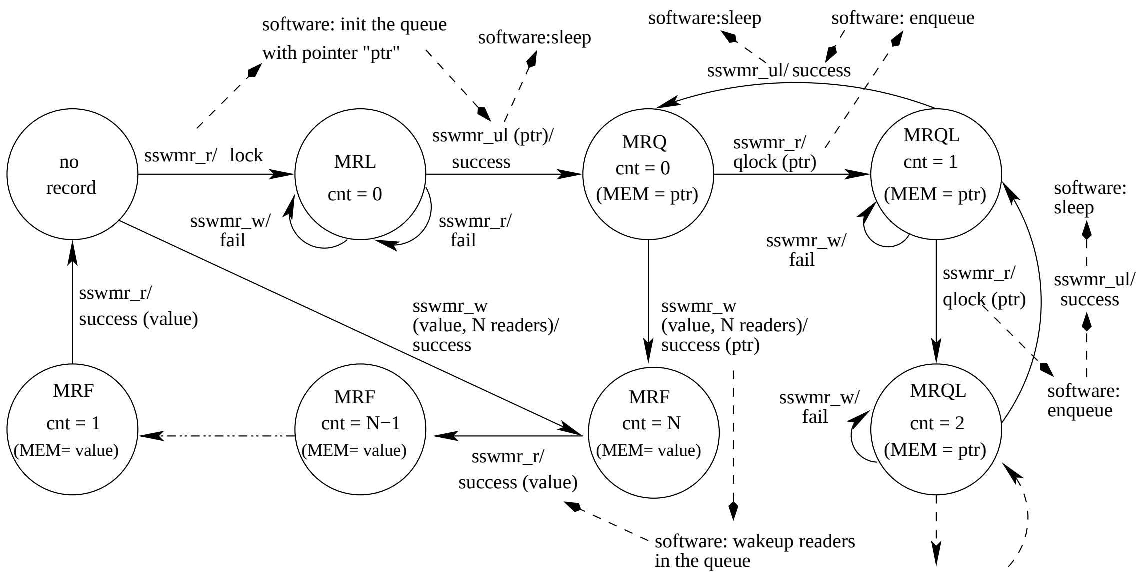
```
(RT, Value) = sswmr_r(MemAddr);
/* sswmr_r: SWMR synchronized read                             */
/* MemAddr: the address of the memory location                 */
/* RT: return value, success, failure, lock mode, or qlock mode */
/* Value: the content of the memory location upon success, or   */
/*        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 7 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 7, 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.

4.3 Hardware Resource Constraints

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 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



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

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. 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 8.

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

```
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 8: Operations of Memory Controller

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 9.

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 5.5, for many benchmarks, only one has a small percentage of synchronization operations that encounter the “full” situation.

5 Evaluation

Our objective in this section 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

```
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 9: Operations of the Software Handler

characteristics of SSB in the context of the IBM 160-core Cyclops-64 (C64) chip architecture [23], which represents a class of many-core architectures that we discussed in Section 2.

5.1 C64 Architecture and Experimental Framework

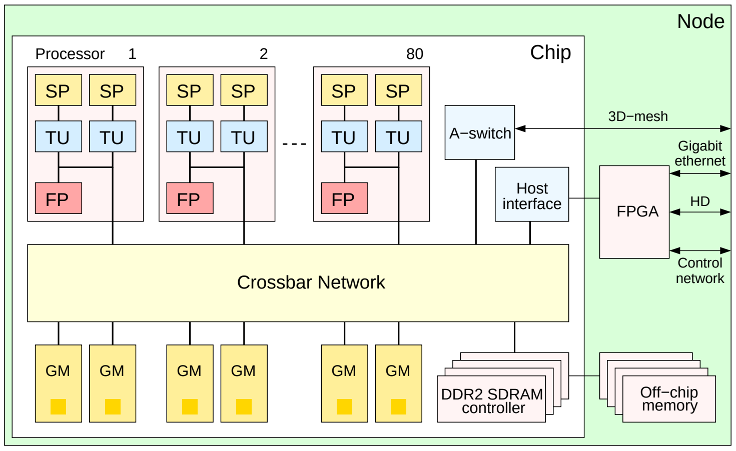


Figure 10: Cyclops-64 Chip Architecture

C64 is evolved from a preliminary design of BlueGene/Cyclops architecture [13]. As shown in Figure 10, the C64 chip contains 160 thread units (TU) (running at 500MHz) and 160 embedded SRAM memory banks (32KB each) in a single silicon die. There are 80 floating point units, each of which is shared by two TUs. A 32KB instruction cache, is shared among 10 TUs. C64 has efficient support for thread level execution, such as ISA-level sleep/wakeup instructions. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; while asleep it can be awaken by another thread through a hardware interrupt. C64 features an explicitly addressable three-level (Scratchpad memory, on-chip SRAM, off-chip DRAM) memory hierarchy without data cache. A portion of each SRAM bank can be configured as the *scratchpad memory (SP)*, which can be accessed by a corresponding TU with very low and deterministic latency. The remaining sections of all on-chip SRAM banks together form the *global memory (GM)* that is uniformly addressable by all TUs. There are 4 memory controllers connected to *off-chip DRAM* banks (up to 2GB). 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 and the SSBs attached to all memory banks. Memory accesses to GM and DRAM go through an on-chip crossbar network, which sustains a 384 GB/s on-chip bandwidth. The crossbar also guarantees a sequential consistency memory model for the C64 chip architecture. Fence-like instructions is not needed to ensure the order between memory operations [23, 60]. C64 provides no hardware support for context switch, and uses a non-preemptive thread execution model. The peak performance of a C64 chip is 80GFLOPS.

The current C64 architecture supports several synchronization mechanisms. Atomic in-memory instructions, such as *fetch-and-add*, can be used to implement various *spin-locks*. The sleep/wakeup instructions can be used to perform post/wait type of synchronization. A 16-bit signal bus, to which all thread units are connected, provides a means to efficiently implement barriers. The *compare-and-swap* (CAS), *linked-load*, and *store-conditional* instructions are not currently supported in the design of C64 chip architecture [23]. However, for the purpose of comparison, we include the CAS instruction in the ISA when simulating the C64 chip architecture.

We implemented the proposed SSB as an extension to the C64 ISA using an execution-driven binary-compatible full-system simulator for the C64 many-core architecture [21]. 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.

In the rest of the section we will compare SSB with the above synchronization mechanism, and answer the following questions: 1) What is the cost of a successful synchronization operation? 2) How effective is SSB for fine-grain mutual exclusion synchronization? 3) How effective is SSB for fine-grain data synchronization? and 4) How effective is SSB in exploiting fine-grain parallelism? The set of benchmarks that we used for experiments are summarized in Table 2.

Table 2: Summary of Benchmarks Analyzed for SSB Behavior

Benchmark	Source	Description
Random Access	HPCC Benchmarks [1]	random updates of memory
Livermore Loop 13	Livermore Loops [25]	2-D particle-in-cell
Livermore Loop 14	Livermore Loops	1-D particle-in-cell
Loop $G1$	SSCA#2 [10]	graph problem
Ordered Integer Set	Common data structure	hash-table based
$K1$, $K2$, $K3$ $K4$, $K5$, $K6$	Kernel Loops from SPEC OMP [52]	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.

5.2 Cost of Successful Synchronization

Previous studies have shown that fine-grain synchronization results in successful synchronization in most cases [36, 57], and this is also true for SSB-based fine-grain synchronization (see Section 5.5). Therefore, it is important to ensure that the cost of a successful synchronization is very low.

5.2.1 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 imple-

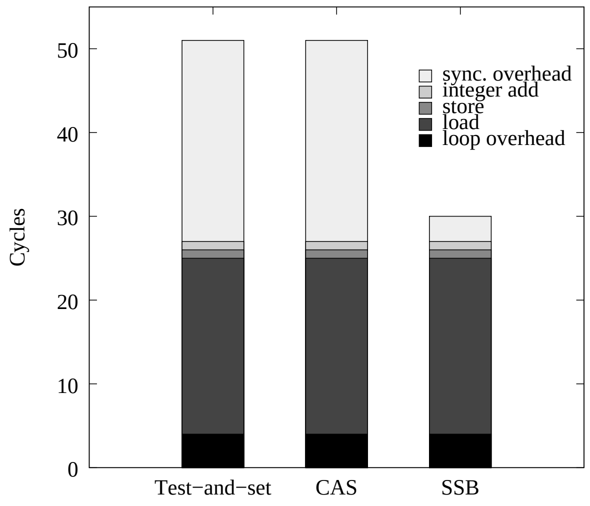


Figure 11: Overheads of Synchronization Mechanisms

mented 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 11 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.

5.2.2 Fine-grain data synchronization

Table 3: 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

In this experiment we use a simple loop of 10,000 iteration with 2 threads. Each iteration contains a barrier operation. We get the reference time by employing one thread to perform a store before the barrier, and the other to perform a load after the barrier. The overhead is computed by comparing the 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 3, the overhead of SSB data synchronization operations are small when performed successfully. The major overhead comes from the difference between a synchronized write and a 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 SSB operations.

5.3 Effectiveness of SSB for Fine-Grain Lock

In this subsection, 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 [1], two of the Livermore loops, and a hash-table based implementation of ordered integer set.

5.3.1 Random Access

As shown in Figure 1, the address of the memory location to be mutually exclusively accessed is only known right before entering the critical section. To ensure correctness, 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.

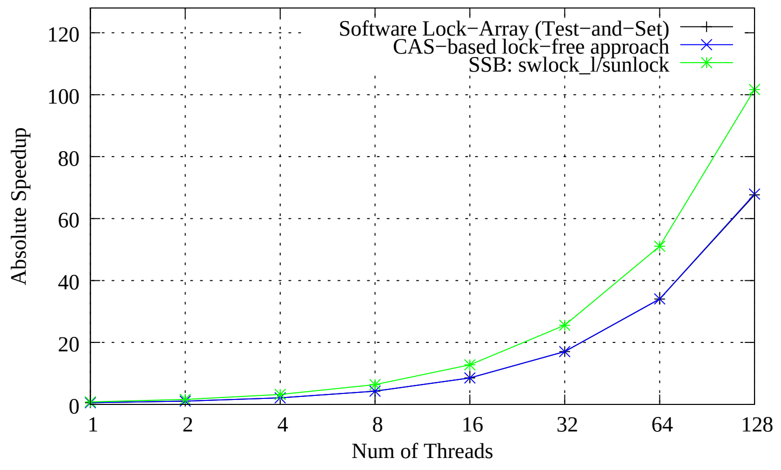


Figure 12: Absolute Speedup of Random Access Benchmark

Figure 12 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 two curves overlapped in Figure 12). 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.

5.3.2 Livermore Loop 13 and 14

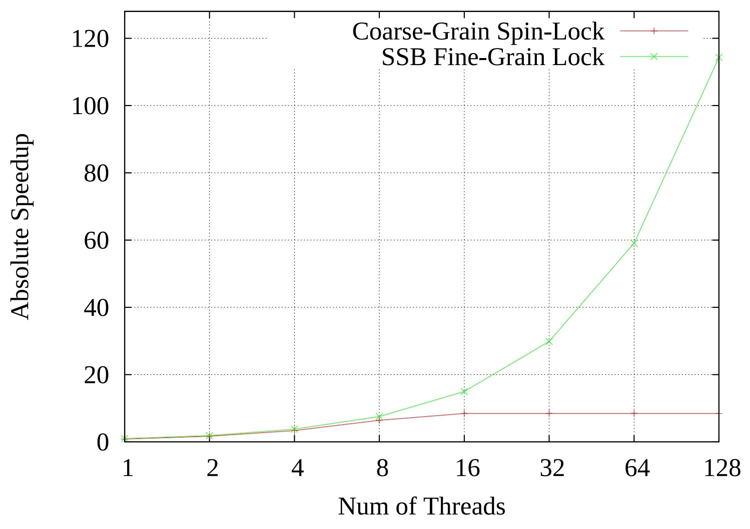


Figure 13: Absolute Speedup of Livermore Loop 13

Because of the cross-iteration dependencies (which cannot be determined statically), Livermore Loops 13 and 14 cannot be easily parallelized [54]. Within each iteration, a few elements of the array are updated. However, the calculation of the indices is 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.

Figure 13 and 14 compares coarse-grain synchronization with SSB. The coarse-grain approach serializes the updates to the array using an MCS [41] 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 13 and 14). As a result, we achieve speedups of 114.3 and 72.4 on 128 threads for Loop 13 and Loop 14, respectively.

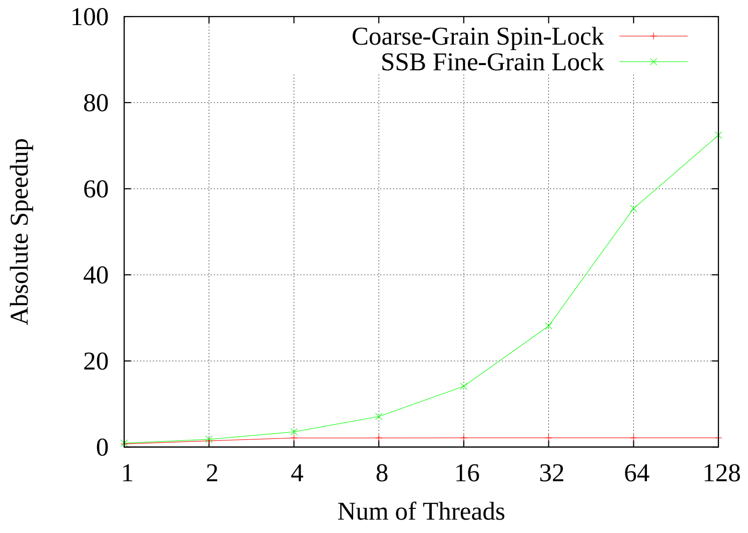


Figure 14: Absolute Speedup of Livermore Loop 14

```
lock_init_arr(&vLock, n, TH);

node_Barrier();

pardo(u, 0, n, 1) {
  for (j=G->outVertexIndex[u]; j<G->outVertexIndex[u+1]; j++) {
    v = G->outVertexList[j];
    if (!isEdgePresent_OutVertex(G, v, u)) {
      my_lock(&vLock[v]);
      inDegree[v]++;
      my_unlock(&vLock[v]);
      impliedEdgeFlag[j] = 1;
      inVertexListSize++;
    }
  }
}

node_Barrier();

lock_destroy_arr(&vLock, n, TH);
```

Figure 15: A Loop ($G1$) Extracted from SSCA#2

5.3.3 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 [10]. A hallmark of the graph problem is the irregular memory accesses, which leads to poor data locality and statically unsolvable synchronization points.

Figure 15 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 [9]. 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.

- **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 15 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 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 16 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 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 16, 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.

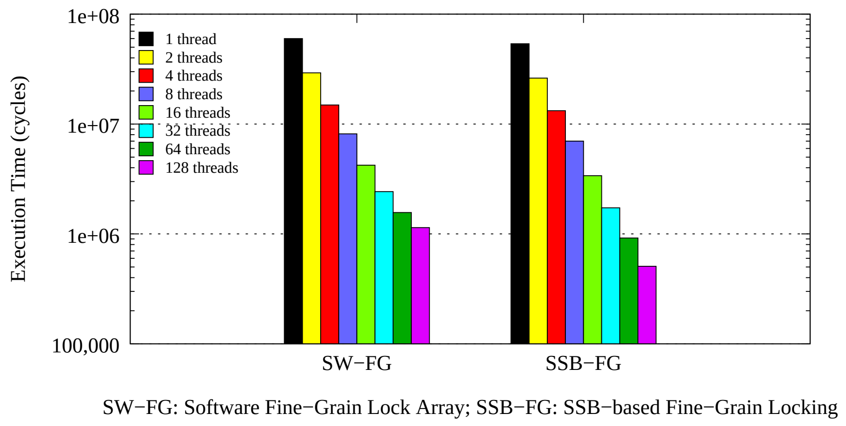


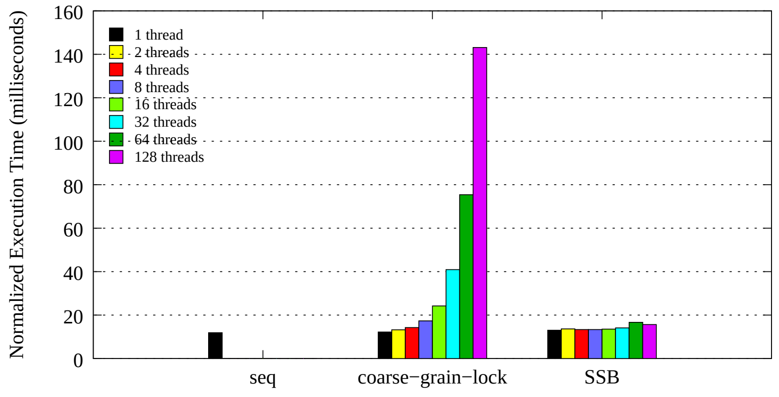
Figure 16: Execution Time of the Loop $G1$ Extracted from SSCA#2

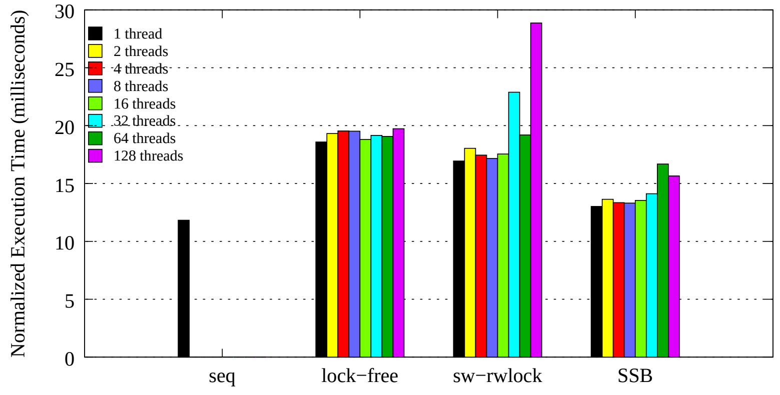
5.3.4 Hash Table Based Ordered Integer Sets

Hash table is a widely used data structure in many applications. 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 MCS spin-lock [20, 41], 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 [42]. The *hazard pointers* mechanism is used to guarantee safe memory reclamation of lock-free objects as well as ABA-safety [43].
- *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.





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

Figure 17: Implement hash table based integer set with different synchronization mechanisms.

Figure 17 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. 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, 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 17, 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.

5.4 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 [35] 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:

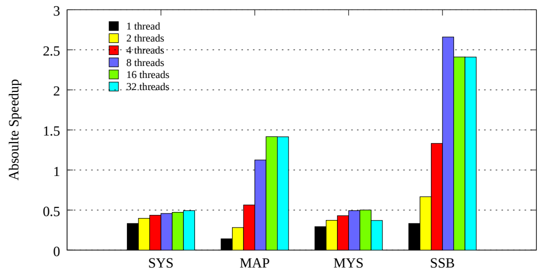
- On multithreaded large-scale multi-core architectures, fine-grain data synchronization mechanism is important and effective for exploiting fine-grain parallelism in scientific application kernels.
- For large-scale multi-core architecture, fine-grain synchronization based parallelization schemes can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

5.4.1 Kernel Loops from SPEC OMP

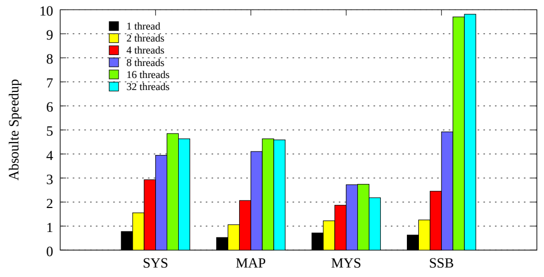
The 6 kernel loops, $K1$, $K2$, ..., $K6$, are extracted from multithreaded applications, such as *314.mgrid* and *318.galgel*. [4] The cross-iteration dependence distance of all the kernels are constant and positive.

[4]These 6 kernel loops are the same ones used in the performance evaluation section of [35].

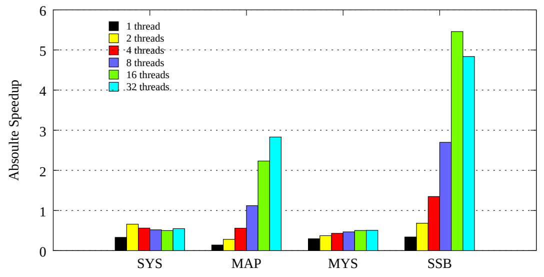
We parallelize those loops by statically assigning iterations to different threads in a round robin fashion. We compared the SSB-based approach with the three software-based synchronization methods (SYS, MAP, and MYS), which are recently proposed by Kejariwal et. al. [35]. For more details, please refer to [35]. For the SSB-based approach, we use SSB SWSR operations to enforce the data dependencies among threads.



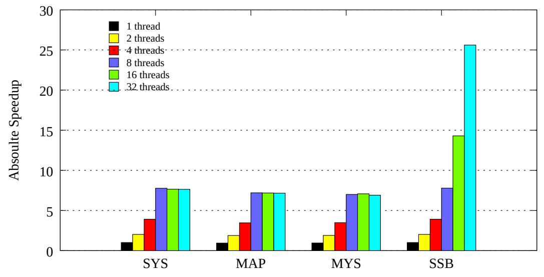
(a) $K1$ (DIST = 8)



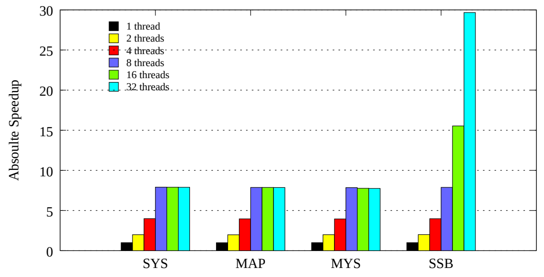
(b) $K2$ (DIST = 16)



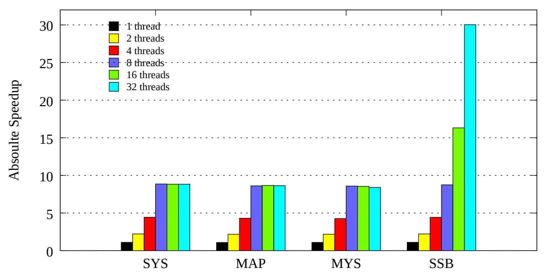
(c) $K3$ (DIST = 8)



(d) $K4$ (DIST = 8)



(e) $K5$ (DIST = 8)



(f) $K6$ (DIST = 8)

Figure 18: Performance of Multithreaded DOACROSS Kernel Loops. (DIST: dependence distance.)

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 18(a), (b), (c)), it is not surprising that SSB-based hardware approach shows 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 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 18(d), (e), (f)). These 6 loops illustrate the effectiveness of SSB-based fine-grain data synchronization (compared to state-of-the-art software approaches) for DOACROSS loops

```
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 19: Sequential version of 1D Laplace Solver

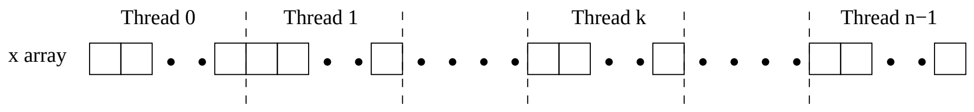


Figure 20: 1D Laplace Solver: Partition the Array among n Threads

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.

5.4.2 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 19). 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.

The multithreaded parallel implementation partitions the 1D array among threads, as shown in Figure 20. 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 21). 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,

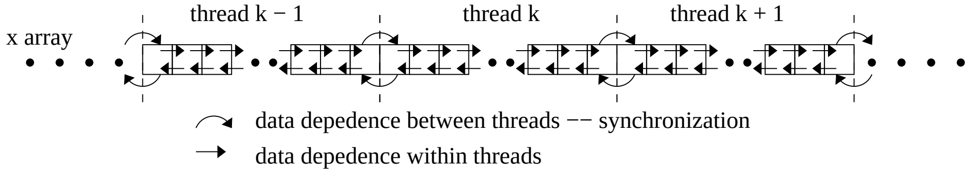

thread k − 1
thread k
thread k + 1
x array
data depedence between threads −− synchronization
data depedence within threads

Figure 21: Data Dependencies and Synchronizations in 1D Laplace Solver

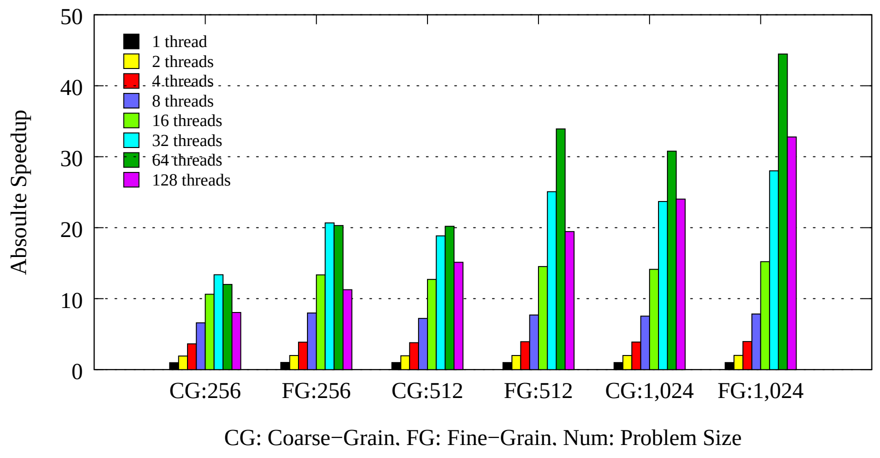


Figure 22: Barrier-based Coarse-Grain Synchronization vs. SSB-based Fine-Grain Synchronization for 1D Laplace Solver. (Problem Size: 512, and 1,024)

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 new iteration.

Figure 22 and 23 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%.

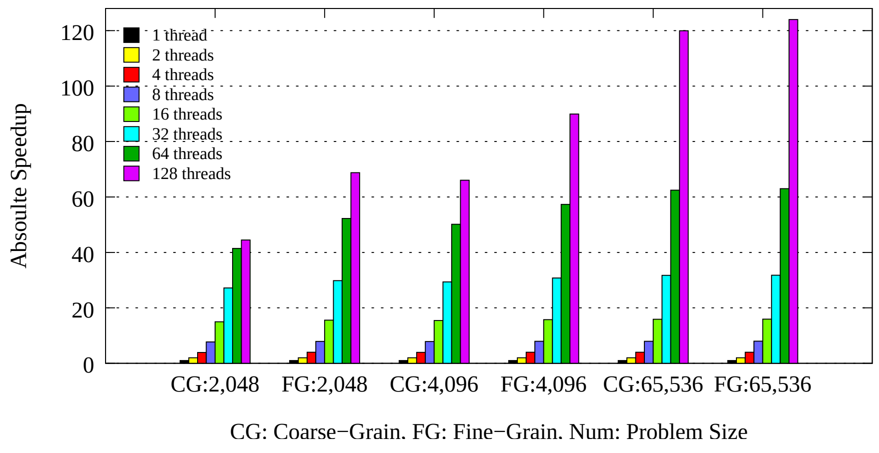


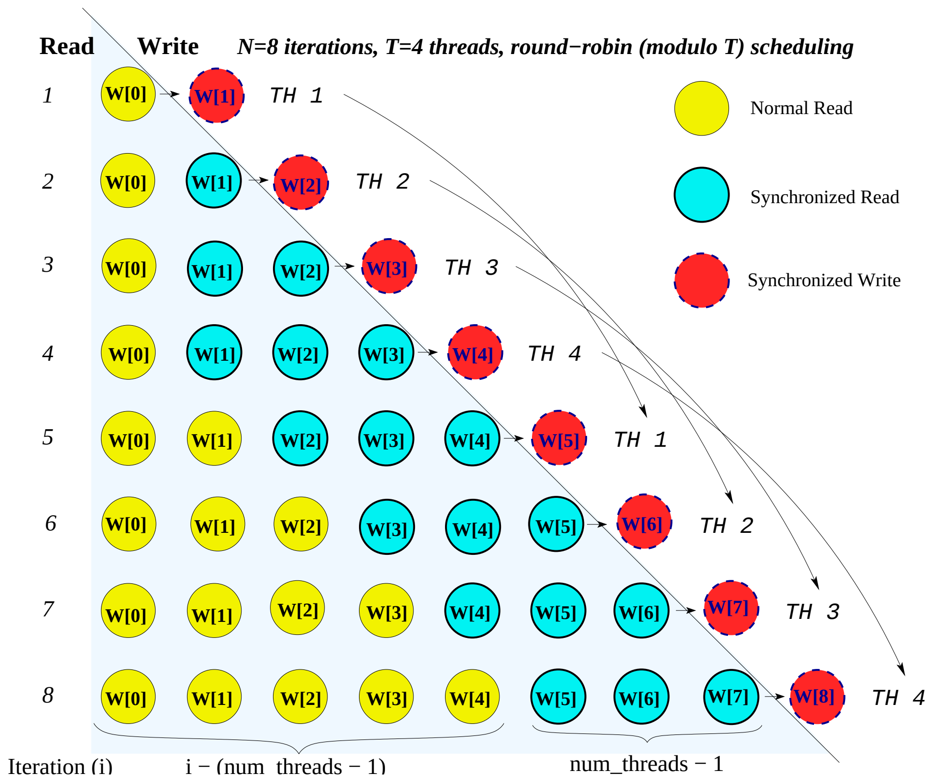
Figure 23: Barrier-based Coarse-Grain Synchronization vs. SSB-based Fine-Grain Synchronization for 1D Laplace Solver. (Problem Size: 2,048, and 4,096)

5.4.3 Linear Recurrence Equations (Livermore Loop 6)

We parallelize the loop shown in Figure 2 by assigning the iterations to different threads in a round-robin fashion. The SSB single-reader-multiple-writer data synchronization mechanism is used to enforce the read-after-write dependencies among iterations.

Our parallelization and synchronization scheme is shown in Figure 24, 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 2 ($S(t) \ll M \times B$) introduced in Section 2.

Figure 25 compares the fine-grain data synchronization based approach with a coarse-grain based implementation as introduced in [25]. For the fine-grain approach, the *chunk_size*, as explained above, 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 25, when *chunk_size* equals to 2 or 4, the speedups are calculated


Read
Write
N=8 iterations, T=4 threads, round−robin (modulo T) scheduling
Normal Read
Synchronized Read
Synchronized Write
1 W[0] W[1] TH 1
2 W[0] W[1] W[2] TH 2
3 W[0] W[1] W[2] W[3] TH 3
4 W[0] W[1] W[2] W[3] W[4] TH 4
5 W[0] W[1] W[2] W[3] W[4] W[5] TH 1
6 W[0] W[1] W[2] W[3] W[4] W[5] W[6] TH 2
7 W[0] W[1] W[2] W[3] W[4] W[5] W[6] W[7] TH 3
8 W[0] W[1] W[2] W[3] W[4] W[5] W[6] W[7] W[8] TH 4
Iteration (i)
i − (num_threads − 1)
num_threads − 1

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

Figure 24: Parallelization and Synchronization of Livermore Loop 6

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 25, by exploiting fine-grain parallelism, the fine-grain data synchronization based approach always performs better when running on a large number of threads. Figure 26 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 26, we can observe that the performance improvement increases significantly when the number of threads is large. 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 global communication caused by barriers.

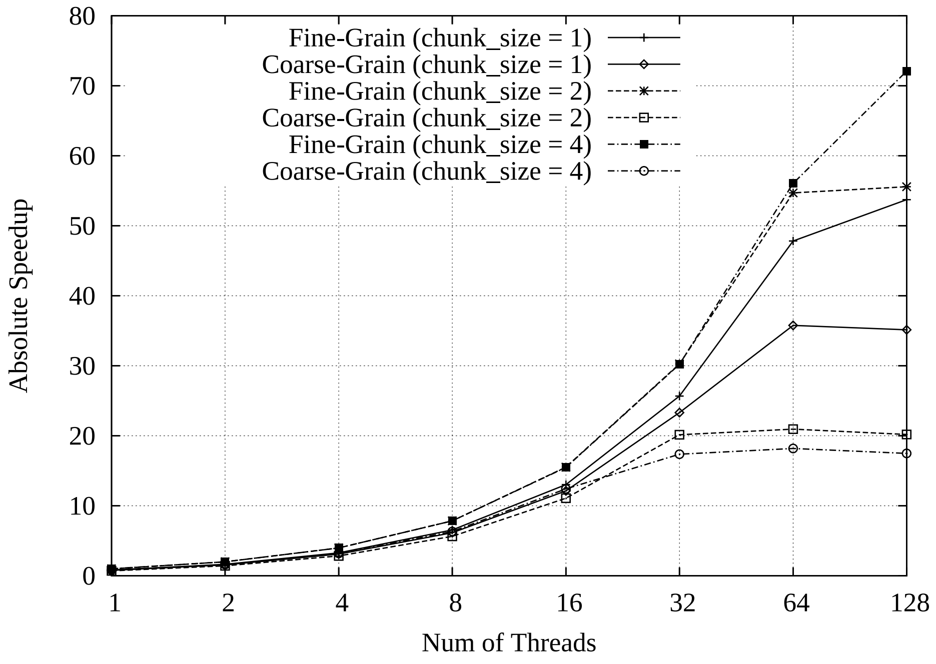


Figure 25: Speedup of Parallelized Livermore Loop 6

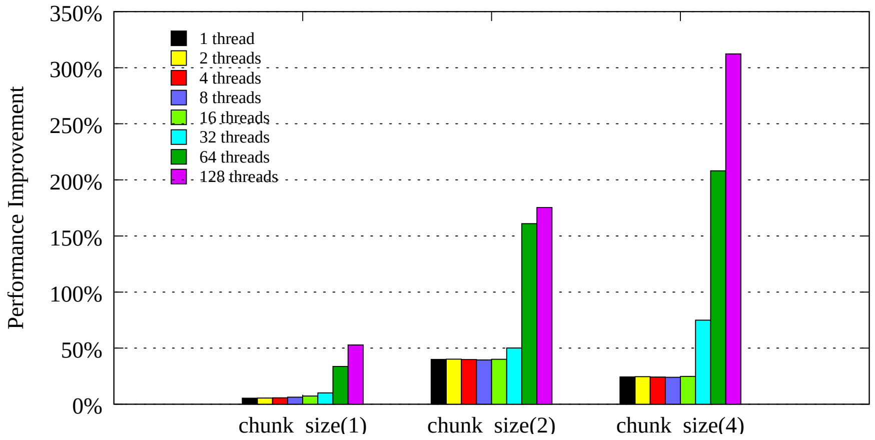


Figure 26: Livermore Loop 6: Fine-Grain vs Coarse-Grain

5.4.4 Wavefront Computation

Wavefront computations are common in scientific applications. Given a matrix (see Figure 27), 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.

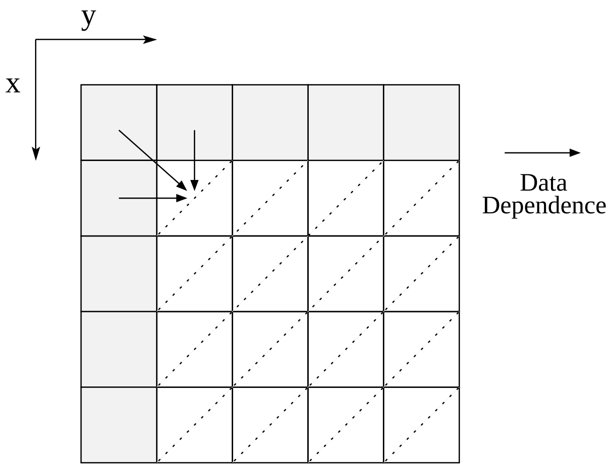
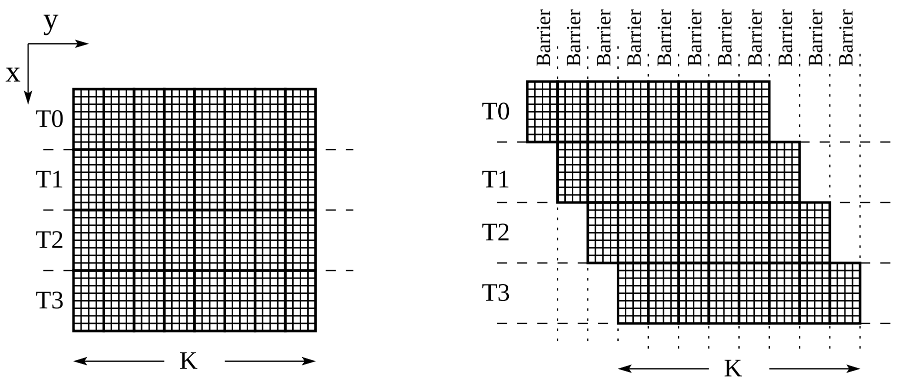


Figure 27: Wavefront Computation



(a) Partition the Solution Space

(b) Computation Scheme in Parallel

Figure 28: Coarse-Grain Parallelization of Wavefront Computation: Number of Threads $T = 4$, Number of Computation Blocks $K = 2 * T = 8$.

Figure 28 illustrate the coarse-grain parallelization scheme inspired by [58]. 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 28(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 29). With this static scheduling policy, to compute an element, only the availability of its above neighbor needs to be checked. 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.

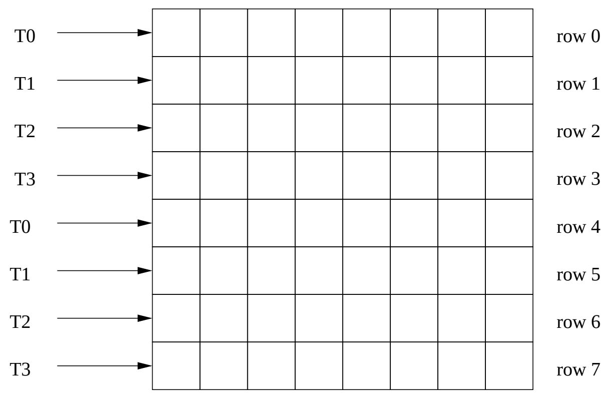


Figure 29: Fine-Grain Parallelization of Wavefront Computation: Number of Threads $T = 4$, Solution Space 8×8.

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.

Our experiments are conducted with a 1024×1024 matrix [5]. Figure 30 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 30 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.

[5]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.

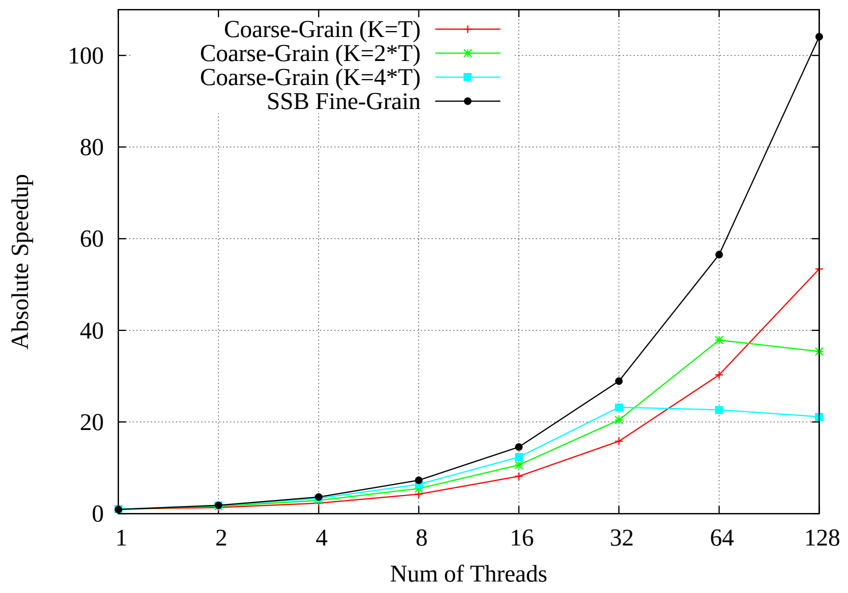


Figure 30: Absolute Speedup of Parallelized Wavefront Computation. T denotes the number of threads, and K denotes the number of computation blocks

From Figure 30, 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 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.

5.5 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 4. 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 5.2).

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 4. For those loops, when the number of threads are smaller than or equal to the dependence distance (shown as DIST in Figure 18), the synchronization successful rates are also very high. Otherwise, the rates are not high, since certain portion of synchronized reads

Table 4: 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

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.

6 Related Work

Past research has indicated that fine-grain parallelism unleashed by some dataflow models that use the I-structure like fine-grain synchronization [8] far exceeded the capacity of a given hardware architecture to effectively exploit the parallelism [17]. Therefore, researchers have worked on solutions that somehow "throttle" the parallelism during program execution. In this paper, given a large number of processing cores and limited per core on-chip memory supported by underlying many-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.

Our SSB design provides an illusion that the entire memory is tagged at word-level, and therefore can be considered as a "virtual tagged memory" design. The major difference between SSB and the classical tagged memory (e.g. full/empty bits) in HEP [51], Tera [5], MDP [19], Sparcle [3], M-Machine [34], the MT processor in Eldorado [26], and other machines, has been explained in Section 1. I-structure [8] memory system employed in some dataflow model based architectures [8, 33] 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.

The M-Machine [34] is another architecture that tags every memory location and allows fast synchronization between three on-chip processors through register-register communication. Hardware mechanisms such as QOLB [32], MAOs on SGI Origin [38], lock box [54] for SMT processor, SoC lock cache [4], AMO [59] 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 [34] also allows fast synchronization between three on-chip processors through register-register communication. Sampson et. al. [50] proposed barrier filters, a hardware mechanism for enabling fast barrier synchronization on multi-core chips.

Recently, hardware transactional memory (TM) [6,29,30,40,45,48,49], 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. Most TM designs need to extend and modify the existing cache coherence protocols and speculative execution techniques. TM systems provide great potential to facilitate multithreading programming. 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.

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 [7, 14, 37, 39, 44, 46, 47]. Those techniques can be combined with SSB-based hardware support to efficiently parallelize do-across loops, especially useful when the synchronization resource requirements are more 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.

The Cyclops-64 [23] is evolved from a preliminary design of Cyclops architecture [13]. 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.

7 Summary

This paper shows how fine-grain synchronization can be effectively and efficiently supported in many-core architecture design using the *synchronization state buffer* (SSB) with only a modest hardware extension. We experimented the SSB design in the context of IBM Cyclops-64 architecture. Using detailed simulation, our experimental results demonstrate the effectiveness and efficiency of our solution for a set of benchmarks with different workload characteristics.

Our current design assumes the 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 mechanisms will be necessary for implementing SSB design. The virtualization of SSB is beyond the scope of the current paper, and we regard this as important future work. Other possible future research includes language extensions to map high-level constructs to the SSB synchronization mechanism, compiler techniques that can optimize the allocation and scheduling of SSB resources, and exploration of 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

This work was supported in part by IBM, ETI, DoD, DoE (DE-FC02-01ER25503), NSF (CNS-0509332), and other government sponsors. We acknowledge Monty Denneau, the architect of the IBM Cyclops-64 architecture. We thank Arun Kejariwal and Xinmin Tian for providing the code of kernel loops extracted from SPEC OMP. We thank all the members of CAPSL group at University of Delaware. Special thanks to Ioannis E. Venetis, Guangming Tan, Juan del Cuvillo, Yuan Zhang, Xiaoming Li, and anonymous reviewers for their invaluable feedback on the paper.

References

[1] HPC chanllenge benchmark. [Online]. Available: http://icl.cs.utk.edu/hpcc/

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

[3] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeoung, G. D’Souza, and M. Parkin, “Sparcle: An evolutionary processor design for large-scale multiprocessors,” *IEEE Micro*, vol. 13, no. 3, pp. 48–61, June 1993.

[4] B. Akgul and V. Mooney, “The system-on-a-chip lock cache,” *International Journal of Design Automation for Embedded Systems*, vol. 7, no. 1-2, pp. 139–174, Sept. 2002.

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The Tera computer system,” *SIGARCH Comput. Archit. News*, vol. 18, no. 3b, pp. 1–6, 1990.

[6] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, “Unbounded transactional memory,” in *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, Feb 2005, pp. 316–327.

[7] J. H. Anderson and M. Moir, “Universal constructions for large objects,” in *International Workshop on Distributed Algorithms*, 1995.

[8] Arvind, R. S. Nikhil, and K. K. Pingali, “I-structures: data structures for parallel computing,” *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 598–632, 1989.

[9] 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*, vol. 58, no. 1, pp. 92–108, 1999.

[10] 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)*, Goa, India, Dec. 2005, pp. 465–476.

[11] 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*, 1991, pp. 538–568.

[12] S. Y. Borkar, H. Mulder, P. Dubey, S. S. Pawlowski, K. C. Kahn, J. R. Rattner, and D. J. Kuck, “Platform 2015: Intel processor and platform evolution for the next decade,” 2005.

[13] C. Cascaval, J. Castanos, L. Ceze, M. Denneau, and et. al., “Evaluation of a multithreaded architecture for cellular computing,” in *the 8th International Symposium on High Performance Computer Architecture (HPCA02)*, Boston, MA, 2002.

[14] D.-K. Chen, “Compiler optimizations for parallel loops with fine-grained synchronization,” Ph.D. dissertation, UIUC, 1994.

[15] L. Chen, Z. Hu, J. Lin, and G. 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.

[16] ClearSpeed Technology, “CSX processor architecture whitepaper,” 2006.

[17] D. E. Culler, K. E. Schauser, and T. 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*, 1993, pp. 153–164.

[18] B. Dally, “Computer architecture in the many-core era,” in *Key note in the 24th Intl. Conf. on Comput. Design (ICCD 2006)*, 2006.

[19] W. J. Dally and et. al., “The message-driven processor,” *IEEE Micro.*, vol. 12, no. 2, pp. 23–39, 1992.

[20] J. del Cuvillo, W. Zhu, and G. 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.

[21] J. del Cuvillo, W. Zhu, Z. Hu, and G. 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.

[22] J. del Cuvillo, W. Zhu, Z. Hu, and G. 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.

[23] M. Denneau and H. S. Warren, Jr., “64-bit Cyclops principles of operation,” IBM, Yorktown Heights, NY, Apr. 2005.

[24] A. E. Eichenberger, K. O’Brien, and et. al., “Optimizing compiler for the Cell processor,” in *in Proc. of 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2005, pp. 161–172.

[25] J. Feo, “An analysis of the computational and parallel complexity of the livermore loops,” *Parallel Computing*, vol. 7, no. 2, pp. 163–185, 1988.

[26] J. Feo and et. al., “Eldorado,” in *Proceedings of the 2nd conference on Computing frontiers*, 2005, pp. 28–34.

[27] M. Gschwind, “Chip multiprocessing and the Cell broadband engine,” in *in Proc. of the 3rd Conf. on Computing frontiers*, 2006.

[28] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, “Synergistic processing in Cell’s multicore architecture,” *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[29] L. Hammond, V. Wong, M. Chen, and et. al., "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, Jun 2004, p. 102.

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

[31] Z. Hu, J. del Cuvillo, W. Zhu, and G. 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.

[32] A. Kägi and D. B. J. R. Goodman, "Efficient synchronization: Let them eat QOLB," in *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, 1997, pp. 170–180.

[33] K. M. Kavi, A. R. Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of cache memories for multi-threaded dataflow architecture," in *Procs. of the 22nd Intl. Symp. on Computer architecture*, 1995.

[34] S. W. Keckler, W. J. Dally, D. Maskit, N. P. Carter, A. Chang, and W. S. Lee, "Exploiting fine-grain thread level parallelism on the MIT multi-ALU processor," in *the 25th annual international symposium on Computer architecture*, 1998, pp. 306–317.

[35] A. Kejariwal, H. Saito, X. Tian, M. Gikar, W. Li, U. Banerjee, A. Nicolau, and C. D. Polychronopoulos, "Lightweight lock-free synchronization methods for multithreading," in *the 20th International Conference on Supercomputing*, Cairns, Australia, June 2006.

[36] D. Kranz and et. al., "Low-cost support for fine-grain synchronization in multiprocessors, Tech. Rep. MIT/LCS/TM-470, 1992.

[37] 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.

[38] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," in *Procs. of the 24th Intl. Symp. on Computer Architecture*, 1997.

[39] Z. Li and W. Abu-Sufah, "A technique for reducing synchronization overhead in large scale multiprocessors," in *Proceedings of the 12th Annual International Symposium on Computer Architectures*, May 1985, pp. 284–291.

[40] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," in *the 33rd International Symposium on Computer Architecture*, Washington, DC, 2006, pp. 53–65.

[41] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization onshared-memory multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[42] M. M. Michael, “High performance dynamic lock-free hash tables and list-based sets,” in *the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, August 2002, pp. 73–82.

[43] M. M. Michael, “Hazard pointers: Safe memory reclamation for lock-free objects,” *IEEE Trans. Parallel Distrib. Syst*, vol. 15, no. 6, pp. 491–504, 2004.

[44] S. P. Midkiff and D. Padua, “Compiler algorithms for synchronization,” *IEEE Trans. on Comput.*, vol. 36, no. 12, pp. 1485–1495, 1987.

[45] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, “LogTM: Log-based transactional memory,” in *Proceedings of the 12th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2006.

[46] M. F. P. O’Boyle, L. Kervella, and F. Bodin, “Synchronization minimization in a SPMD execution model,” *J. Parallel Distrib. Comput.*, vol. 29, no. 2, pp. 196–210, 1995.

[47] R. Rajamony and A. L. Cox, “Optimally synchronizing DOACROSS loops on shared memory multiprocessors,” in *Proceedings of 1997 International Conference on Parallel Architectures and Compiliation Techniques*, 1997.

[48] R. Rajwar and J. 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*, Oct 2002, pp. 5–17.

[49] R. Rajwar, M. Herlihy, and K. Lai, “Virtualizing transactional memory,” in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2005, pp. 494–505.

[50] J. Sampson, R. Gonzalez, J.-F. Collard, N. Jouppi, M. Schlansker, and B. Calder, “Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers,” in *Procs. of the Intl. Symp. on Microarchitecture*, 2006.

[51] B. Smith, “The architecture of HEP,” in *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, ser. Scientific Computation Series, J. S. Kowalik, Ed. Cambridge, MA: MIT Press, 1985, pp. 41–55.

[52] Standard Performance Evaluation Corporation, “SPEC OpenMP benchmark suite.” [Online]. Available: http://www.spec.org/omp/

[53] G. Tan, N. Sun, and G. R. Gao, “A parallel dynamic programming algorithm on a multi-core architecture,” in *Procs. of 19th ACM Symp. on Parallelism in Algorithms and Architectures*, Jun. 2007.

[54] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy, “Supporting fine-grained synchronization on a simultaneous multithreading processor,” in *the 5th Intl. Symp. on High-Performance Computer Architecture*, Orlando, Florida, Jan. 9–13, 1999, pp. 54–58.

[55] S. Vangal, J. Howard, G. Ruhl, and et. al., “An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS,” in *Procs. of 2007 Intl. Solid-State Circuits Conf.*, Feb. 2007.

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

[57] D. Yeung and A. 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*, 1993, pp. 187–192.

[58] D. Yeung and A. 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*. New York, NY, USA: ACM Press, 1993, pp. 187–192.

[59] L. Zhang, Z. Fang, and J. B. Carter, “Highly efficient synchronization based on active memory operations,” in *Procs. of 18th Intl. Parallel and Distrib. Processing Symp.*, 2004.

[60] Y. Zhang and et. al., “Sequential consistency revisit: the sufficient condition and method to reason the consistency model of a multiprocessor-on-a-chip architecture,” in *Intl. Conf. of Parallel and Distrib. Computing and Networks*, Innsbruck, Austria, Feb 2005.