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

Weirong Zhu
Department of ECE
University of Delaware
Newark, DE 19711
weirong@capsl.udel.edu

Vugranam C. Sreedhar
T.J. Watson Research Center
IBM Research
Hawthorne, NY 10532
vugranam@us.ibm.com

Ziang Hu, Guang R. Gao
Department of ECE
University of Delaware
Newark, DE 19711
{hu,ggao}@capsl.udel.edu

ABSTRACT

Efficient fine-grain synchronization is extremely important to effectively harness the computational power of 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 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.

Categories and Subject Descriptors: C.0 [Computer Systems Organization]: General

General Terms: Design

Keywords: Many-Core, Fine-Grain Synchronization, SSB

1. INTRODUCTION

High-performance processor design is rapidly moving towards many-core architectures that integrate 10s (or beyond) of processing cores on a single chip [12, 7]. Intel recently announced its research prototype many-core design with 80 cores on a single die [42]. IBM Cyclops-64 will support 160 hardware thread units in one chip [16]. The granularity of parallelism that can be efficiently exploited in such many-core chips is often limited because of the lack of effective architectural support for efficient fine-grain synchronization. Software-only solutions, with very limited architectural support, often lead to high synchronization overhead, high storage cost, and poor scalability. It is often difficult or even impossible to harness fine-grain parallelism at compilation time. Consider the example in Figure 1(a), which simulates the kernel DOALL loop in the Random Access HPCC benchmark [1] implemented using OpenMP. The critical section ensures the read-modify-write operations in the loop to be performed atomically. Unstructured references like the one shown in Figure 1(a) 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; }
}
```

(a) 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];
```

(b) Livermore Loop 6

Figure 1: Examples

Now consider the Livermore Loop 6 shown in Figure 1(b), which represents widely used linear recurrence equations [17]. At each iteration `i` of the outer loop, the computation of `W[i]` depends on elements of `W` (`W[1]`, `W[2]`, ... , `W[i-1]`) that are calculated in all previous iterations. It is difficult to parallelize the loop at compilation time because of the cross-iteration dependencies [41]. Again, a fine-grain synchronization mechanism is essential to enforce the data dependences among concurrent threads.

There are several design choices that one can employ to implement fine-grain synchronization. For instance, HEP [38], Tera [5], MDP [13], Sparcle [3], M-Machine [24], the MT processor in Eldorado [18], 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 by associating additional access state bits with each word in memory. 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 high cost associated with tagging every word in the entire memory.

ISCA'07, June 9–13, 2007, San Diego, California, USA.

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 are actively participating in synchronization.* To further elaborate this observation, consider the kernel loop shown in Figure 1(a). Now let T be the number of threads and assume $T \ll N$, where N is the size of the table `y[]`. In the example, we can 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 $S \ll N$. 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 one can focus on recording and managing synchronization states of only those actively synchronized memory words. One could make a similar observation for the Livermore Loop 6 shown in Figure 1(b) (see Section 5.4).

Based on this observation, we introduce a novel synchronization architecture, with a modest hardware extension to many-core architectures, called the *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 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 support a rich set of synchronization functionalities. 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 SSB

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 [42] and the Larrabee mini-cores chip [2], and the IBM Cyclops-64 (C64) chip architecture [16]. 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 [19], the Cyclops-64 [16] chip, and the ClearSpeed CSX architecture [11]) employ *explicitly programmable local memory store* for each processing core rather than coherent data cache. Such 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 [19]. From the software perspective, the local store with low and deterministic access latency can be effectively exploited by compilers to further improve the resulting code [19]. Unlike many synchronization mechanisms based on the existence of coherent cache, another design objective of SSB is to make no such assumption, and thus it can also be implemented for many-core designs with local stores.

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 [15]. 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 [21, 40, 10, 43]. 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 during the parallel execution only a small set of memory locations are actively participating in synchronization. We formalize this 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:

$$S(t) \leq \alpha(t) \times T, \quad (1)$$

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

$$S(t) \leq \alpha(t) \times T \ll M \times B, \quad (2)$$

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

$$S_b = S(t)/B \ll M, \quad (3)$$

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 such cost, and therefore we want:

$$E_b \ll M_b \quad (4)$$

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:

$$E_b \geq S_b \quad (5)$$

Let us generalize the above relation as follows:

$$E_b \simeq \beta \times S_b \quad (6)$$

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 hurt since the buffer will fill up and we have to fall back to a software mechanism for synchronization. In practice, architects can determine the number of entries, and the level of set associativity of an SSB according to the target applications, the transistor budget, the power consumption requirements, and other design factors.

Each SSB functions as a look-up table. Given the small size of each SSB, the lookup function can be easily implemented with common hardware technology. One merit of SSB is its de-centralized nature. Because of the independence of each SSB, the hardware cost for implementing SSB increases only linearly proportional to the number of memory banks, and the complexity of hardware logic remains the same for each SSB. Therefore, SSB is a scalable fine-grain synchronization solution for many-core chips.

3.2 Structure of SSB

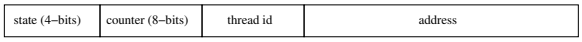


Figure 2: One SSB Entry

The overall structure of an SSB entry is shown in Figure 2. 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-bit counter; and (4) a 4-bit field that can support up-to 16 different synchronization modes. The address bits are used as a *key* to search in the buffer. The remaining three fields form the synchronization state for a 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, which we will explain in the next section. Table 1 shows different synchronization modes that we support in our design. An entry in SSB is allocated and released according to its state and the function of an SSB instruction operating on it.

Table 1: SSB State Bits

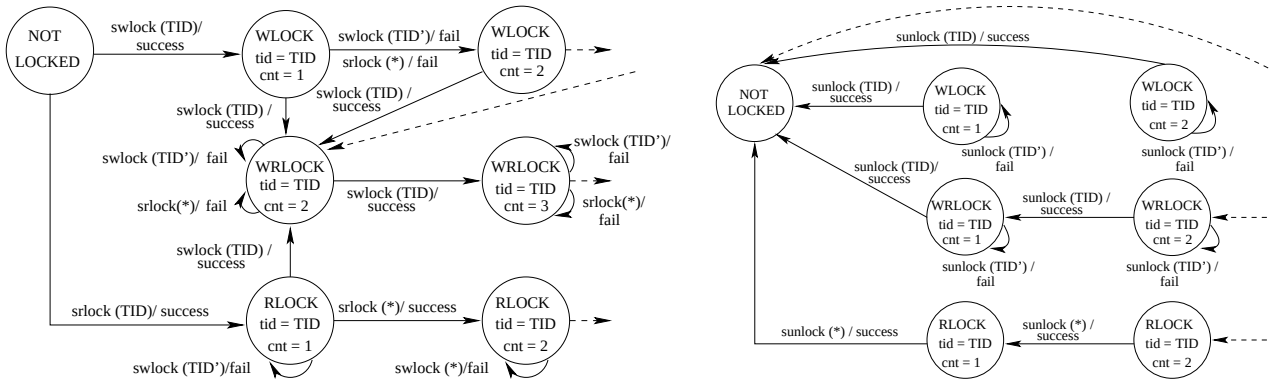
State	Function	Description
0x0000	WLOCK	Write Lock
0x0001	RLOCK	Read Lock
0x0010	WRLOCK	Write-Recursive Lock
0x0011	SR1	Single-Writer-Single-Reader Mode 1
0x0100	SR2	Single-Writer-Single-Reader Mode 2
0x0101	MRF	Single-Writer-Multiple-Reader Full Mode
0x0110	MRL	Single-Writer-Multiple-Reader Lock Mode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

Memory instructions, including SSB instructions, are handled in a FIFO manner when arriving at a memory bank through the on-chip interconnection network. In our design, with one memory transaction, an SSB instruction not only performs the synchronization on the memory location, but also brings the datum to the processor on success.

4. AN ARCHITECTURAL MODEL FOR SSB

4.1 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 the location. In the SSB of the corresponding memory bank, an entry for this address, if does not exist, 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,



(a) states transition caused by swlock_l and srlock_l operations (b) states transition caused by sunlock operation

Figure 3: State transition diagram of SSB lock/unlock operations. **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 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".**

which 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 compilation time. SSB provides the following operations to perform the lock and 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 location           */
```

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. sunlock releases the lock previously acquired.

See Figure 3(a), swlock_l acquires the *write lock* for location MemAddr. If there is no record for this location in SSB, which means it is not locked yet, 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 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 is shown in Figure 3(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

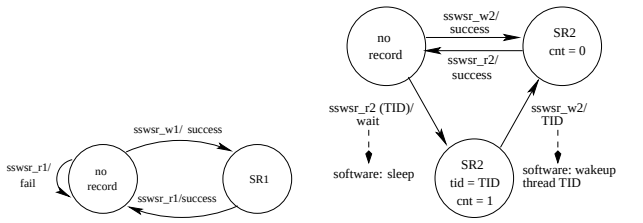
SSB can help the programmer to exploit data-level parallelism by allowing a program to perform synchronized reads and writes at the word-level in memory. The single-writer-single-reader (SWSR) synchronization enforces order between a thread that produces the data and another thread that consumes the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

See Figure 4(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 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. See Figure 4(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 also returns the thread id (TID) of the reader. The writer then can wake up the sleeping reader, which can now retrieve the value by sswsr_r2 and free the corresponding entry in the SSB.



(a) Mode 1: a busy-wait approach (b) Mode 2: a sleep-wakeup approach

Figure 4: State transition diagram of SSB Single-Writer-Single-Reader operations. "software:" means the operation is performed by software.

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. SSB provides three instructions to support this type of data synchronization. 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 5 shows how SSB SWMR operations interact with software to perform the data synchronization between one writer and multiple readers. Ideally, the sswmr_w write operation is executed before all the reads. 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. Each of the following sswmr_r operations reads the value from the memory and decrements the "cnt" by 1. When all the reads finish and the "cnt" reaches 0, the corresponding entry in SSB is freed.

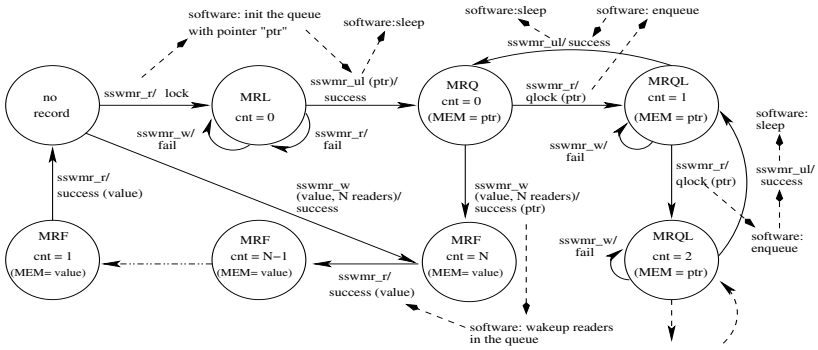
However, it is possible that some readers issue the sswmr_r read operations before the write. The first such sswmr_r instruction allocates an entry in the SSB and sets the state to MRL (lock mode). Then the thread that issues this read will initialize a wait queue, put itself into the queue, and issue a sswmr_ul instruction with the pointer to the tail of the wait queue as a parameter. The sswmr_ul stores the pointer into the memory location, and switches the state to MRQ (queue mode). The following sswmr_r operations issued by other threads will get this pointer, with which a thread can enqueue itself. As shown in Figure 5, 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 a 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 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 not full, HW only solution
No	ON	OFF	3: HSSB full, set SBIT on, trap to SW
No	Any	ON	4: Entries in SSSB, trap to SW

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



The "MEM =" in the parentheses indicates the content of the memory location that is monitored by this SSB entry. 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.

Figure 5: State transition diagram of SSB Single-Writer-Multiple-Reader Operations.

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 entry to HW
ON	No	ON	4: SW only solution

To check the SBIT, FBIT, search the HSSB for matching entry, and flush entry from SSSB to HSSSB, 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, which is case 2, and the handler takes the same action as case 1. Otherwise, it performs the operations on SSSB, then checks the FBIT. If the FBIT is OFF, which is case 3, the handler attempts to flush the entry to the HSSB. If successful, the handler removes the software entry from the SSSB. The remaining steps 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 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, and compare SSB with other synchronization mechanisms. We explore the characteristics of SSB in the context of the IBM 160-core Cyclops-64 (C64) chip architecture [16], which represents a class of many-core architectures that we discussed in Section 2.

5.1 C64 Architecture and Experimental Framework

C64 is evolved from a preliminary design of Cyclops architecture [8]. The C64 chip contains 160 thread units (TU) (running at 500MHz) and 160 embedded SRAM memory banks (32KB each) in a single silicon die, and with a peak performance of 80GFLOPS. 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. C64 features an explicitly addressable three-level memory hierarchy (Scratchpad memory, on-chip SRAM, off-chip DRAM) 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 [16]. C64 provides no hardware support for context switch, and currently uses a non-preemptive thread execution model.

The current C64 architecture supports several synchronization mechanisms. Atomic in-memory instructions, such as *test-and-set*, can be used to implement various *spin-locks*. The sleep/wakeup instructions can be used for post/wait type of synchronization. A 16-bit signal bus, to which all thread units are connected, supports the efficient implementation of barriers. The *compare-and-swap* (CAS), *load-linked*, and *store-conditional* instructions are not currently supported in the design of C64 chip architecture [16]. However, for the purpose of comparison, we implemented the CAS instruction in the simulator.

We use detailed simulation to evaluate SSB. We use an execution-driven full-system simulator for the C64 architecture, which models all the components of the C64 chip design, including the 160 TUs, the memory modules, and the on-chip crossbar network. The three-level memory hierarchy is modeled in detail, including the contention in memory banks and the crossbar network [14]. We implemented the proposed SSB extension to C64 in the simulator. In our simulation, we use a 16-entry SSB for each on-chip memory bank, and a 1,024-entry SSB for each off-chip memory bank. Both are 8-way set associative.

In the rest of the section, we will answer the following questions: 1) What is the cost of a successful synchronization operation? 2) How effective is SSB for fine-grain lock synchronization? 3) How effective is SSB for fine-grain data synchronization? and 4) How effective is SSB in exploiting

Table 2: Summary of Benchmarks Analyzed for SSB Behavior

Benchmark	Source	Description	Data Set	Synchronization
Random Access	HPCC Benchmarks [1]	random updates of memory	2^{17} 64-bit integers	write lock
Livermore Loop 13	Livermore Loops [17]	2-D particle-in-cell	4K doubles for h table, 512 iterations	write lock
Livermore Loop 14	Livermore Loops	1-D particle-in-cell	4K doubles for rh table, 2,048 iterations	write lock
Ordered Integer Set	Common data structure	hash-table based	25 buckets, average load 100	write/read lock
$K1$, $K2$, $K3$ $K4$, $K5$, $K6$	Kernel Loops from SPEC OMP [39]	DOAcross Loops with constant & positive dependence distances	5000 iterations	SWSR data synch.
Livermore Loop 6	Livermore Loops	linear recurrence equations	5K doubles	SWMR data sync.
2D Wavefront	scientific application kernel	2D wavefront computation	$1K \times 1K$ doubles	SWSR data sync.

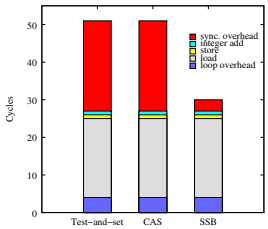


Figure 6: Overheads of Mutual Exclusion

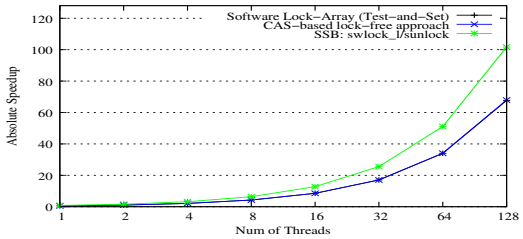


Figure 7: Speedup of Random Access

fine-grain parallelism? The set of benchmarks that we used for experiments are summarized in Table 2.

5.2 Cost of Successful Synchronization

Previous studies have shown that fine-grain synchronization results in successful synchronization in most cases [26, 44]. In this section we demonstrate that the cost of successful synchronization for SSB is very low.

Fine-grain lock: To measure the overhead of different synchronization mechanisms, we use a simple loop that iterates 10,000 times and at each iteration a 64-bit integer value is loaded from on-chip SRAM, a simple arithmetic operation is performed on the value, and the result is stored back to the memory. A reference time is obtained by executing the loop sequentially without using any synchronization. Then the synchronization overhead is calculated by comparing the reference time with the execution time of the same code extended with synchronization operations. When using a test-and-set spin lock, a lock has to be acquired (released) before (after) accessing the memory location. A lock-free approach can be implemented using the *compare-and-swap (CAS)* instruction to commit the result into memory if the content of the memory location has not changed since the last load. SSB-based synchronization is similar to the spin lock in this case. The loop with synchronization is also executed on a single thread, thus all the synchronization operations (lock acquisition or CAS commitment) are successful. Figure 6 shows that SSB incurs the lowest cost among the three mechanisms. One reason for this is that an SSB instruction performs a successful synchronization and brings the datum to the processor in one memory transaction, and that keeps the cost low.

Table 3: Overhead of data synchronization

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

Fine-grain data synchronization: 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 for Fine-Grain Lock

In this section, we examine the effectiveness of SSB for fine-grain locking using the first four benchmarks listed in Table 2.

Random Access. The unstructured reference in the code (see Figure 1(a)) normally results in a single lock being assigned to the whole array, which serializes the execution. One solution is to allocate an array of locks with the same size as y, so that a thread can acquire the corresponding lock for an element of y dynamically. However, this *lock-array* approach doubles the memory usage. Using SSB, 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 with no extra memory usage.

Figure 7 compares three parallelization schemes using different fine-grain synchronization mechanisms. The table is placed in the on-chip SRAM. The software lock-array approach provides scalable performance, however, it incurs large memory usage overhead. The CAS-based lock-free approach achieves a similar speedup curve as the lock-array one (the two curves overlapped in Figure 7). The SSB-based solution indicates the best performance by fully exploiting the fine-grain parallelism with low cost synchronization operations (see Section 5.2). When running on 128 threads, it yields an absolute speedup of 101, outperforming the other two approaches by 50.3% and 49.7% respectively.

Livermore Loops. Because of the cross-iteration dependencies, Livermore Loops 13 and 14 cannot be easily parallelized statically due to the irregular data access [41]. Within each iteration, a few elements of the array are updated.

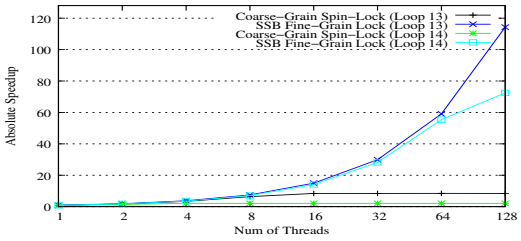
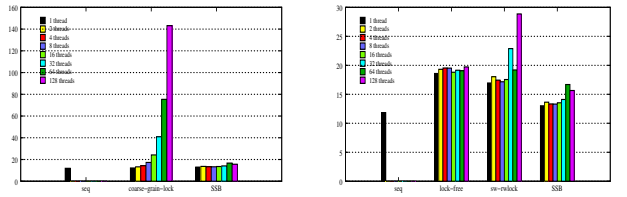


Figure 8: Speedup of Livermore Loops



(a) SSB vs Coarse-Grain (b) SSB vs Software-based Fine-Grain

Figure 9: Hash Table Based Ordered Integer Set. ***Y-axis: normalized execution time by num of threads (milliseconds)***

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.

Figure 8 compares coarse-grain synchronization with SSB. The coarse-grain approach serializes the updates to the array using a MCS spin-lock [29] 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 synchronization exploits the inherent parallelism in the code without unnecessarily serializing the updates to non-conflicting locations. As a result, we achieve speedups of 114.3 and 72.4 on 128 threads for Loop 13 and Loop 14, respectively.

Hash Table Based Ordered Integer Sets. 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. We implemented four different versions of concurrent hash tables:

- *Coarse-grain lock based version:* each bucket is protected by a MCS spin-lock [29].
- *Lock-free version:* uses Michael's lock-free hash table algorithm [30]. The *hazard pointers* mechanism is used to guarantee safe memory reclamation of lock-free objects as well as ABA-safety [31].
- *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 searches through a selected ordered linked list. When the position where the key should 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 and no extra memory usage.

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

Figure 9 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 9, 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 for Fine-Grain Data Synchronization

In this section, we evaluate the performance of SSB-based fine-grain data synchronization with DOACROSS-style kernel loops. We demonstrate how these kernels can be parallelized to exploit fine-grain parallelism with the coordination between SSB hardware and software.

Kernel Loops from SPEC OMP. The 6 kernel loops, $K1$, $K2$, ..., $K6$, are extracted from multithreaded applications, such as *314.mgrid* and *318.galgel* [39]. The cross-iteration dependence distance of all the kernels are constant and positive. We parallelize those loops by statically assigning iterations to different threads in a round robin fashion. We compared the SSB-based approach with the three software-based synchronization methods (SYS, MAP, and MYS), which are recently proposed by Kejariwal et. al [25]. For more details, please refer to [25]. For the SSB-based approach, we use SSB SWSR operations to enforce the data dependences among threads.

The workloads for each iteration of $K1$, $K2$ and $K3$ are small. For instance, there is only one arithmetic operation in the loop body of $K1$. Because of the low computation to synchronization ratio, none of the methods show significant absolute speedup. However, in all cases (Figure 10(a), (b), (c)), it is not surprising that SSB-based approaches show better performance than software methods. For kernel $K4$, $K5$, $K6$, all with a two-level loop nest, the workloads inside each iteration of the outer loop are large. The software methods can only exploit the parallelism of the outer loop. The SSB-based method can naturally 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 10(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

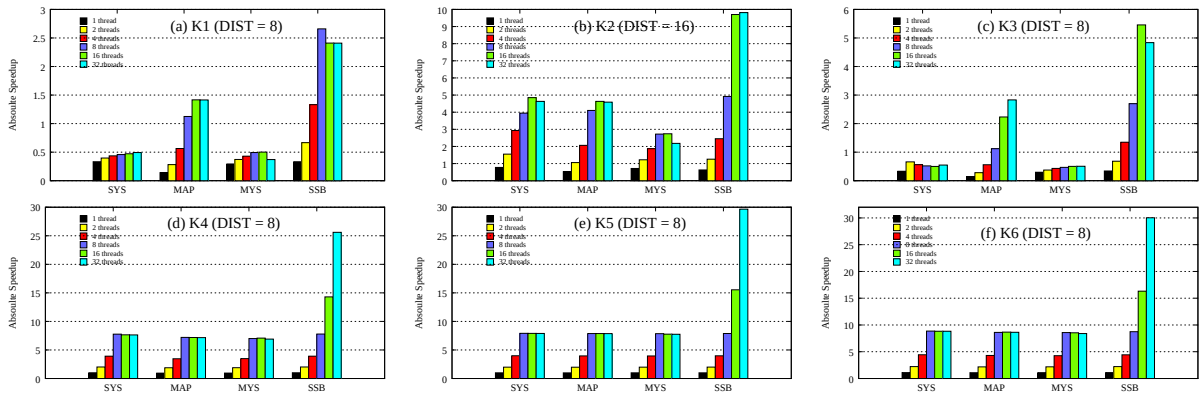


Figure 10: Performance of Multithreaded DOACROSS Kernel Loops. (DIST: dependence distance.)

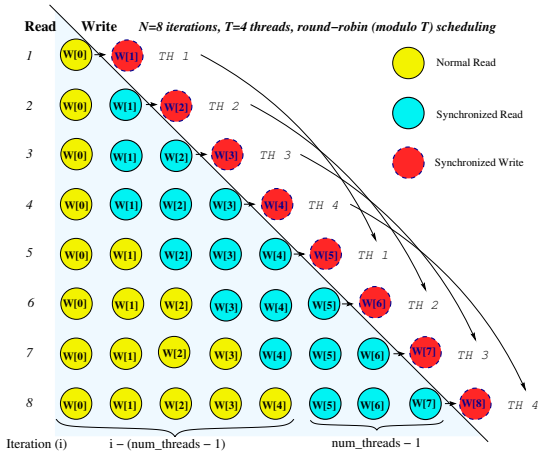


Figure 11: Parallelization and Synchronization of Livermore Loop 6

with simple cross-iteration dependencies. The following two benchmarks illustrates how SSB can help exploiting fine-grain parallelism of applications with complex data dependences, which cannot be easily handled by software methods.

Linear Recurrence Equations (Livermore Loop 6). To parallelize the loop shown in Figure 1(b), we assign the iterations to threads using the round-robin scheduling. The SSB SWMR data synchronization mechanism is used to enforce the data dependences between iterations.

Figure 11 illustrates our synchronization scheme through an example that executes 8 iterations with 4 threads. Thread 1 marks `W[1]` as available via a synchronized write after it completes iteration 1. It then starts iteration 5 according to the scheduling policy. In iteration 5, the computation of `W[5]` depends on `W[1]` to `W[4]`. However, it is not necessary to explicitly wait for `W[1]`, since thread 1 itself computed `W[1]` in iteration 1. Accordingly, thread 2 does not need to check the availability of `W[1]` and `W[2]` when executes iteration 6, since it computed $W[2]$ with the readiness of `W[1]` in iteration 2.

With this scheme, when a thread starts an iteration `i` to compute `W[i]`, it first uses normal load instructions to read from `W[0]` to `W[(i-1)-(T-1)]`, where `T` is the number of threads. Then synchronized reads (`sswmr_r`) are used to fetch the other `T-1` elements of `W`. After completing the iteration, the thread issues a synchronized write (`sswmr_w`) to `W[i]` to inform `T-1` readers. Therefore, the number of synchronization reads and writes required for each iteration does not depend on the problem size, but on the number

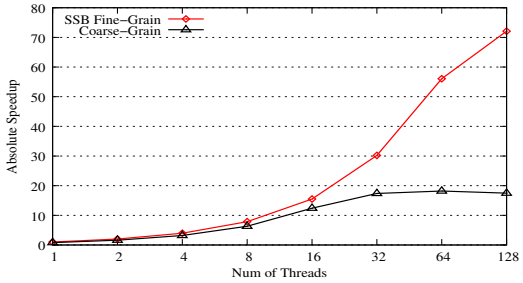


Figure 12: Speedup of Livermore Loop 6

of threads. It is now clear that this application kernel also satisfy Equation 2 ($S(t) \ll M \times B$) introduced in Section 2.

Figure 12 compares the fine-grain parallelization approach with a coarse-grain implementation based on [17]. Both versions are based on a sequential version, whose outer loop has been unrolled 4 times. A speedup is also calculated against this sequential code. By exploiting fine-grain parallelism, the fine-grain data synchronization based approach demonstrate significant performance advantage over the coarse-grain counterpart. For instance, when running with 128 threads, the SSB-based fine-grain approach yields an absolute speedup of 72, a 312% performance improvement over the coarse-grain scheme.

Wavefront Computation. Wavefront is a common computation form in scientific codes. For a 2D matrix with initialized left and top edges, each remaining element is computed from its neighbors to the left, above, and above-left. To parallelize the code, we treat each row of the matrix as a task, and assign rows to threads using round-robin scheduling. As a result, only the presence of the above neighbor of an element need to be checked before computing this element. In our implementation, we group 8 consecutive elements in a row as a block. For each block, a thread only loads/stores the first element via SSB synchronized read (`sswsr_r2`)/write (`sswsr_w2`). The other elements in the block are accessed using normal load/store instructions.

Figure 13 shows the speedup of our parallelization of the wavefront computation on a $1,024 \times 1,024$ matrix. We increase the size of on-chip SRAM in simulator such that the matrix can be stored on-chip. The speedup is calculated against the sequential version with inner loop being unrolled 8 times. The multithreaded implementation of wavefront computation with fine-grain data synchronization demonstrates the capability of the SSB to exploit the parallelism. For example, the SSB-based fine-grain version shows an absolute speedup of 104 when executed with 128 threads.

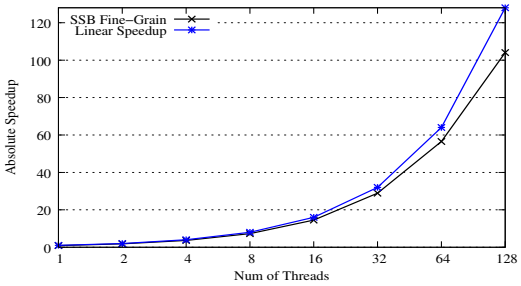


Figure 13: Speedup of Wavefront

Table 4: SSB Synchronization Success Rates and SSB Full Rates

Benchmark	Success Rate		SSB Full Rate	
	64 Threads	128 Threads	64 Threads	128 Threads
Random Access	99.98%	99.96%	0	0
Livermore Loop 13	99.11%	98.42%	0	0
Livermore Loop 14	99.72%	99.59%	0	0
Ordered Integer Set	99.97%	99.93%	0	0.0004%
Livermore Loop 6	87.52%	72.13	0	0
Wavefront	99.86%	99.83%	0	0

5.5 Effectiveness of SSB

In Table 4, we report the percentage of successful synchronizations of all synchronizations issued for 6 benchmarks. We can observe that most fine-grain synchronizations are executed successfully even when number of threads is large. This is preferred because the cost of successful synchronization operation is very low (see Section 5.2). The Livermore Loop 6 has relatively low successful rate compared to others. This is because a 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 10), the synchronization successful rates are also very high. Otherwise, the rates are not high, since a certain portion of synchronized reads are destined to fail at the 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. For a set of multithreaded benchmarks with different workload characteristics, we can see that a small SSB for each memory bank is usually sufficient for recording the synchronization states of active synchronized data units.

6. RELATED WORK

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 [38], Tera [5], MDP [13], Sparcle [3], M-Machine [24], the MT processor in Eldorado [18], and other machines, has been explained in Section 1. I-structure [6] memory system employed in some dataflow model based architectures [6, 23] exploits similar design as full/empty bits based memory system. Tagging each word of the entire memory requires modification to off-the-shelf SRAM or DRAM technologies and introduces significant storage cost. Because of such cost, the number of state bits that can be tagged to a word has to be small, which can only be used to implement limited synchronization functionalities. Because of the small storage cost, SSB can afford to form much larger states in each entry, thus can potentially support more synchronization functionality.

Hardware mechanisms such as QOLB [22], MAOs on SGI Origin [27], lock box [41] for SMT processor, SoC lock cache [4], AMO [45] 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 [24] also allows fast synchronization between three on-chip processors through register-register communication. Sampson et. al. [37] proposed barrier filters, a hardware mechanism for enabling fast barrier synchronization on multi-core chips.

Transactional memory (TM) using non-blocking synchronization is proposed as a replacement to lock-based synchronization [20, 35, 36, 28]. Most hardware TM designs need to extend and modify the existing cache coherence protocols and speculative execution techniques. Our current SSB design relies on blocking synchronization mechanism, and it will be interesting to see how to explore non-blocking synchronization in an SSB-like design.

Finally, various loop optimization techniques have been developed to minimize the amount of fine-grain synchronization for parallelized do-across loops [32, 9, 33, 34]. Those techniques can be combined with SSB-based hardware support to further improve the resulting code, especially when the synchronization resource requirements exceeds the number of SSB entries provided.

7. CONCLUSION AND FUTURE WORK

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.

8. ACKNOWLEDGMENT

This work was supported in part by IBM, ETI, DoD, DoE (DE-FC02-01ER25503), NSF (CNS-0509332), and other gov-

ernment 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.

9. REFERENCES

[1] HPC challenge benchmark. 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*, 13(3):48–61, June 1993.

[4] B. Akgul and V. Mooney. The system-on-a-chip lock cache. *Intl. Journal of Design Automation for Embedded Systems*, 7(1-2):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*, 18(3b):1–6, 1990.

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

[7] 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.

[8] C. Cascaval, J. Castanos, L. Ceze, M. Denneau, and et. al. Evaluation of a multithreaded architecture for cellular computing. In *Procs. of 8th Intl. Symp. on High Performance Computer Architecture*, Boston, MA, 2002.

[9] D.-K. Chen. *Compiler Optimizations for Parallel Loops with Fine-Grained Synchronization*. PhD thesis, UIUC, 1994.

[10] 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.

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

[12] W. J. Dally. Computer architecture in the many-core era. In *Keynote at the 24th Intl. Conf. on Comput. Design*, 2006.

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

[14] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. FAST: A functionally accurate simulation toolset for the Cyclops64 cellular architecture. In *1st Workshop on Modeling, Benchmarking, and Simulation*, Madison, WI, Jun. 2005.

[15] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. Toward a software infrastructure for the Cyclops-64 cellular architecture. In *Procs. of 20th Intl. Symp. on High Performance Computing Systems and Applications*, St. John's, NL, Canada, 2006.

[16] M. Denneau and H. S. Warren, Jr. 64-bit Cyclops: Principles of operation, Apr. 2005.

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

[18] J. Feo and et. al. Eldorado. In *Procs of the 2nd Conf. on Computing frontiers*, pages 28–34, 2005.

[19] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and et. al.. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[20] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Procs. of the 20th Intl. Symp. on Computer architecture*, 1993.

[21] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao. Optimization of dense matrix multiplication on IBM Cyclops-64: Challenges and experiences. In *Procs. of the 12nd Intl. European Conf. on Parallel Processing*, Aug. 2006.

[22] A. Kägi and D. B. J. R. Goodman. Efficient synchronization: Let them eat QOLB. In *Procs. of the 24th Intl. Symp. on Computer Architecture*, pages 170–180, 1997.

[23] 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.

[24] 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 *Procs. of the 25th Intl. Symp. on Computer architecture*, 1998.

[25] 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 Intl. Conf. on Supercomputing*, Cairns, Australia, 2006.

[26] D. Kranz and et. al. Low-cost support for fine-grain synchronization in multiprocessors. Technical Report MIT/LCS/TM-470, 1992.

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

[28] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Procs. of the 33rd Intl. Symp. on Computer Architecture*, 2006.

[29] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. on Computer Systems*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[30] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *the 14th Annual ACM Symp. on Parallel Algorithms and Architectures*, Aug. 2002.

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

[32] S. P. Midkiff and D. Padua. Compiler algorithms for synchronization. *IEEE Trans. on Comput.*, 36(12):1485–1495, 1987.

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

[34] R. Rajamony and A. L. Cox. Optimally synchronizing DOACROSS loops on shared memory multiprocessors. In *Procs. of 1997 Intl. Conf. on Parallel Architectures and Compilation Techniques*, 1997.

[35] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Procs. of the 19th Symp. on Architectural Support for Programming Languages and Operating Systems*. 2002.

[36] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Procs. of the 32nd Intl. Symp. on Computer Architecture*, Jun. 2005.

[37] 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.

[38] B. Smith. The architecture of HEP. In *Parallel MIMD Computation: HEP Supercomputer and Its Applications*, Scientific Computation Series, pages 41–55. MIT Press, Cambridge, MA, 1985.

[39] SPEC. SPEC OpenMP benchmark suite.

[40] 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.

[41] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Procs. of the 5th Intl. Symp. on High-Performance Computer Architecture*, 1999.

[42] 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.

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

[44] D. Yeung and A. Agarwal. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Procs of the 4th ACM Symp. on Principles and practice of parallel programming*, 1993.

[45] 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.