

MiSAR: Minimalistic Synchronization Accelerator with Resource Overflow Management

Ching-Kai Liang Milos Prvulovic
Georgia Institute of Technology
`{chingkai,milos}@cc.gatech.edu`

Abstract

While numerous hardware synchronization mechanisms have been proposed, they either no longer function or suffer great performance loss when their hardware resources are exceeded, or they add significant complexity and cost to handle such resource overflows. Additionally, prior hardware synchronization proposals focus on one type (barrier or lock) of synchronization, so several mechanisms are likely to be needed to support real applications, many of which use locks, barriers, and/or condition variables.

This paper proposes MiSAR, a minimalistic synchronization accelerator (MSA) that supports all three commonly used types of synchronization (locks, barriers, and condition variables), and a novel overflow management unit (OMU) that dynamically manages its (very) limited hardware synchronization resources. The OMU allows safe and efficient dynamic transitions between using hardware (MSA) and software synchronization implementations. This allows the MSA's resources to be used only for currently-active synchronization operations, providing significant performance benefits even when the number of synchronization variables used in the program is much larger than the MSA's resources. Because it allows a safe transition between hardware and software synchronization, the OMU also facilitates thread suspend/resume, migration, and other thread-management activities. Finally, the MSA/OMU combination decouples the instruction set support (how the program invokes hardware-supported synchronization) from the actual implementation of the accelerator, allowing different accelerators (or even wholesale removal of the accelerator) in the future without changes to OMU-compatible application or system code. We show that, even with only 2 MSA entries in each tile, the MSA/OMU combination on average performs within 3% of ideal (zero-latency) synchronization, and achieves a speedup of 1.43X over the software (pthreads) implementation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISCA'15, June 13-17, 2015, Portland, OR USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06\$15.00
DOI: <http://dx.doi.org/10.1145/2749469.2750396>

1. Introduction

Synchronization latency is critical for achieving scalable performance on many-core processors. Numerous hardware mechanisms for low-latency synchronization have been proposed [7, 12, 20] and even used in prototype and commercial supercomputing machines [3, 5, 6, 11, 14, 15, 22]. As general-purpose processors have shifted their focus from solely increasing single-core performance to providing more cores, there has been a renewed interest in hardware support for synchronization [1, 2, 17, 23, 26], this time for a much broader range of systems, programmers, and users. Examples of recently proposed hardware synchronization mechanisms include utilizing a small buffer attached to the on-chip memory controller to perform synchronization and allow trylock support [26], incorporating a lock control unit to both the core and memory controller to allow efficient reader-writer lock [23], or leveraging low latency signal propagation over transmission lines for lock and barrier synchronization [1, 2, 17].

Because previous research on hardware supported synchronization mostly focused on how to reduce synchronization latency, most such work assumed that a sufficient amount of hardware resources is available, with only limited consideration on how to handle limited hardware resources. This, however, limits the adoption of hardware synchronization accelerators both because of high cost (many applications use a large number of synchronization addresses for which resources would be needed) and correctness (some applications can exceed the resources that were considered sufficient at hardware design time).

Most prior work tackles limited hardware resources using one of the following three mechanisms. The simplest mechanism is to have an a-priori partitioning of synchronization addresses into hardware-supported and software-supported ones. Programmers will thus use hardware synchronization instructions for some and software library calls for other synchronization variables. However, this places a heavy burden on programmers because: 1) they must decide which synchronization approach to use for which synchronization variable, 2) debug problems that occur when a synchronization variable erroneously mixes synchronization implementations, and hardware resources are oversubscribed. Another mechanism is to simply stall the synchronization operation until resources are available. Although this does not require any programmer intervention, it can result in great performance loss, or even

deadlock if resources are not sufficient. The third mechanism is to treat the insufficient hardware resources as an exception. The exception handler will then decide to wait and try again, or use a software synchronization mechanism. Such a fallback results in significant performance penalty, so sufficient resources are needed to keep the number of the fallbacks very low. Furthermore, naively falling back to a software implementation can break the synchronization semantics, and additional overheads (and possibly additional hardware mechanisms) are needed to prevent such problems.

Also, previous proposals have focused on only accelerating (supporting) one type of synchronization. This would result in significant hardware cost/complexity to support the overall synchronization needs of real workloads, where different applications (or even the same application) may use locks, barriers, and/or condition variables. Each hardware synchronization mechanism (e.g. one for locks and another for barriers) may have its own software interface and its own verification complexity, which complicates adoption by both hardware architects and by programmers.

In this paper, we propose a minimalistic synchronization accelerator (MSA). The MSA is a distributed synchronization accelerator for tile-based many-core chips. It follows the POSIX pthread synchronization semantics and supports all three common types of synchronization (locks, barriers and condition variables) but has very few entries in each tile. We also propose a novel hardware overflow management unit (OMU) to efficiently manage limited hardware synchronization resources. The OMU keeps track of synchronization addresses that are currently active in software, so we can prevent these addresses from also being handled in hardware. The OMU also enables the accelerator to rapidly allocate/deallocate hardware resources to improve utilization of its (few) entries. Finally, we propose ISA extensions for hardware synchronization. These ISA extensions facilitate adoption by allowing synchronization libraries to only be modified once (to support the new instructions) and then used with any hardware synchronization implementations that support the ISA's fallback semantics – including trivial implementations with no actual hardware synchronization support.

Our results indicate that in a 64-core many-core processor, the OMU allows an MSA with only 2 entries per tile to cover 93% of synchronization operations on average, achieving an average speedup of 1.43X (up to 7.59X in streamcluster!) over the software (pthreads) implementation, and performing within 3% of ideal (zero-latency) synchronization.

In the rest of this paper, we review related work (Section 2), then provide an overview of our hardware support (Section 3), how to use it for each type of synchronization (Section 4), and how to quickly re-acquire low-contention locks (Section 5). We then present our experimental results (Section 6) and our conclusions (Section 7).

2. Related Work

Hardware support for synchronization generally improves synchronization in two ways. First, by implementing the synchronization semantics in hardware, which avoids the inefficiencies in updating the synchronization state in software. Examples include accelerators for barriers [13, 21, 25] that track barrier's arrival state and detect the all-arrived condition without the overhead of updating the arrival count variable in a critical section. Lock accelerators [9, 13, 25, 26] maintain the lock's owned/free state in hardware and thus help arbitrate which requestor is the next to get the lock once it is freed.

The other way is to improve synchronization latency by directly notifying the waiting threads to avoid the coherence “ping-pong” involved in software-only synchronization. For example, a software-only implementation of lock handoff involves sending an invalidation when releasing the lock, waiting for that invalidation to reach all the sharers (typically all cores waiting for that lock), a cache read miss on (at least) the next-to-acquire core, a transfer of the lock's block into that cache, and then sending an upgrade request (invalidation) when actually acquiring the lock. In contrast, a direct-notification lock accelerator [2, 4, 12, 23] typically involves sending a single message from the releasor to the next-acquirer. A similar flurry of coherence activity is involved in signaling barrier release in software, and is avoided in hardware accelerators [1, 7, 17] by directly signaling the barrier release to waiting cores.

Synchronization support has also been used in distributed supercomputer systems, e.g., efficient broadcast networks have been used to accelerate barrier operations [15, 18], and fetch-and-add operations have been used for efficient barrier counting [5, 11, 22]. Our work focuses on more tightly coupled many-core systems, and provides support for all three common types of synchronization.

In addition to improving synchronization latency, most hardware synchronization accelerators also have to handle what happens when hardware resources of the accelerator are exhausted. Several solutions have been adopted in prior work, such as requiring programmers to manually partition synchronization variables [1, 4, 7, 9, 17] into those that always use software and those that always use hardware, using the memory as a resource buffer [23], or switching to a software exception handler [26]. Unfortunately, programmer-implemented partitioning is not portable to architectures that have fewer resources, use of main memory complicates the implementation and adds latency, and software exception handlers are difficult to implement correctly and can incur large overhead when fallbacks are too frequent. Utilizing the memory as a resource buffer can reduce the amount of software exception events [23], however, still requires the exception handler to resize the resource table. In addition, going to main memory to access the resource buffer increases the overall synchronization latency. In contrast, our approach uses a small OMU local to each tile to efficiently and correctly fall back to an

Work	Synchronization Primitives	Notification	Resource overhead	Dedicated Network	Resource Overflow
Lock Table[9]	Lock	Indirect	$O(N_{lock})$	No	SW
AMO[25]	Lock, Barrier	Indirect	0	No	N/A
Tagged Memory[13]	Lock, Barrier	Indirect	$O(N_{mem})$	No	N/A
QOLB[12]	Lock	Direct	$O(N_{core})$	No	SW
SSB[26]	Lock	Indirect	$O(N_{activeLock})$	No	SW
LCU[23]	Lock	Direct	$O(N_{core})$	No	HW/SW
barrierFilter[21]	Barrier	Indirect	$O(N_{barrier})$	No	Stall
Lock Cache[4]	Lock	Direct	$O(N_{lock}N_{core})$	Yes	Stall
GLocks[2]	Lock	Direct	$O(N_{lock})$	Yes	None
bitwiseAND/NOR[7]	Barrier	Direct	$O(N_{barrier})$	Yes	None
GBarrier[1]	Barrier	Direct	$O(N_{barrier})$	Yes	None
TLSync[17]	Barrier	Direct	$O(N_{barrier})$	Yes	None
MSA/OMU (Our proposal)	Lock, Barrier, Condition Variable	Direct	$O(N_{core})$	No	HW

Table 1: Summary of hardware synchronization approaches

existing (e.g. pthreads) software implementation when needed, while also improving utilization of the (very limited) hardware synchronization resources. A more detail discussion of our scheme verses software exception handler will be discussed in Section 3.2.

Table 1 summarizes the prior proposals for multi-core synchronization: which synchronization types they support, whether they provide direct notification, the hardware cost (in terms of added state), whether a specialized network is required, and how hardware resources overflow is managed. For resource overflow, SW corresponds to simply falling back to a software handler when resources are exhausted, whereas HW resolves it in hardware. For LCU [23], it will first fallback to the memory and only if memory overflows will it require a software handler, thus we mark it HW / SW.

In general, accelerators that provide direct notification support only one type of synchronization (e.g. only lock or only barrier), and direct-notification barrier proposals mostly rely on dedicated networks. Also, many of the mechanisms require recording state information that is proportional to the number of locks or barriers in the system - potentially many different locations, especially for programs that use large arrays of locks. In addition, so far no barrier accelerator has tackled the problem of resource overflow. In contrast, our proposed approach supports all three types of synchronization (locks, barriers, and condition variables), with direct notification over the existing on-chip interconnect, and with $O(N_{core})$ hardware resource overhead.

3. Low-Cost Hardware Synchronization

Our proposed MSA is designed for a tiled many-core chip, where each tile contains a core and its local caches, a network-on-chip (NoC) router, a slice of the last-level cache (LLC) and coherence directory, and a slice of the synchronization accelerator. However, it can be adapted for use in other settings, e.g. those with broadcast interconnects (buses), centralized instead of distributed LLCs, etc.

A single slice of our synchronization accelerator is shown in Figure 1. It contains a (small) number of synchronization entries, and each entry tracks synchronization state of a single

synchronization address. An entry in the MSA is a global “clearing house” for all synchronization operations for that particular address. To simplify interactions with coherence, we distribute the entire MSA according to the coherence home of the synchronization address: if an MSA entry is associated with a synchronization address, that entry must be in the LLC home tile of that synchronization address.

Each MSA entry contains the synchronization address it is associated with and a valid (V) bit. It also contains what type of synchronization it is currently used for, a bit vector (HWQueue), and an auxiliary information field. The HWQueue utilizes one bit per core to record which cores are waiting on that synchronization address, and also the lock owner in case of locks. The use of the auxiliary field depends on synchronization type, as will be explained later. Here we assume that each core runs only one thread. To support hardware multi-threading, the HWQueue would be augmented to have 1-bit per hardware thread. Note that, even with 64 cores and 2 threads per core, the overall state of a single-entry MSA would be less than 264 bits (33 bytes) in each of the 64 tiles.

Software interacts with the MSA using a set of 6 instructions, each corresponding to a synchronization operation (LOCK, BARRIER, COND_WAIT, etc.). Each instruction has a return value that is either SUCCESS, FAIL, or ABORT. The instruction returns SUCCESS when the synchronization operation was successfully performed, FAIL when the operation cannot be performed in hardware, and ABORT when the operation was terminated by MSA due to OS thread scheduling. A more detailed discussion of ABORT, and how it differs from

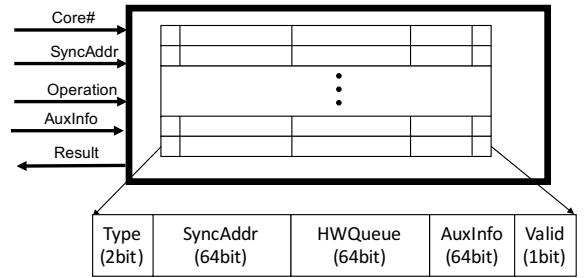


Figure 1: Minimalistic Synchronization Accelerator (MSA)

`FAIL`, is provided in Section 4. To simplify integration into the processor pipeline and to simplify interaction with memory consistency, each synchronization instruction acts as a memory fence and its actual synchronization activity begins only when the instruction is the next to commit. We fully model the resulting pipeline stalls in our experiments and find that they are negligible in most applications.

3.1. Allocate/Deallocate MSA Entry

An MSA entry is allocated (if available) when a synchronization “acquire” request (`LOCK`, `BARRIER`, or `COND_WAIT`) is received by the home of the synchronization address. The entry is evicted when its HWQueue becomes empty, i.e. when no thread waits for or owns the lock, when the barrier is released, or when no thread waits for a condition variable. As indicated earlier, if no MSA entry is available, the MSA simply returns `FAIL`, which results in using software implementation for the synchronization operation.

The MSA does not allocate a new entry for “release” requests (`UNLOCK`, `COND_SIGNAL` and `COND_BCAST`), so they fail if a matching entry is not found. This helps ensure that, if an acquire-type operation used a software implementation (`LOCK`, `BARRIER`, or `COND_WAIT` returned `FAIL`), a release will also “default-to-software”.

3.2. Overflow Management Unit (OMU)

The Overflow Management Unit (OMU) ensures correct synchronization semantics when an MSA entry is not available. The OMU keeps track of the synchronization addresses that currently have waiting (or lock-owning) threads in software. The OMU consists of a small set of counters indexed (without tagging) by the synchronization address. Once a thread’s acquire-type synchronization operation falls back to software, the counter that corresponds to the synchronization address will be incremented. The counter is decremented when the operation completes (for locks, when the lock is released). When an acquire-type operation does not find an MSA entry, we find the OMU counter for that address to check if an MSA entry can be allocated for it (OMU counter is zero), or if the operation must be done in software to maintain correctness (OMU counter is $>$ zero). Note that this requires the entering and exiting of the synchronization operation to be visible to the OMU. For locks, the entering and exiting results in attempting `LOCK/UNLOCK` instructions (which `FAIL` because that lock is handled in software). For barriers and condition variables, entering is similarly exposed to hardware (a *FAILED* `BARRIER` or `COND_WAIT` instruction). However, barrier and condition wait that complete in software would normally not be visible to hardware, so we add the `FINISH` instruction at the end of software barrier and condition wait to inform the OMU that the operation has completed (so it can decrement the corresponding counter).

The accelerator only grants new hardware resources (allocates an MSA entry) for an “acquire” request when there is no

already “active” (waiting or lock-owning) synchronization on that address in software. To illustrate why this is necessary, consider a synchronization accelerator that has already *FAILED* several `LOCK` requests for a given address because MSA resources were not available. As a result, the lock is currently owned by a thread and multiple threads are waiting for it in software. Meanwhile, an MSA entry becomes available. Then a new `LOCK` request for the same variable would allocate an MSA entry. As far as the MSA knows, the lock is free so it would be granted to this thread, thus breaking lock semantics – two threads are in the critical section, one granted entry by the software fallback and one by the MSA. The OMU prevents this situation because the counter that corresponds to the lock is non-zero as long as any thread is owning or waiting for the lock in software. When a new request is made, the non-zero counter in the OMU steers the request safely to software. Only when the thread becomes free (no thread owns it or waits for it) in software will it become eligible for MSA entry allocation when the next request is made. For high-contention locks, this may keep the lock in software for a long time. However, in all the benchmarks we used, such continuous-requests activity eventually has a “lull” in requests that allows the software activity to drain out, allowing the MSA to be used on the next burst of activity. In our evaluation we have only seen one application that shows noticeable performance degradation from this problem. In most of the cases, bursts of activity on the same lock, even when steered to software, usually “drain out” relatively quickly and allow the lock to be given an MSA entry (if one is available).

Since the OMU uses a small number of counters without tagging them, different synchronization addresses may alias to the same counter. This potentially affects performance – a synchronization variable may unnecessarily be steered to software instead of granted an MSA entry. This can be avoided by using enough OMU counters, or even using counting Bloom filters instead of simple counters. However, the aliasing in the OMU does not affect correctness – the variable that is unnecessarily steered to software cannot have any MSA-handled operations already in progress. This is because a synchronization request always first checks the MSA. If an entry is found, the operation proceeds in hardware (no OMU lookup). The OMU lookup occurs only when the MSA entry is *not* found. Therefore, a synchronization address that already has an MSA entry will continue to use the MSA until the HWQueue becomes empty and the MSA entry is freed. This makes it possible for a synchronization variable to keep owning an MSA entry by continuously making acquire requests on that variable. Just like for software-steered streaks of requests, this is not a correctness problem and in the benchmarks we used it is also not a significant performance problem.

Note that one could eliminate OMU entirely by simply allocating/deallocating when an entry is initialized and destroyed. However, this significantly reduces the coverage of the accelerator, since from our evaluation, some applications will

use more than thousands of locks. In addition, the problem becomes even more problematic when there are multiple applications. An application may end up occupying all the entries while being suspended, thus leaving active applications with no hardware resources to use.

Several proposals have opted to use a software handler solution for hardware resource overflow [23, 26]. They utilize two bits (FBIT/SBIT) to record the status of each slice of accelerator. FBIT is set/cleared when the accelerator is full/empty, and SBIT is set/cleared when there are active entries in the software. In order to provide atomicity, the software handler must acquire a per-slice lock first. In addition, the status of the accelerator needs to be re-checked because, by the time the software handler acquires the lock, the accelerator's state may have already been changed. This adds latency to each lock operation when no matching entry is found in hardware, so resource overflow needs to be very rare. Additionally, special instructions are required to let the software handler insert an entry back into hardware, which adds complexity and requires message exchange between the core and the accelerator. In contrast, our OMU resolves resource overflow locally, and provides graceful performance degradation when resources overflow.

4. Synchronization Primitives

4.1. Lock Synchronization

Lock acquire/release is requested by a program through LOCK/UNLOCK instructions. The LOCK instruction results in sending a request to the MSA in the home tile of the synchronization address. If an MSA entry is already allocated for this address, or if an entry can be allocated, the HWQueue bit for the requesting core would be set to 1. If no other HWQueue bit is one, the accelerator returns a SUCCESS message, and the LOCK instruction returns SUCCESS, which indicates that the requesting core has acquired the lock. If the lock is currently held by another core, the HWQueue bit for that other core would be 1 and the requesting thread would not be granted the lock. In this case, the MSA simply delays the response. This prevents the requesting core's LOCK instruction from being committed, stalling its core until the lock is obtained.

An UNLOCK instruction also sends a message to the accelerator, which clears the cores' bit in the HWQueue and checks the remaining bits. If any other bit is set, one of them is selected and MSA responds to that core with a SUCCESS message. That core's LOCK instruction now returns SUCCESS (it acquired the lock) while the others in the HWQueue continue to wait. To ensure fairness, the MSA in each tile maintains one (for the entire MSA, not for each entry) next-bit-to-check (NBTC) register. When more than one waiting core is found in the HWQueue after an UNLOCK, the next core to release is selected starting at the NBTC position and the NBTC register is updated to point to the bit-position after the released one.

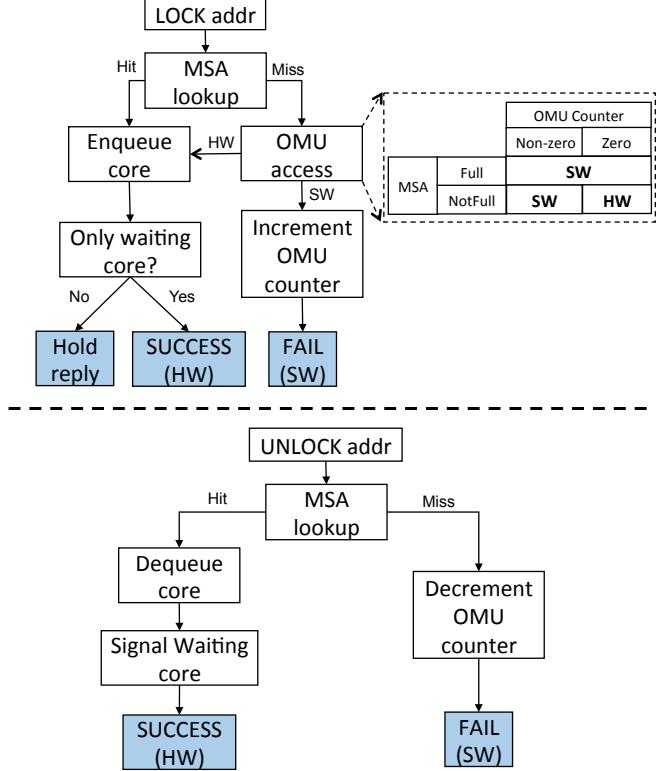


Figure 2: State Diagram for Lock/Unlock Operations

4.1.1. MSA/OMU State Diagram: Figure 2 shows the state diagram of the synchronization accelerator for lock/unlock operations. Once a LOCK request is received, it first checks if a matching entry exists in MSA. A hit in MSA will result in handling the lock operation in hardware. A miss, however, will result in querying the OMU. If the MSA is not full and the OMU counter is zero, then a new entry is inserted into MSA and thus result in utilizing the hardware accelerator. Otherwise the OMU counter is incremented and the request is responded with a FAIL message.

For UNLOCK, if a matching MSA entry is found, then the UNLOCK operation is performed by MSA. Otherwise the OMU counter is decremented and the request is responded with a FAIL message.

4.1.2. Thread Suspension, Migration, and Interrupts: When the core is interrupted for context-switching (or any other reason) while the instruction at the top of the ROB is a LOCK instruction, a SUSPEND request is sent to the lock's MSA. Upon receiving the SUSPEND request, the MSA clears the corresponding bit in the HWQueue, dequeuing the core from the lock's waiting list. When the thread is resumed on this (or another) core, it re-executes the LOCK instruction. Recall that a LOCK instruction that is not at the head of the ROB has not sent its request to the MSA yet, so it is simply squashed and, when the thread continues execution, re-executed.

The situation is slightly different when the thread that owns the lock is suspended. In this case, the MSA will not be notified because the LOCK instruction has already completed

(retired). Other threads in the HWQueue continue to wait (because the lock is still held by the suspended thread). When the thread is resumed, eventually it executes an UNLOCK instruction that sends a message to the MSA. If the thread resumes on the same core, the MSA will behave correctly – it clears that core’s bit in HWQueue and signals the next waiting core. However, if the thread resumes on another core, the UNLOCK request will come from a core that does not have the HWQueue bit set. In this scenario, the MSA does not know which core originally issued the LOCK request – it is one of the cores whose HWQueue bits are 1, but we do not know which one. To resolve this situation, the MSA simply replies that the UNLOCK was successful, then replies to all cores in the HWQueue with an ABORT message, frees the MSA entry, and increments the OMU counter by the appropriate amount. This causes all waiting threads to fall back to a software lock implementation. Note that at this point the lock is free and has no threads waiting in hardware, so it is safe to fall back to software. Since our proposed mechanism has very little overhead when falling back to a software lock, this sacrifices the opportunity for hardware acceleration but does not incur a noticeable overhead beyond that.

```

Lock (*lock) {
1 result = LOCK lock; /* execute HW lock inst */
2 if result==FAIL || result==ABORT then
3 | pthread_mutex_lock(lock)
4 end
5 }

6 Unlock (*lock) {
7 result = UNLOCK lock; /* execute HW unlock inst */
8 if result==FAIL then
9 | pthread_mutex_unlock(lock)
10 end
11 }

```

Algorithm 1: Modified Lock/Unlock Algorithm

4.1.3. Algorithm: Algorithm 1 shows the lock algorithm adapted to use the MSA. We execute the LOCK instruction first. If this instruction succeeds, the lock was obtained in hardware and the thread proceeds into the critical section. If the LOCK instruction returns FAIL (or ABORT), we fall back to the software lock algorithm. For this fall-back, we simply use pthread_mutex_lock algorithm, but any other software lock algorithm can be substituted. The unlock operation is adapted similarly to first try to use the MSA and fall back to software if the hardware UNLOCK fails.

Interestingly, this ISA can trivially be supported by failing all LOCK/UNLOCK instructions, with little overhead (see Section 6) compared to code that uses the software-only pthread implementation directly. This always-fail possibility is an important feature of our approach - it allows our ISA and synchronization library changes to be implemented without committing the processor designers to perpetual future MSA/OMU (or any other) support for synchronization.

4.2. Barrier Synchronization

When the BARRIER instruction is executed, similarly to the LOCK instruction, a request is sent to the MSA home tile and the corresponding HWQueue bit is set if an matching MSA entry is found. This request contains the barrier’s “goal” count, which the MSA entry stores in the AuxInfo field. When the “goal” number of bits are set in the HWQueue, all those cores are sent SUCCESS responses. If the barrier cannot be handled in hardware¹,the accelerator immediately returns FAIL and the requesting core must fall back to a software implementation of barrier synchronization.

As with locks, hardware and software are prevented from simultaneously implementing the same barrier. Without this, a few arriving threads may be handled in software (e.g. because no MSA entry is available), and the rest of the arriving threads may be handled by the MSA (an entry became available). In this scenario, neither the software barrier implementation nor the MSA would ever reach the barrier’s target count, which would deadlock all the threads that participate in that barrier.

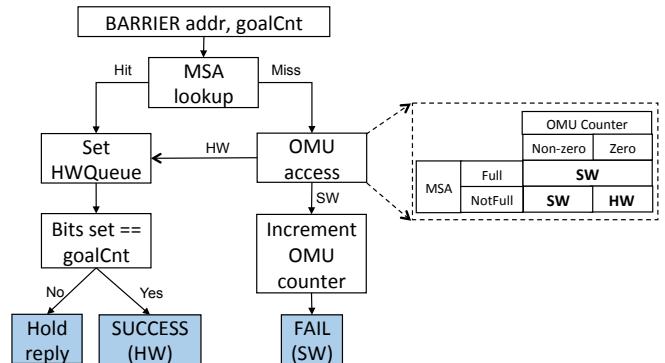


Figure 3: State Diagram for the Barrier Operation

4.2.1. MSA/OMU State Diagram: Figure 3 shows the state diagram of the synchronization accelerator for the barrier operation. When it receives a BARRIER message, the MSA checks for a matching entry. If such an entry is found, the corresponding HWQueue bit is set and, if enough HWQueue bits are set, the barrier is released (send SUCCESS to all participating cores). If no matching MSA entry is found, the OMU is queried and we either allocate a new MSA entry or return FAIL.

4.2.2. Thread Suspension, Migration, and Interrupts: When a thread is interrupted while waiting at the barrier, the BARRIER instruction would be at the top of the ROB which results in the core sending a SUSPEND request to the MSA tile. However, unlike locks which will simply dequeue the requesting core, for barriers we send FAIL (or ABORT) responses to all participating cores, i.e. we force the barrier to fall back to software.

¹Because no MSA entry is available, or because the OMU indicates that other threads have already arrived in software

We note that it might be possible to handle thread suspend/migration in a more efficient way. An additional counter would be added to count inactive-but-arrived-to-barrier threads, and this counter would also need to be decremented when the thread resumes execution. Another source of complexity would be to ensure that all threads are correctly notified when the barrier is released – even those threads that are absent (suspended) when the last thread arrives to the barrier. This requires the hardware accelerator to keep track of which threads have been signaled and which have not yet been signalled. The approach we use (fall back to software) reduces both hardware cost and its verification complexity.

```

1 Barrier (*barr) {
2     result=BARRIER barr, goal_count; /* execute HW barrier inst */
3     if(result==FAIL || result==ABORT then
4         pthread_barrier_wait(barr);
5         FINISH barr; /* notify OMU of exiting barrier */
6     }

```

Algorithm 2: Modified Barrier Algorithm

4.2.3. Algorithm: Algorithm 2 shows the barrier code adapted to use the hardware accelerator. Like for locks, the modification involves trying the hardware synchronization first and falling back to software if that fails. The only major difference is that, once the software barrier implementation exits, we send a FINISH request to the OMU in the barrier’s home node. This ensures the OMU to keep track of how many threads remain within the software barrier code. This FINISH instruction was not needed for locks because the exit notification was provided by the UNLOCK instruction. For barriers, a failed BARRIER instruction only indicates the entry into the software implementation, but the exit from the software barrier can be many cycles later (when all threads have arrived).

4.3. Condition Variable

Condition variables are supported through COND_WAIT, COND_SIGNAL, and COND_BCAST instructions. We follow standard POSIX condition variable semantics, where a wait operation waits for signals/broadcasts but also temporarily (while waiting) unlocks the associated lock.

A COND_WAIT request involves sending an UNLOCK request to the lock’s home tile while enqueueing the core in the HWQueue for the condition variable. The enqueueing of a core is accomplished by setting the corresponding bit in the HWQueue. No response is sent until the core is released (by COND_SIGNAL or COND_BCAST). When no MSA entry is available for the condition variable, a FAIL response is sent back, so the COND_WAIT instruction returns FAIL, and the condition variable wait must be implemented in software.

A COND_SIGNAL instruction sends a message to the MSA home of the condition variable. If a matching MSA entry is found, SUCCESS is returned to the signaling thread, and one of the waiting cores from the HWQueue is selected for wakeup

and its HWQueue bit is cleared. The next step is to re-acquire the lock that was released when that core began waiting, so we send a LOCK request to the lock’s home on behalf of the waiting core. The lock home tile will then respond to the waiting core with a SUCCESS message when it eventually acquires the lock, and the COND_WAIT instruction on that core returns SUCCESS.

The COND_BCAST instruction is similar, except that it results in waking up all cores in the HWQueue, not just one. This results in multiple LOCK requests to the lock’s home tile where each has to wait to actually be granted the lock.

In our hardware condition variable implementation, the condition variable’s home tile sends the LOCK request and the lock’s MSA responds to the waiting core only when the lock is acquired. The associated lock address is thus stored in the AuxInfo field when receiving the COND_WAIT request. The advantage of this approach is that the COND_WAIT instruction, if successful, completes the entire condition wait operation. Another option would be to separate the condition wait into the “release lock and wait for signal/broadcast” and “re-acquire the lock we released”, i.e. to have the condition variable’s home respond directly to the waiting core with SUCCESS when the signal/broadcast is received, and require the lock to be re-acquired by executing a LOCK instruction. We do not use this alternative to avoid including “under the hood” workings of synchronization implementation in the ISA definition.

If no MSA entry is found for the condition variable, the home responds to the COND_SIGNAL and COND_BCAST messages with a FAIL response. When the corresponding signal/broadcast instruction completes with a FAIL result, the thread implements the signal/broadcast operation in software.

4.3.1. MSA/OMU State Diagram: Figure 4 shows the state diagram of MSA for handling a condition wait operation. Once a COND_WAIT request is received, it first checks if a matching entry exists in MSA. A hit in the MSA will result in handling the condition variable operation in hardware, whereas a miss will result in querying the OMU.

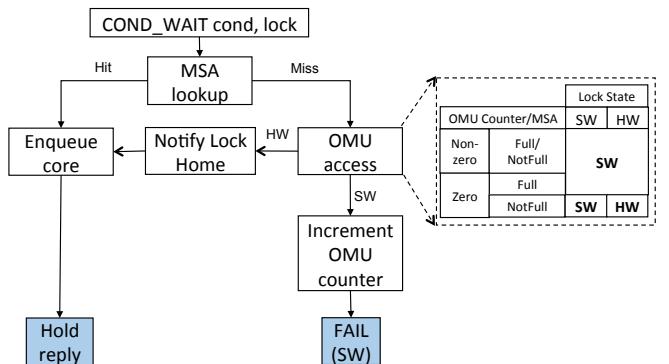


Figure 4: State Diagram for the Condition Wait Operation

For OMU access, additional lock state information is used to determine the OMU response. If both the lock and the

condition variable can be handled in hardware, a new entry is allocated for the condition variable. This ensures that, if a condition variable is implemented in hardware, its associated lock is also implemented in hardware. If the lock is handled in software, then condition variables that uses that lock will be handled in software, too. This avoids the relatively complex corner case when the condition variable is handled in hardware but its lock is handled in software. The implementation of the condition wait in this case would require additional synchronization (using an auxiliary lock) to ensure correctness, which would in turn require breaking up the `COND_WAIT` instruction into sub-operations such as “non-blocking enqueue” and “wait for signal/broadcast” and additional complexity to handle the potential failure of each such instruction.

In Figure 4, the OMU indicates “HW” when it is safe to insert a new entry into the MSA. To make this decision, it needs to know whether the lock has (or can get) an MSA entry in its own home tile. Recall that for `COND_WAIT` requests, the condition variable’s home sends an unlock request to the lock’s home, and that an MSA entry for a condition variable is allocated when a `COND_WAIT` message is received and no MSA entry already matches it. Thus, when the condition variable’s home gets a `COND_WAIT` request with no already-matching MSA entry, it first checks if an MSA entry is available. If not, it responds with `FAIL`. If an entry is available, it is reserved (but not yet allocated), and a special “unlock and pin entry” (`UNLOCK&PIN`) message is send to the lock’s home. When it receives the `UNLOCK&PIN` message, the lock’s home performs a normal `UNLOCK` attempt. If it fails, a `FAIL` response is sent to the condition variable’s home, which frees the reserved MSA entry and returns `FAIL` for the `COND_WAIT` operation. If the `UNLOCK` succeeds for the `UNLOCK&PIN` request, the lock’s home pins the lock’ MSA entry so it cannot be deallocated (even if its HWQueue is empty) as long as the condition variable has an MSA entry, and then returns `SUCCESS` in response to the `UNLOCK&PIN` request. When the response is received by the condition variable’s home, it changes the reserved MSA entry into an allocated one and continues with its `COND_WAIT` operation normally.

When the condition variable’s home releases a core from its HWQueue, recall that this results in sending the lock’s home a `LOCK` request to re-acquire the lock that was released when entering the `COND_WAIT` operation. If this was the last core in the HWQueue, the condition variable’s MSA entry becomes free. To notify the lock’s home that the condition variable no longer requires the lock to be pinned to its MSA entry, the `LOCK` request sent to the lock’s home in this situation is changed into a special `LOCK&UNPIN` request. When this request is received by the lock’s home, it decrements the lock’s AuxInfo counter and then processes the `LOCK` part of the request.

The pinning of lock MSA entries is implemented by tracking (in the lock’s AuxInfo field) how many condition variables are currently “pinning” this lock. This counter is incre-

mented when the `UNLOCK&PIN` request succeeds and is decremented when the `LOCK&UNPIN` request arrives. Note that the `LOCK&UNPIN` request always succeeds because, when the request arrives at the lock’s home, the lock is pinned (AuxInfo is non-zero) to its MSA entry.

4.3.2. Thread Suspension, Migration, and Interrupts:

When a thread is interrupted while waiting at the condition variable, it returns without re-acquiring the lock. First, the core will send a `SUSPEND` request to the home MSA of the synchronization (condition variable) address. Upon receiving the `SUSPEND` request, the MSA removes the thread from its HWQueue and sends an `ABORT` response back. Note that this is very similar to releasing a waiting thread, except that we respond directly to the requestor without obtaining the lock. The fallback for the `ABORT` result of a `COND_WAIT` instruction is to re-acquire the lock (using Algorithm 1) and then execute a `FINISH` instruction. Note that the suspended/migrated/interrupted thread completes the `COND_WAIT` instruction and only continues to execute the fallback code when it begins to run again.

If no signal/broadcast events have actually occurred by the time the thread re-acquires the lock and exits its condition wait library call, the end result is a spurious wakeup of that thread. However, spurious wakeups of `cond_wait` are allowed by its POSIX semantics for very similar reasons to ours – a thread that needs to handle a signal (like `SIGQUIT`, `SIGTERM`, and other interrupt-like events, not `cond_signal`) needs to exit `pthread_cond_wait` prematurely and thus has a spurious wakeup. The spurious wakeup possibility requires use of a while loop that re-checks the condition when the condition wait returns. If the re-check fails, recall that it still holds the lock that it re-acquired when exiting the spuriously-successful condition wait. Thus the thread can safely call the condition wait again. The essential property of this is that a condition signal/broadcast must wake up thread(s) that is waiting for it, but a thread can also be woken up even if no signal/broadcast has occurred.

Interestingly, it is possible to implement condition variables in software in such a way that eliminates the possibility of spurious wakeups. A common implementation of this approach uses timestamps to track when the last broadcast and the last “wasted” signal (no thread woken up) occurred. It is possible to use our `COND_WAIT` instruction under such semantics, but it requires the reading of these timestamps prior to attempting to do a condition wait in hardware (`COND_WAIT` instruction). When the instruction is aborted and the condition variable’s associated lock is re-acquired, the timestamps would be checked again to see if we should succeed and return (signal/broadcast did occur since our wait originally began) or go back to waiting.

4.3.3. Algorithm: Algorithm 3 show the modified condition_wait and condition signal/broadcast. Similar to barriers, if the condition variable is handled in software and a thread has been signalled, it also needs to send a `FINISH` message

to MSA to decrement the OMU counter. Unlike locks and barriers, condition variables handle the FAIL and ABORT cases separately. As described in Section 4.3.2, an ABORT results in re-acquiring the lock and (possibly spuriously) returning control to the application.

```

CondWait (*cond, *lock) {
1 result = COND_WAIT cond, lock; /* execute HW cond_wait inst */
2 if result==FAIL then
3     | sw_cond_wait(cond, lock);
4     | FINISH cond;           /* notify OMU of exiting condition
5     | variable */
6 end
7 else if result==ABORT then
8     | LOCK(lock);
9     | FINISH cond;           /* notify OMU of exiting condition
10    | variable */
11 end
12 CondSignal (*cond) {
13 release = COND_SIGNAL cond;
14 if release==FAIL then
15     | sw_cond_signal(cond)
16 end
17 }
18 CondBroadcast (*cond) {
19 release = COND_BCAST cond;
20 if release==FAIL then
21     | sw_cond_broadcast(cond)
22 end
23 }
24 }
```

Algorithm 3: Modified Condition Variable Algorithms

We use “sw_cond_wait” as our software fallback algorithm instead of the original pthread function `pthread_cond_wait`. This is because the pthread function internally calls the software lock operations. Our `sw_cond_wait` implementation is identical to `pthread_cond_wait`, except that the lock operations it calls are the hardware-with-software-fallback lock/unlock functions from Algorithm 1. This is needed because, while we prevent a condition variable from using the MSA if its lock is implemented in software, it is possible for the condition variable to be implemented in software while its lock is implemented in hardware. Therefore, the software implementation of `cond_wait` needs to use the *Lock/Unlock* function defined in Algorithm 1.

5. Optimization

Programs access the MSA through a set of synchronization instructions that send requests to the synchronization address’s home tile. In the event of the operation cannot be performed in hardware, this will add an on-chip round-trip latency before it falls back to software synchronization.

For barriers and condition variables, this round-trip overhead is small compared to the overall latency of the software implementation. For locks, however, the software fallback can have low latency if the lock variable was previously owned by the same core and still resides in the core’s private (e.g. L1) cache. In that case, the lock can be acquired in software with-

out any coherence traffic. In such cases, the added round-trip latency to consult the home’s MSA/OMU adds an overhead that is not negligible relative to the latency of the software fallback alone. A potential optimization would be to profile the application and identify locks that are both low-contention and acquired quickly (L1 hit), and not attempt to use the hardware for such locks at all. However, we prefer solutions that avoids placing additional burdens on application programmers (our modified synchronization algorithms only require changes to the synchronization part of the runtime library). Therefore, we propose an optimization that allows skipping many doomed-to-succeed MSA/OMU checks transparently to both synchronization library and application programmers.

The optimization uses the presence of a (writable) cache block that contains the synchronization address as an proxy for “can acquire the lock without informing the home”. When the hardware accelerator grants the lock ownership to a core, along with replying the request with a “SUCCESS” message, it also grants the core an exclusive ownership (E state in the MESI protocol) of the cache block, invalidating any other cached instances of this block.

Upon receiving the cache block and (successfully) completing the `LOCK` instruction, the core will put the block in its L1 cache and set the “HWSync” bit (a new bit that is added to each line in the cache) for its cache line. This bit indicates that the core was the last one to successfully complete a hardware lock operation for that cache line. In contrast, a normal read or write request will bring in the cache block without setting the “HWSync” bit. Note that, since the synchronization accelerator resides with the home node of the cache block, it can easily retrieve the cache state information of a particular cache block and cause the block to be sent along with the response.

The `UNLOCK` instruction does not clear the HWSync bit. When that core issues the next `LOCK` request, if its L1 cache still has the cache block with “HWSync” equal to one, the core can send a `LOCK_SILENT` notification to the home tile of the lock but its `LOCK` instruction can return “SUCCESS” immediately. This notifies the MSA that the core has re-acquired the lock, allowing the MSA entry to be updated, but avoids adding the round-trip latency if the lock is quickly re-acquired by the same thread that held it previously.

6. Evaluation

We evaluate synchronization approaches using SESC [19], a cycle-accurate architectural simulator. We model 16-core and 64-core processors, with 2-issue out-of-order cores and private IL1 and DL1 caches. The L2 cache is a distributed shared last-level cache, so each core has a slice of the L2 cache and a router for the packet-switched 2D mesh network-on-chip (NoC). We model the NoC using Booksim [10], a cycle-accurate NoC simulator that we integrated into SESC.

In our evaluation MSA/OMU- N models our hardware synchronization with an N -entry MSA and a four-counter OMU in each slice. We also evaluate MSA-0 configuration, which

does not have any hardware synchronization support and trivially implements our instructions by always returning FAIL (without sending a message to the home node). This configuration is used with the same modified synchronization library, so it shows how much overhead would be added by these modified algorithms in a machine that does not provide actual MSA/OMU hardware, e.g. if the new instructions are adopted to exploit our MSA/OMU hardware and then this hardware is eliminated in some future versions of the processor. Another configuration we evaluate is MSA-*inf* where we model a MSA with an infinite number of entries (so no OMU is needed). This configuration provides insight into how much performance is lost due to limited MSA size.

The benchmarks we use are the (entire) Splash2 [24] and PARSEC [8] benchmark suites. All benchmarks are complied with the GCC 4.6.3 compiler suite using -O3 optimization. For non-baseline runs, we replace the pthread synchronization library calls with more advanced software implementations (MCS lock and tournament barrier [16]), synchronization library that utilizes algorithms discussed in Section 4, along with different types of MSA (MSA-0/*inf*, or MSA/OMU), depending on the synchronization approach used in that run.

6.1. Raw Synchronization Latency

Figure 5 shows the raw cycle count directly attributable to synchronization, excluding the waiting time that would be present even with an ideal (zero-latency) synchronization. Note that this figure uses a logarithmic scale.

We model the no-contention case for locks using disjoint sets of locks in different threads, and measure the time between entering and exiting the `lock()` function. All synchronization approaches perform similarly in this case, except for MSA/OMU-2; because for no-contention, our HWSync-bit optimization scheme results in most `LOCK` instructions to succeed without waiting for the MSA’s response. This avoids both the overheads of software implementations and the round-trip latency of a non-optimized hardware implementation. The high-contention case is modeled by having all threads access the same lock. Lock handoff is measured from the cycle in which a thread enters `unlock()` to the cycle in which the released `lock()` exits. In this case, pthread_mutexlock and spinlock have high handoff latency with a poor scaling trend (from 16 to 64 cores). The more scalable MCS lock has a faster

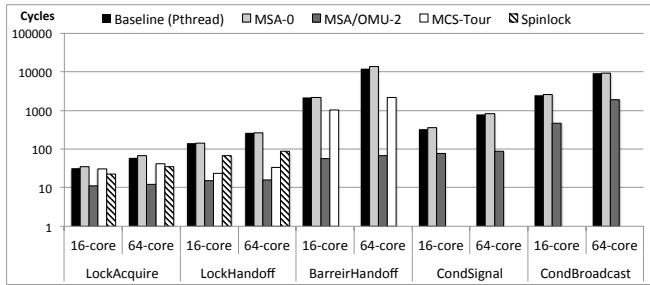


Figure 5: Raw Synchronization Latency

handoff and scales better than the pthreads lock implementations. With high contention, our MSA/OMU-2 configuration does not benefit from the HWSync-bit optimization, but nevertheless has the lowest handoff latency and best scaling trend because the MSA implements lock handoff efficiently.

For barriers we measure latency from the time that the last-arriving thread enters `barrier()` to the time all threads have exit. Our MSA/OMU approach provides an order-of-magnitude improvement over the best software implementation (tournament barrier).

For condition variables, the latency is measured from entering `cond_signal()` or `cond_broadcast()` to the exit from the released `cond_wait()`. The MSA/OMU-2 configuration improves significantly over the software-only implementation. Part of the reason for this improvement is from improving the latency of condition variable notifications, but another reason for the improvement is that MSA/OMU-2 also provides quick handoff of the lock associated with the condition variable.

In all these cases, MSA-0 incurs a minimum overhead compared to the baseline (pthread) scheme. This shows that our modifications to the synchronization library do not result in significant overheads when using the fallback path, i.e. if a processor does not have hardware support, it can trivially implement our ISA extensions and use the same hardware-capable synchronization code. This may be an important consideration for processor manufacturers - after adding the synchronization instructions and our MSA/OMU hardware, the processor manufacturer can drop MSA/OMU support in future generations of the processor without breaking compatibility with software that uses the new instructions.

6.2. Benchmark Evaluation

Figure 6 shows the overall application speedup, relative to the pthread baseline, for Splash and PARSEC. The averages shown are for *all 26 benchmarks in Splash and PARSEC suites*, but to reduce clutter we show in the figures only those individual applications where *Ideal* synchronization shows at least 4% benefit compared to the baseline.

The MSA-*inf* results are on average within 3% of the Ideal (zero-latency) case. Where differences are noticeable, they mostly come from message latencies to and from the synchronization variable’s home. The difference is largest in 64-core execution of *radiosity*, where lock synchronization is frequent, but with many low-contention locks. Furthermore, each lock tends to be used by different threads, so our HWSync-bit optimization hides the round-trip communication latency for only 20% of lock acquire requests. For *fluidanimate*, the difference between MSA-*inf* and *Ideal* is 8%. This application also has frequent operations on low-contention locks, but each lock tends to be used by the same core, allowing our HWSync-bit optimization to hide round-trip communication for 90% of lock requests.

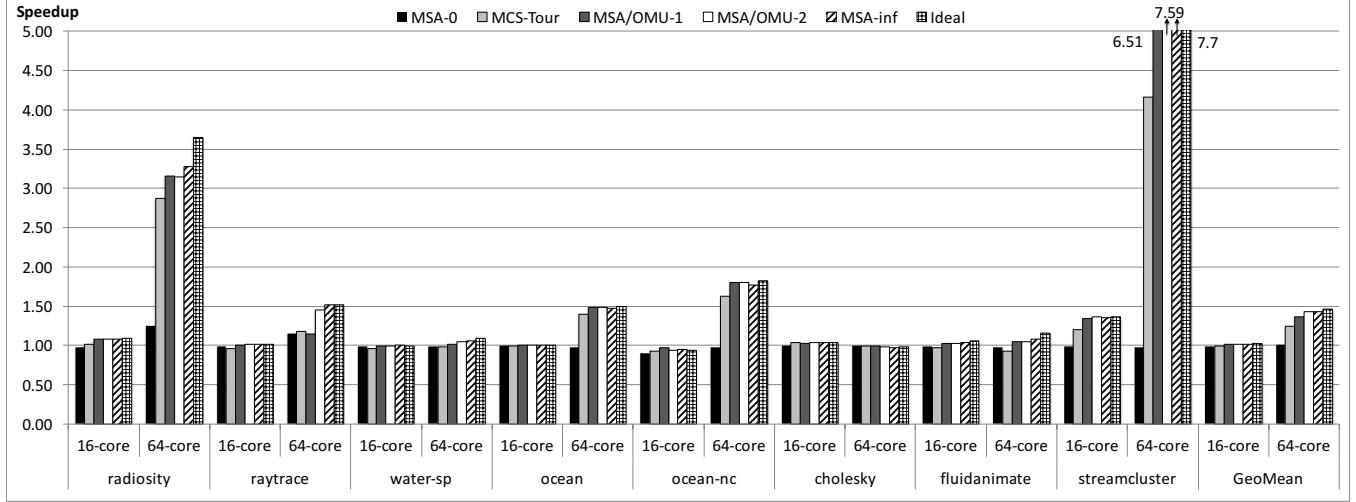


Figure 6: Overall application performance improvement (Speedup)

Interestingly, *Ideal* synchronization in *ocean-nc* with 16 threads performs worse than the software baseline. We verified that the time spent on synchronization is dramatically improved (only the necessary waiting time remains) in *Ideal*, but the non-synchronization code executes with a lower IPC, primarily due to increased burstiness of cache misses (all threads leave the barrier in the exact same cycle). This “better is worse” effect is also present (to a lesser degree) in other synchronization-accelerated configurations.

Among realistic hardware implementations, *MSA/OMU-1* configuration achieves average performance within 6% of the *MSA-inf*, and *MSA/OMU-2* performs similar to *MSA-inf*. We conclude that, with the OMU, few MSA entries are needed to achieve most of the hardware-synchronization performance potential.

The *MSA-0* results are within 1% of the baseline software implementation. This confirms that our synchronization library and the ISA modifications can be implemented across entire processor families, even if some processors in those families have no actual MSA/OMU hardware. Another interesting point is for *radiosity* and *raytrace*, *MSA-0* actually shows speedup compared to the baseline. For *radiosity*, the speedup comes from the reduction of empty task queue searches, which results in 47% decrease of lock accesses. For *raytrace*, the amount of lock access did not show any significant changes. However, the average lock handoff latency for the most-contended lock was reduced by 2X. This difference comes from the changes in lock acquire order, which would affect the lock handoff latency under our distributed shared last-level cache with non-uniform cache-to-cache transfer latency.

Finally, *MCS-Tour* benefits applications with high-contention locks or frequent barrier operations. For *fluidanimate*, *MCS-Tour* shows some performance loss because MCS locks have larger overhead for no-contention locks. On average, *MCS-Tour* shows a 24% speedup, but *MSA/OMU*

achieves an additional 19% speedup over this advanced software implementation.

6.3. Coverage Improvement from OMU

Figure 7 shows the percentage of synchronization operations, averaged across all Splash-2 and PARSEC benchmarks, handled by the MSA with and without the OMU. Without the OMU, MSA entries cannot be safely deallocated, so the very first synchronization variables that are used by the application are the ones that get MSA entries (and keep them “forever”). We observe a significant increase in coverage of synchronization operations with the OMU. For 64-tile MSA-2, for example, the OMU allows 93% of operations to utilize the MSA, compared to only 56% without the OMU. More importantly, the OMU naturally handles the transition from using one set of variables to another, e.g. when one application ends and another begins. Without the OMU, a separate mechanism would be needed to inform the MSA when the synchronization variable address that allocated an entry is no longer used for synchronization.

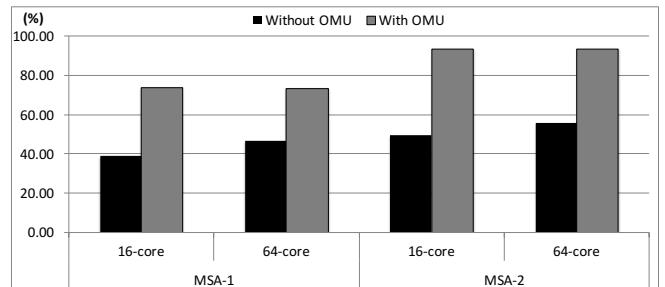


Figure 7: Coverage of Synchronization Operations

6.4. Lock Optimization

Figure 8 shows the speedup achieved in *fluidanimate* with and without the HWSync-bit optimization. Recall that the opti-

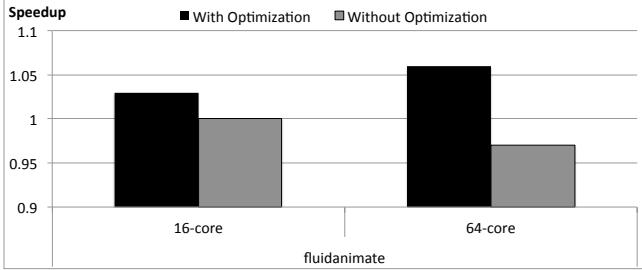


Figure 8: Effect of HWSync-bit optimization on Fluidanimate

mization allows a core to acquire a lock which it previously held (if its block is still in the L1 cache) without waiting for the lock home’s response. *fluidanimate* uses many locks, but has low lock contention because each lock tends to be acquired by the same thread repeatedly. Without the HWSync-bit optimization, it is often the case that the software lock (that hits in the L1 cache) has lower latency than the hardware one (request to MSA, wait for response). This increased latency cancels out the gains which is provided by MSA/OMU, which leads to a slowdown in a 64-core machine. With the HWSync-bit optimization, the hardware locks are uniformly lower-latency than software implementation, so MSA/OMU performance shows a speedup versus a software implementation, and this speedup increases with the number of cores.

6.5. Synchronization Breakdown

Figure 9 shows the speedup for supporting only one type of synchronization (locks or barriers) by the MSA in a 64-core machine. For barrier-intensive applications such as *ocean-nc* and *streamcluster*, the speedup is lost when MSA only supports locks. For lock-intensive applications, such as *radiosity* and *fluidanimate*, most or all of the speedup is lost when only supporting barriers. Interestingly, *raytrace* is a lock-intensive application, but it shows a lower speedup for *MSA-LockOnly* than for *MSA-BarrierOnly*. This is because, in *MSA-LockOnly*, the absence of barrier handling results in different allocation of MSA entries, causing one of the more contended locks to suffer more software fallback. However, the speedup becomes similar to *MSA/OMU* when we increase the MSA entries from 2 to 4.

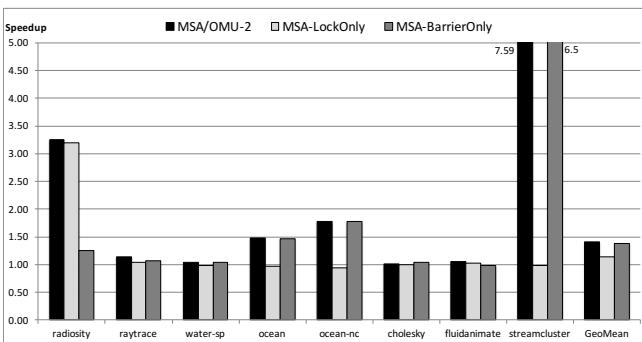


Figure 9: Speedup comparison when MSA only supports lock or barrier operation

7. Conclusion

Numerous hardware synchronization accelerators have been proposed in the past that focuses on accelerating synchronization operations. However, they normally assume sufficient amount of hardware resources to limit the overhead in handling resource overflow. In addition, most work supports only one type of operation, which significantly increases the total cost of providing overall synchronization support for most applications needs.

In this paper, we have proposed *MiSAR*, a minimalistic synchronization accelerator (MSA) that supports the three commonly used synchronizations, along with a small and efficient overflow management unit (OMU) that safely and dynamically manages the MSA’s limited hardware resources. Our results indicate that in a 64-core processor, the OMU allows a 2-entry-per-tile MSA to service 93% of synchronization operations on average, achieving an average speedup of 1.43X (up to 7.59X in *streamcluster*!) over the software (pthreads) implementation, and performing within 3% of ideal (zero-latency) synchronization.

8. Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1320717. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

- [1] J. Abellán, J. Fernández, and M. Acacio, “A g-line-based network for fast and efficient barrier synchronization in many-core cmps,” in *Parallel Processing (ICPP), 2010 39th International Conference on*, Sept 2010, pp. 267–276.
- [2] J. Abellán, J. Fernández, and M. Acacio, “Glocks: Efficient support for highly-contended locks in many-core cmps,” in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 893–905.
- [3] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, “The mit alewife machine: architecture and performance,” in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA ’95. New York, NY, USA: ACM, 1995, pp. 2–13. Available: <http://doi.acm.org/10.1145/223982.223985>
- [4] B. S. Akgul, J. Lee, and V. J. Mooney, “A system-on-a-chip lock cache with task preemption support,” in *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, ser. CASES ’01. New York, NY, USA: ACM, 2001, pp. 149–157. Available: <http://doi.acm.org/10.1145/502217.502242>
- [5] G. Almási, C. Archer, J. G. Castaños, J. A. Gunnels, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. Ratterman, B. D. Steinmacher-Burow, W. Gropp, and B. Toonen, “Design and implementation of message-passing services for the blue gene/l supercomputer,” *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 393–406, march 2005.
- [6] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, “The tera computer system,” in *Proceedings of the 4th international conference on Supercomputing*, ser. ICS ’90. New York, NY, USA: ACM, 1990, pp. 1–6. Available: <http://doi.acm.org/10.1145/77726.255132>
- [7] C. J. Beckmann and C. D. Polychronopoulos, “Fast barrier synchronization hardware,” in *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, ser. Supercomputing ’90. Los

- Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 180–189. Available: <http://dl.acm.org/citation.cfm?id=110382.110433>
- [8] C. Bienia, “Benchmarking modern multiprocessors,” Ph.D. dissertation, Princeton University, January 2011.
- [9] M.-C. Chiang, “Memory system design for bus-based multiprocessors,” Ph.D. dissertation, Madison, WI, USA, 1992, uMI Order No. GAX92-09300.
- [10] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.
- [11] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, “The nyu ultracomputer—designing a mimd, shared-memory parallel machine (extended abstract),” in *Proceedings of the 9th annual symposium on Computer Architecture*, ser. ISCA ’82. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982, pp. 27–42. Available: <http://dl.acm.org/citation.cfm?id=800048.801711>
- [12] A. Kägi, D. Burger, and J. R. Goodman, “Efficient synchronization: let them eat qolb,” in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA ’97. New York, NY, USA: ACM, 1997, pp. 170–180. Available: <http://doi.acm.org/10.1145/264107.264166>
- [13] S. Keckler, W. Dally, D. Maskit, N. Carter, A. Chang, and W. Lee, “Exploiting fine-grain thread level parallelism on the mit multi-alu processor,” in *Computer Architecture, 1998. Proceedings. The 25th Annual International Symposium on*, jun-1 jul 1998, pp. 306 –317.
- [14] J. Laudon and D. Lenoski, “The sgi origin: a ccnuma highly scalable server,” in *Proceedings of the 24th annual international symposium on Computer architecture*, ser. ISCA ’97. New York, NY, USA: ACM, 1997, pp. 241–251. Available: <http://doi.acm.org/10.1145/264107.264206>
- [15] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak, “The network architecture of the connection machine cm-5 (extended abstract),” in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA ’92. New York, NY, USA: ACM, 1992, pp. 272–285. Available: <http://doi.acm.org/10.1145/140901.141883>
- [16] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for scalable synchronization on shared-memory multiprocessors,” *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991. Available: <http://doi.acm.org/10.1145/103727.103729>
- [17] J. Oh, M. Prvulovic, and A. Zajic, “Tlsync: Support for multiple fast barriers using on-chip transmission lines,” in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, june 2011, pp. 105 –115.
- [18] F. Petrini, J. Fernandez, E. Frachtenberg, and S. Coll, “Scalable collective communication on the asci q machine,” in *High Performance Interconnects, 2003. Proceedings. 11th Symposium on*, aug. 2003, pp. 54 – 59.
- [19] J. Renau, B. Fraguera, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, “Sesc simulator, january 2005.”
- [20] J. T. Robinson, “A fast general-purpose hardware synchronization mechanism,” in *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, ser. SIGMOD ’85. New York, NY, USA: ACM, 1985, pp. 122–130. Available: <http://doi.acm.org/10.1145/318898.318910>
- [21] J. Sampson, R. González, J.-F. Collard, N. P. Jouppi, M. Schlansker, and B. Calder, “Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers,” in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 235–246. Available: <http://dx.doi.org/www.library.gatech.edu:2048/10.1109/MICRO.2006.23>
- [22] S. L. Scott, “Synchronization and communication in the t3e multiprocessor,” in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-VII. New York, NY, USA: ACM, 1996, pp. 26–36. Available: <http://doi.acm.org/10.1145/237090.237144>
- [23] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero, “Architectural support for fair reader-writer locking,” in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’43. Washington, DC, USA: IEEE Computer Society, 2010, pp. 275–286. Available: <http://dx.doi.org/10.1109/MICRO.2010.12>
- [24] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: characterization and methodological considerations,” in *Proceedings of the 22nd annual international symposium on Computer architecture*, ser. ISCA ’95. New York, NY, USA: ACM, 1995, pp. 24–36. Available: <http://doi.acm.org/10.1145/223982.223990>
- [25] L. Zhang, Z. Fang, and J. Carter, “Highly efficient synchronization based on active memory operations,” in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, april 2004, p. 58.
- [26] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao, “Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures,” in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA ’07. New York, NY, USA: ACM, 2007, pp. 35–45. Available: <http://doi.acm.org/10.1145/1250662.1250668>