Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications*

José F. Martínez† and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801 USA
http://iacoma.cs.uiuc.edu

ABSTRACT

Barriers, locks, and flags are synchronizing operations widely used by programmers and parallelizing compilers to produce race-free parallel programs. Often times, these operations are placed suboptimally, either because of conservative assumptions about the program, or merely for code simplicity.

We propose *Speculative Synchronization*, which applies the philosophy behind Thread-Level Speculation (TLS) to explicitly parallel applications. Speculative threads execute past active barriers, busy locks, and unset flags instead of waiting. The proposed hardware checks for conflicting accesses and, if a violation is detected, the offending speculative thread is rolled back to the synchronization point and restarted on the fly. TLS's principle of always keeping a *safe thread* is key to our proposal: in any speculative barrier, lock, or flag, the existence of one or more safe threads at all times guarantees forward progress, even in the presence of access conflicts or speculative buffer overflow. Our proposal requires simple hardware and no programming effort. Furthermore, it can coexist with conventional synchronization at run time.

We use simulations to evaluate 5 compiler- and hand-parallelized applications. Our results show a reduction in the time lost to synchronization of 34% on average, and a reduction in overall program execution time of 7.4% on average.

1 INTRODUCTION

Proper synchronization between threads is crucial to the correct execution of parallel programs. Popular synchronization operations used by programmers and parallelizing compilers include barriers, locks, and flags. For example, parallelizing compilers typically use global barriers to separate sections of parallel code. Also, programmers frequently use locks and barriers in the form of M4 macros [22] or OpenMP directives [5] to ensure that codes are race-free.

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM, Intel, and Hewlett-Packard.

†José F. Martínez is currently with the Computer Systems Laboratory, Cornell University, Ithaca, NY 14853 USA.

ASPLOS X 10/02 San Jose, CA, USA

Often times, synchronization operations are placed conservatively. This happens when the programmer or the compiler cannot determine whether code sections will be race-free at run time. For example, data may be accessed through a hashing function where conflicts are only occasional. Conservative synchronization may also be used for simplicity, if disambiguation is possible but requires an effort that the programmer or the compiler cannot afford. Unfortunately, conservative synchronization may come at a performance cost when it stalls threads unnecessarily. In these cases, we would like threads to execute the synchronized code without stalling.

Recent research in Thread-Level Speculation (TLS) has proposed mechanisms for optimistically executing nonanalyzable serial codes in parallel (e.g. [4, 10, 12, 19, 23, 31]). Under TLS, special support checks for cross-thread dependence violations at run time, and forces offending speculative threads to squash and restart on the fly. At all times, there is at least one *safe thread*. While speculative threads venture into unsafe program sections, the safe thread executes code nonspeculatively. As a result, even if all the speculative work is useless, forward progress is guaranteed by the safe thread.

In this paper, we propose *Speculative Synchronization*, which applies the philosophy behind TLS to *explicitly parallel* (rather than serial) applications. Speculative threads go past active barriers, busy locks, and unset flags instead of waiting. The hardware monitors for conflicting accesses. If a violation is detected, the offending speculative thread is rolled back to the synchronization point and restarted on the fly.

TLS's principle of always keeping a safe thread is key to our proposal. In any speculative barrier, lock, or flag, the existence of one or more safe threads at all times guarantees forward progress, even in the presence of access conflicts and speculative buffer overflow. This fact, plus the support for speculative barriers and flags, sets our proposal apart from lock-free optimistic synchronization schemes of similar hardware simplicity, such as Transactional Memory [16] and Speculative Lock Elision [27]. In these schemes, which only apply to critical sections, the speculative mechanism by itself does not guarantee forward progress (Section 7).

Speculative Synchronization requires simple hardware: one bit per line and some simple logic in the caches, plus support for register checkpointing. Moreover, by retargeting high-level synchronization constructs to use this hardware (e.g. M4 macros or OpenMP directives), Speculative Synchronization can be made transparent to programmers and parallelizing compilers. Finally, Speculative Synchronization is fully compatible with conventional synchronization and can coexist with it at run time.

Overall, our evaluation of 5 compiler- and hand-parallelized applications shows promising results: the time lost to synchronization is reduced by 34% on average, while the overall program execution time is reduced by 7.4% on average.

This paper is organized as follows: Section 2 outlines the con-

cept of Speculative Synchronization; Section 3 describes its implementation and Section 4 its software interface; Section 5 presents the evaluation environment and Section 6 the evaluation itself; Section 7 compares our approach to two relevant lock-free optimistic synchronization schemes, and proposes Adaptive Speculative Synchronization; finally, Section 8 describes other related work.

2 CONCEPT

2.1 Thread-Level Speculation

TLS extracts speculative threads from serial code and submits them for execution in parallel with a safe thread. Speculative threads venture into unsafe program sections. The goal is to extract parallelism from the code.

TLS is aware of the order in which such program sections would run in a single-threaded execution. Consequently, threads are assigned epoch numbers, where the lowest one corresponds to the safe thread. As threads execute, the hardware checks for cross-thread dependence violations. For example, if a thread reads a variable and, later on, another thread with a lower epoch number writes it, a true dependence has been violated. In this case, the offending reader thread is squashed and restarted on the fly. In many TLS systems, name dependences never cause squashes.

Speculative threads keep their unsafe memory state in buffers; in many TLS proposals, processor caches fulfill this role. If a thread is squashed, its memory state is discarded. If, instead, all the thread's predecessors complete successfully, the thread becomes safe. Then, it can merge its memory state with that of the system.

When the memory state of a speculative thread is about to overflow its buffer, the thread simply stalls. It will either resume execution when all its predecessors complete successfully, or restart from the beginning if it gets squashed before then. However, the safe thread never gets squashed due to dependences or buffer overflow. Therefore, forward progress of the application is guaranteed.

2.2 Speculative Synchronization

Speculative Sychronization applies TLS's concepts to *explicitly parallel* (rather than serial) codes. The goal is to enable extra concurrency in the presence of conservatively placed synchronization and, sometimes, even when data access conflicts between threads do exist. To attack the problem, we limit the scope of TLS's concepts in two ways. First, we do not support any ordering among *speculative* threads; instead, we use a single epoch number for all of them. Second, in addition to true dependences, we trigger a squash even when name dependences are violated across threads. These two limitations simplify the hardware substantially.

Under Speculative Sychronization, threads are allowed to speculatively execute past active barriers, busy locks, and unset flags. Under conventional synchronization, such threads would be waiting; now they are allowed to execute unsafe code. However, every lock, flag, and barrier has one or more *safe* threads: in a lock, the lock owner is safe; in a flag, the producer is safe; in a barrier, the lagging threads are safe. Safe threads cannot get squashed or stall due to speculation. Therefore, forward progress of the application is guaranteed.

A synchronized region is concurrently executed by safe and speculative threads, and the hardware checks for cross-thread dependence violations. As in TLS, as long as dependences are not violated, threads are allowed to proceed. Access conflicts between safe and speculative threads are not violations if they happen *in order*, i.e. the access from the safe thread happens before the access from the speculative one. Any *out-of-order* conflict between a safe and a speculative thread causes the squashing of the speculative thread and its rollback to the synchronization point. No ordering exists among speculative threads; thus, if two speculative threads issue conflicting accesses, one of them is squashed and rolled back to the synchronization point. Overall, since safe threads can make progress regardless of the success of speculative threads, performance in the worst case is still in the order of conventional synchronization.

Speculative threads keep their memory state in caches until they become safe. When a speculative thread becomes safe, it *commits* (i.e., makes visible) its memory state to the system. The circumstances under which a speculative thread becomes safe differ between locks, flags, and barriers; we explain this next. Finally, if the cache of a speculative thread is about to overflow, the thread stalls and waits to becomes safe.

In the following, we limit our discussion to deadlock-free parallel codes; codes that can deadlock at run time are out of our scope. Furthermore, without loss of generality, we assume a release consistency model. Adapting to stricter models is trivial. Moreover, Speculative Sychronization remains equally relevant in these models because barriers, locks, and flags are widely used in all cases.

2.3 Speculative Locks

Among the threads competing for a speculative lock, there is always one safe thread—the lock owner. All other contenders venture into the critical section speculatively. Figure 1*(a)* shows an example of a speculative lock with five threads. Thread A found the lock free and acquired it, becoming the owner. Thread A is, therefore, safe. Threads B, C, and E found the lock busy and proceeded into the critical section speculatively. Thread D has not yet reached the acquire point and is safe.

The existence of a lock owner has implications. The final outcome has to be consistent with that of a conventional lock in which the lock owner executes the critical section atomically *before* any of the speculative threads. (In a conventional lock, the speculative threads would be waiting at the acquire point.) On the one hand, this implies that it is correct for speculative threads to consume values produced by the lock owner. On the other hand, speculative threads cannot commit while the owner is in the critical section. In the figure, threads B and E have completed the critical section and are executing code past the release point,[1] and C is still inside the critical section. All three threads remain speculative as long as A owns the lock.

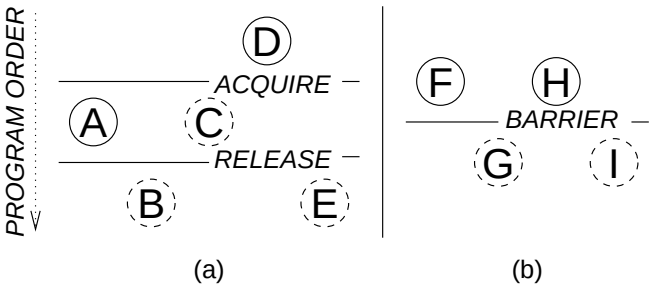

Figure 1: Example of a speculative lock *(a)* and barrier *(b)*. Dashed and solid circles denote speculative and safe threads, respectively.

Eventually, the lock owner (thread A) completes the critical section and releases the lock. At this point, the speculative threads that have also completed the critical section (threads B and E) can immediately become safe and commit their speculative memory state.

[1] The fact that B and E have completed the critical section is remembered by the hardware. We describe the implementation in detail later.

They do so *without acquiring the lock*. This is race-free because these threads have completely executed the critical section and did not have conflicts with the owner or other threads. On the other hand, the speculative threads still inside the critical section (only thread C in our case) compete for ownership of the lock. One of them acquires the lock, also becoming safe and committing its speculative memory state. The losers remain speculative.

The action after the release is semantically equivalent to the following scenario under a conventional lock: after the release by the owner, all the speculative threads past the release point, one by one in some nondeterministic order, execute the critical section atomically; then, one of the threads competing for the lock acquires ownership and enters the critical section. In Figure 1*(a)*, this corresponds to a conventional lock whose critical section is traversed in (A, B, E, C) or (A, E, B, C) order.

2.4 Speculative Flags and Barriers

Flags and barriers are one-to-many and many-to-many synchronization operations, respectively. Flags are variables produced by one thread and consumed by others. Under conventional synchronization, consumers test the flag and proceed through only when it reflects permission from the producer. Under Speculative Synchronization, a flag whose value would normally stall consumer threads, instead allows them to proceed speculatively. Such threads remain speculative until the right flag value is produced, at which point they all become safe and commit their state. Forward progress is guaranteed by the producer thread remaining safe throughout.

Conceptually, barriers are equivalent to flags where the producer is the last thread to arrive. Under Speculative Synchronization, threads arriving to a barrier become speculative and continue (threads G and I in Figure 1*(b)*). Threads moving toward the barrier remain safe (threads F and H) and, therefore, guarantee forward progress. When the last thread reaches the barrier, all speculative threads become safe and commit their state.

3 IMPLEMENTATION

Speculative Synchronization is supported with simple hardware that we describe in this section. In the following, we start by describing the main hardware module. After that, we explain in detail how it works for single and multiple locks first, and then for flags and barriers.

3.1 Speculative Synchronization Unit

The main module that we use to support Speculative Synchronization is the *Speculative Synchronization Unit (SSU)*. The SSU consists of some storage and some control logic that we add to the cache hierarchy of each processor in a shared memory multiprocessor. The SSU physically resides in the on-chip controller of the local cache hierarchy, typically L1+L2 (Figure 2). Its function is to offload from the processor the operations on one synchronization variable, so that the processor can move ahead and execute code speculatively.

The SSU provides space for one extra cache line at the L1 level, which holds the synchronization variable under speculation. This extra cache line is accessible by local and remote requests. However, only the SSU can allocate it. The local cache hierarchy (L1+L2 in Figure 2) is used as the buffer for speculative data. To distinguish data accessed speculatively from the rest, the SSU keeps one *Speculative* bit per line in the local cache hierarchy. The Speculative bit for a line is set when the line is read or written speculatively. Lines whose Speculative bit is set cannot be displaced beyond the local cache hierarchy.

The SSU also has two state bits called *Acquire* and *Release*. The

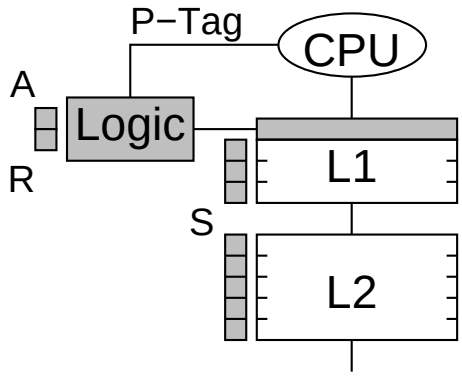

Figure 2: The shaded areas show the Speculative Synchronization Unit (SSU) in a two-level cache hierarchy. The SSU consists of a Speculative (S) bit in each conventional line in the caches, an Acquire (A) and a Release (R) bit, an extra cache line, and some logic.

Acquire and Release bits are set if the SSU has a pending acquire and release operation, respectively, on the synchronization variable. Speculative, Acquire, and Release bits may only be set if the SSU is active, i.e. it is handling a synchronization variable. When the SSU is idle, all these bits remain at zero.

Overall, we can see that the SSU storage requirements are modest. For a 32KB L1 cache and a 1MB L2 cache with 64B lines, the SSU needs about 2KB of storage.

3.2 Supporting Speculative Locks

To describe how the SSU works for locks, we examine lock request, lock acquire, lock release, access conflict, cache overflow, and *exposed SSU* operation.

3.2.1 Lock Request

While we can use different primitives to implement a lock acquire operation, without loss of generality, in this paper we use Test&Test&Set (T&T&S). Figure 3 shows a T&T&S loop on lock *loc1*. In the example and in the rest of the paper, a zero value means that the lock is free.

```
L: ld $1,loc1  ; S1
   bnz $1,L    ; S2
   t&s $1,loc1 ; S3
   bnz $1,L    ; S4
```

Figure 3: Example of Test&Test&Set operation.

When a processor reaches an acquire, it invokes a library procedure (Section 4) that issues a request to the SSU with the address of the lock. At this point, SSU and processor proceed independently as follows:

SSU Side

The SSU sets its Acquire and Release bits, fetches the lock variable into its extra cache line, and initiates a T&T&S loop on it to obtain lock ownership (Figure 3). If the lock is busy, the SSU keeps spinning locally on it until the lock is updated externally and a coherence message is received. (In practice, the SSU need not actually "spin"—since it sits in the cache controller, it can simply wait for the coherence message before retrying.)

Processor Side

As in TLS, in order to allow quick rollback of squashed threads, we need to checkpoint the architectural register state at the beginning of the speculative section. We envision this to be done with a checkpoint mark or instruction that backs up the architectural register map [35], or the actual architectural register values.[2] The checkpoint instruction is included in the library procedure for acquire (Section 4), right after the request to the SSU. No flushing of the pipeline is needed.

The processor continues execution into the critical section. Memory system accesses by the processor after the acquire in program order are deemed speculative by the SSU for as long as the Acquire bit is set. The SSU must be able to distinguish these accesses from those that precede the acquire in program order. One conservative method to ensure this is to insert a memory fence prior to performing the checkpoint. Unfortunately, such an approach, while correct, would have too high a performance cost.

Instead, we use processor hints similar to the way ASI address tags extend memory addresses issued by SPARC processors [33]. In our case, we only need a single bit, which we call the *Processor Tag*, or *P-Tag* bit. The P-Tag bit is issued by the processor along with every memory address, and is fed into the SSU (Figure 2). Our checkpointing instruction is enhanced to also reverse the P-Tag bit of all memory operations that follow it in *program* order. This way, the processor can immediately proceed to the critical section and the SSU can still determine which memory accesses are before and which are after the checkpoint in program order. The SSU then sets the Speculative bit only for lines whose accesses have been tagged appropriately by the P-Tag bit. Note that this mechanism does not impose any restriction in the order in which the processor issues accesses to memory.

With this support, cache lines accessed speculatively are marked without affecting performance. Note that when a thread performs a first speculative access to a line that is dirty in any cache, including its own, the coherence protocol must write back the line to memory. This is necessary to keep a safe copy of the line in main memory. It also enables the conventional Dirty bit in the caches to be used in combination with the Speculative bit to mark cache lines that have been speculatively written.

At any time, if the thread is squashed (Section 3.2.4), the processor completes any nonspeculative work, flushes the pipeline, flash-invalidates all dirty cache lines with the Speculative bit set, flash-clears all Speculative bits, and restores the checkpointed register state. To perform the flash-invalidate and flash-clear operations, we need special hardware that does each of them in at most a few cycles. More details are given in Section 3.2.4.

3.2.2 Lock Acquire

The SSU keeps "spinning" on the lock variable until it reads a zero. At this point, it attempts a T&S operation (statement *S3* in Figure 3). If the operation fails, the SSU goes back to the spin-test. However, if the T&S succeeds, the local processor becomes the lock owner. This is the case for thread C in the example of Section 2.3 after thread A releases the lock. In this case, the SSU completes action: it resets the Acquire bit and flash-clears all Speculative bits, effectively turning the thread safe and committing all cached values. At this point, the SSU becomes idle. Other SSUs trying to acquire the lock will read that the lock is owned.

There is one exception to this mechanism when, at the time the lock is freed by the owner, the speculative thread has already completed its critical section. We address this case next.

[2]Backing up architectural register values could be done in a handful of cycles and would free up potentially valuable renaming registers.

3.2.3 Lock Release

The processor executes a release store to the synchronization variable when all the memory operations inside the critical section have completed.[3] Under Speculative Synchronization, if the lock has already been acquired by the SSU and, therefore, the SSU is idle, the release store completes normally. If, instead, the SSU is still trying to acquire ownership for that lock, the SSU intercepts the release store and takes notice by clearing its Release bit. This enables the SSU to remember that the critical section has been fully executed by the speculative thread. We call this event *Release While Speculative*. Then, the SSU keeps spinning for ownership because the Acquire bit is still set. Note that the execution of the speculative thread is not disrupted.

In general, when the SSU reads that the lock has been freed externally, before attempting the T&S operation, it checks the Release bit. If the Release bit is still set, the SSU issues the T&S operation to compete for the lock, as described in Section 3.2.2. If, instead, the bit is clear, the SSU knows that the local thread has gone through a Release While Speculative operation and, therefore, has completed all memory operations prior to the release. As a result, the SSU can aggressively *pretend* that ownership is acquired and released instantly. Therefore, the Acquire bit is cleared, all the Speculative bits are flash-cleared, and the SSU becomes idle. In this case, the thread has become safe without ever performing the T&S operation. This is the action taken by threads B and E in the example of Section 2.3 after thread A releases the lock.

As indicated in Section 2.3, this optimization is race-free since: (1) the Release bit in the speculative thread is cleared only after *all* memory operations in the critical section have completed without conflict, and (2) a free lock value indicates that the previous lock owner has completed the critical section as well. If, at the time the speculative thread is about to become safe, an incoming invalidation is in flight from a third processor for a line marked speculative, two things can happen: If the invalidation arrives before the speculative thread has committed, the thread is squashed. This is suboptimal, but correct. Alternatively, if the thread has already committed, the invalidation is serviced conventionally.

3.2.4 Access Conflict

The underlying cache coherence protocol naturally detects access conflicts. Such conflicts manifest in a thread receiving an external invalidation to a cached line, or an external read to a dirty cached line.

If such external messages are received by lines not marked Speculative, they are serviced normally. In particular, messages to the lock owner or to any other safe thread never result in squashes, since none of their cache lines is marked Speculative. Note that the originator thread of such a message could be speculative; in this case, by normally servicing the request, we are effectively supporting *in-order* conflicts from a safe to a speculative thread without squashing.

On the other hand, if a speculative thread receives an external message for a line marked Speculative, the conflict is resolved by squashing the receiving thread. The originator thread may be safe or speculative. If the former, an *out-of-order* conflict has taken place, and thus the squash is warranted.[4] If the latter, we squash the receiving thread, since our proposal does not define an order between speculative threads. In any case, the originator is never squashed.

Once triggered, the squash mechanism proceeds as follows: The SSU flash-invalidates all dirty cache lines with the Speculative bit

[3]Of course, by this time, under both conventional and Speculative Synchronization, the processor may have executed code past the release point.

[4]More sophisticated hardware could disambiguate out-of-order name dependences, and potentially avoid the squash. As indicated in Section 2.2, we choose not to support it for simplicity.

set, flash-clears all Speculative bits and, if the speculative thread had past the release point, it sets the Release bit again. In addition, the SSU forces the processor to restore its checkpointed register state. In this way, the thread quickly rolls back to the acquire point. The flash invalidation is simply a flash clear of the Valid bit, qualified with the Speculative and Dirty bits (NAND gating). Finally, note that we do not invalidate cache lines that have been speculatively read but not modified, since they are coherent with main memory.

If the squash was triggered by an external read to a dirty speculative line in the cache, the node replies without supplying any data. The coherence protocol then regards the state for that cache line as stale, and supplies a clean copy from memory to the requester. This is similar to the case in conventional MESI protocols where a node is queried by the directory for a clean line in state Exclusive that was silently displaced from the cache.

3.2.5 Cache Overflow

Cache lines whose Speculative bit is set cannot be displaced beyond the local cache hierarchy, because they record past speculative accesses. Moreover, if their Dirty bit is also set, their data is unsafe. If a replacement becomes necessary at the outermost level of the local cache hierarchy, the cache controller tries to select a cache line not marked speculative. If no evictable candidate is found, the node stalls until the thread is granted ownership of the lock or it is squashed. Stalling does not jeopardize forward progress, since there always exists a lock owner. The lock owner will eventually release the lock, and the node whose SSU then gains ownership (and any speculative thread that had gone through a Release While Speculative operation on that lock) will be able to handle cache conflicts without stalling. Safe threads do not have lines marked speculative and, therefore, replace cache lines on misses as usual.

3.2.6 Exposed SSU

At times it may be best not to allow threads to speculate beyond a certain point. This can happen, for example, if a certain access is known to be irreversible (e.g. I/O) or to cause conflicts. In this case, the programmer or parallelizing compiler can force the speculative thread to *spin-wait* on the SSU state until it becomes idle (Section 4). Thus, the thread will wait until it either becomes safe or gets squashed. Naturally, if the SSU is already idle, no spinning will take place. We call this action *exposing the SSU* to the local thread. In general, although we envision Speculative Synchronization to be transparent to the programmer and the compiler in practically all cases (Section 4), it is important to provide a mechanism for the software to have this capability.

3.3 Supporting Multiple Locks

Speculative threads may meet a second acquire point. This can happen if there are nested locks or several consecutive critical sections. One approach for these two cases is to expose the SSU to the thread prior to attempting the second acquire. However, a more aggressive approach can avoid unnecessary stalls.

Upon receiving a lock acquire request from the processor (Section 3.2.1), the SSU checks its Acquire bit. If it is clear, the SSU is idle and can service the request as usual. If the SSU is busy, we first consider the more general case where the acquire request is for a lock variable different from the one currently being handled by the SSU. In this case, the SSU rejects the request, no checkpointing is done, and the speculative thread itself handles the second lock using ordinary T&T&S code. No additional support is required.

Handling the second lock using ordinary T&T&S code is correct because, since the thread is speculative, accesses to that lock variable are also considered speculative. Upon the thread reading the value of the lock, the line is marked speculative in the cache. If the lock is busy, the thread spins on it locally. If it is free, the thread takes it and proceeds to the critical section; however, the modification to the lock is contained in the local cache hierarchy, since this is a speculative access. The lock is treated as any other speculative data.

There are two possible final outcomes to this situation. On the one hand, the thread could get squashed. This will occur if there is a conflict with another thread on any cached line marked speculative, including the one that contains the second lock variable itself. In this case, the squash procedure will roll back the thread to the acquire of the first lock (the one handled by the SSU). As usual, all updates to speculative data will be discarded. This includes any speculative update to the second lock variable.

On the other hand, the SSU may complete action on the first lock and render the thread safe. As always, this commits all speculatively accessed data—including the second lock itself. If the thread was originally spinning on this second lock, it will continue to do so safely. Otherwise, any action taken speculatively by the thread on the second lock (acquire and possibly release) will now commit to the rest of the system. This is correct because, if any other thread had tried to manipulate the second lock, it would have triggered a squash.

Finally, there is a special case when the second acquire is to the same lock variable as the one already being handled by the SSU. In this case, the SSU holds the request until the thread becomes safe, or until it completes execution of the first critical section (and the SSU clears the Release bit), whichever is first. If the former, the SSU completes action as usual and then accepts the new acquire request. If the latter (case of Release While Speculative in Section 3.2.3), the SSU simply sets again the Release bit and accepts the acquire request. This way, the SSU effectively merges the two critical sections into a single one. In this case, a second checkpoint is not performed. When the thread eventually commits, it will do so for both critical sections at once. On the other hand, if the thread gets squashed, it will roll back to the first (checkpointed) acquire.

3.4 Speculative Flags and Barriers

To implement speculative flags, we leverage the Release While Speculative support in speculative locks (Section 3.2.3). Recall that, in such a scenario, the SSU of the speculative thread is left with the Release bit clear, and spinning until the lock is set by the owner to the free value. When the lock is set to the free value, the speculative thread becomes safe immediately, without the SSU ever performing T&S (since the Release bit is clear). This mechanism exactly matches the desired behavior of a thread that speculatively executes past an unset flag.

Consequently, on a speculative flag read, the SSU acts exactly as in the case of a speculative lock request, except that the Release bit is kept clear to allow Test but not T&S. The processor is allowed to go past the unset flag speculatively. Naturally, unlike in a speculative lock, in the event of a squash, the Release bit is not set back. We explain later in Section 4 that, as part of a speculative flag request, the thread supplies the "pass" value of the flag to the SSU.

It is possible that a thread speculating past a flag may try to access the same flag again. The SSU handles this situation by simply holding up such an access, until the speculative thread either becomes safe or gets squashed.

Barriers are often implemented using locks and flags as illustrated in Figure 4 [13]. Since the SSU can implement both speculative locks and flags, support for speculative barriers comes for free.

Under conventional synchronization, a thread arriving early to a barrier updates barrier counter *count* and waits spinning on statement *S2*. The counter update is in a critical section protected by lock *c*. Under Speculative Synchronization, it is inadvisable for a

```
local_f = !local_f;
lock(c);
count++; // increment count
if(count==total) { // last one
   count = 0; // reset count
   f = local_f; // toggle (S1)
   unlock(c);
}
else { // not last one
   unlock(c);
   while(f != local_f); // spin (S2)
}
```

Figure 4: Example of bit-reversal barrier code.

thread to enter this critical section while its SSU is busy, since a second thread arriving at the barrier will surely cause conflicts on both the lock and the counter, forcing a rollback of the first thread if it is still speculative—all the way to the synchronization point handled by its SSU. Even if the thread arrives at the barrier in a safe state, the critical section is so small that it is preferable to reserve the SSU for the upcoming flag spin in statement *S2*. Consequently, threads execute this critical section conventionally and speculate on the flag.

To support this behavior, our library code for barriers (Section 4) exposes the SSU to the thread before attempting to acquire *c*, so that speculative threads have a chance to become safe and commit their work. Then, conventional synchronization is used to acquire and release *c*. Finally, when the thread reaches the flag spin (statement *S2*), it issues a speculative flag request and proceeds past the barrier speculatively. Later on, as the last thread arrives and toggles the flag (statement *S1*), all other threads become safe and commit.

3.5 Other Issues

There are a few other related issues that we briefly consider:

Support for Multiprogramming. In a multiprogrammed environment, the operating system may preempt some of the threads of an application. When a speculative thread is preempted, it is squashed and the local SSU is freed up. Any new thread that runs on that processor can use the SSU. When the first thread is finally rescheduled somewhere, it resumes from the synchronization point. On the other hand, safe threads are handled as in a conventional system; in particular, they are never squashed in a context switch. Finally, since Speculative Synchronization is esentially a lock-based technique, it may exhibit convoying under certain scheduling conditions. We address this issue in Section 7.

Exception Handling. When a speculative thread suffers an exception, there is no easy way of knowing whether the cause was legitimate; it could be due to consuming incorrect values speculatively. Consequently, the speculative thread is rolled back in all cases.

False Sharing. Since our implementation uses the memory line as the unit of coherence, false sharing may cause thread squashes. However, our implementation will benefit from the many existing techniques that reduce false sharing. Any such technique that requires per-word state in the caches will also require per-word Speculative bits.

Other Synchronization Primitives. While our discussion has assumed the use of T&T&S, the SSU can be adapted to support other types of synchronization primitives. For example, it could support scalable primitives such as queue-based locks—in this case, each SSU would spin on its own location in the queue, until the content is flipped by the predecessor SSU in the queue. In general, each synchronization primitive may have different operational and performance implications. Further analysis of this issue is the subject of future work.

3.6 Summary

The proposed implementation of Speculative Synchronization has three key characteristics:

1. It supports speculative execution of barriers, flags, and locks in a unified manner.
2. One or more safe threads exist at all times. Safe threads are never squashed due to access conflicts or stalled due to cache overflow. Thus, the performance in the worst case is typically in the same order as conventional synchronization. Furthermore, all in-order conflicts from safe threads to speculative threads are tolerated.
3. It is compatible with conventional synchronization: legacy code that uses conventional synchronization can be run simultaneously with Speculative Synchronization code in the same program.

The implementation also has several additional good aspects. First, the hardware required is simple. Second, under the right conditions (Release While Speculative case), speculative threads can commit the execution of a critical section without ever having to acquire the lock. Third, conflicting accesses are detected on the fly, and offending threads are squashed and eagerly restarted. Fourth, commit and squash operations take approximately constant time, irrespective of the amount of speculative data or the number of processors. Finally, situations involving multiple locks are handled transparently, without unnecessarily stalling, and at no extra cost.

4 SOFTWARE INTERFACE

Explicit high-level synchronization constructs, e.g. M4 macros and OpenMP directives, are widely used by programmers and parallelizing compilers to produce parallel code. These synchronization constructs provide an opportunity to enable Speculative Synchronization transparently. Specifically, we retarget such constructs to encapsulate calls to SSU library procedures. Such library procedures access the SSU via a set of memory-mapped registers. The SSU library comprises three procedures:

ssu_lock(addr) requests a lock acquire operation on variable *addr*. If the SSU accepts the request, the processor performs a register checkpoint (Section 3.2.1) and the SSU initiates the lock acquire. In this case, the procedure returns a nonzero value. If, instead, the SSU rejects the request, typically because the SSU is already busy with another variable, the procedure returns a zero.

ssu_spin(addr,value) requests a spin operation on variable *addr*, where *value* is the "pass" value. If the SSU accepts the request, the processor performs a register checkpoint and the SSU initiates the spin. As before, the procedure returns a nonzero value. Similarly as before, if the SSU rejects the request, the procedure returns a zero.

ssu_idle() returns zero if the SSU is busy, or a nonzero value if it is idle and therefore available.

These library procedures are enough to build macros for speculative locks, flags, and barriers. Consider Table 1, which shows an example of conventional M4 macros on the left side. The right side shows the corresponding speculative M4 macros. The two groups of macros are very similar. The differences are shown in bold face.

The speculative lock (SS_LOCK) and speculative spin (SS_SPIN) macros first try to utilize the SSU, but they revert to the conventional macros (LOCK, SPIN) if the request is rejected. The conventional macro for barriers (BARRIER) uses the typical bit-reversal code. The speculative version (SS_BARRIER), first calls a new macro to expose the SSU (SS_EXPOSE). The SSU is exposed to guarantee safe state before continuing (Section 3.4). Then, the counter is updated using conventional locking. Finally, the spin is attempted using the SSU.

Conventional M4 Macros (Existing)	*Speculative M4 Macros (Proposed)*
`LOCK('{ lock($1);}')`	`SS_LOCK('{ if(!ssu_lock($1)) LOCK($1)}')`
`UNLOCK('{ unlock($1);}')`	`SS_UNLOCK('{ UNLOCK($1)}')`
`SPIN('{ while($1 != $2);}')`	`SS_SPIN('{ if(!ssu_spin($1,$2)) SPIN($1,$2)}')`
	`SS_EXPOSE('{ while(!ssu_idle());}')`
`BARRIER('{ $1.lf[PID] = !$1.lf[PID]; LOCK($1.lock) $1.c++; if($1.c == NUMPROC) { $1.c = 0; $1.f = $1.lf[PID]; UNLOCK($1.lock) } else { UNLOCK($1.lock) SPIN($1.f,$1.lf[PID]) }}')`	`SS_BARRIER('{ $1.lf[PID] = !$1.lf[PID]; SS_EXPOSE LOCK($1.lock) $1.c++; if($1.c == NUMPROC) { $1.c = 0; $1.f = $1.lf[PID]; UNLOCK($1.lock) } else { UNLOCK($1.lock) SS_SPIN($1.f,$1.lf[PID]) }}')`

Table 1: Example M4 macros for conventional synchronization operations and their corresponding speculative ones. The barrier code uses the typical bit-reversal technique. The differences are shown in bold face.

The programmer can enable Speculative Synchronization by simply using these macros instead of the conventional ones. Likewise, parallelizing compilers can be trivially enhanced to generate code with Speculative Synchronization. Indeed, a compilation switch can be used to generate code with speculative macros (typically barriers) rather than with conventional ones.

In summary, Speculative Synchronization has a clean, lean software interface for both programmers and parallelizing compilers. Legacy codes can run because conventional synchronization is still fully functional. In fact, both types of synchronization can coexist at run time in a program.

5 EVALUATION ENVIRONMENT

To evaluate Speculative Synchronization, we use simulations driven by several parallel applications. In this section, we describe the machine architecture modeled and the applications executed.

5.1 Architecture Modeled

We use an execution-driven simulation framework [18] to model in detail a CC-NUMA multiprocessor with 16 or 64 nodes. The system uses the release memory consistency model and a cache coherence protocol along the lines of DASH [21]. Each node has one processor and a two-level hierarchy of write-back caches. The processor is a 4-issue out-of-order superscalar with register renaming, branch prediction, and nonblocking memory operations. The cache sizes are kept small to capture the behavior that real-size input data would exhibit on actual machines, as suggested in [34]; larger cache sizes would generally favor Speculative Synchronization because fewer overflow-induced stalls would occur. Shared data pages are placed round-robin in the memory modules, while private data pages are allocated locally. Table 2 lists the main parameters of the architecture. All traffic and resource contention are modeled in detail except for contention at the network routers, where a constant delay is assumed. We conservatively assess a 15-cycle penalty for a checkpoint of architectural registers.

Processor	1GHz
Issue	4−issue dynamic, 128−entry ROB
ALU	3 integer, 2 floating point
LD/ST	2 units, 16 LD, 16 ST
Branch Prediction	2048−entry 2−bit saturating counter
Branch Penalty	7 cycles
Memory	CC−NUMA, MESI protocol
L1 Cache	1GHz, 16KB, 64B lines, 2−way
L2 Cache	500MHz, 256(64)KB, 64B lines, 8−way
Memory Bus	250MHz, split transaction, 16B width
Main Memory	100MHz SDRAM, interleaved, 60ns row miss
Cache RT: L1, L2	2ns, 12ns
Mem. RT: Local, Neighbor	95ns, 175ns
Network	Hypercube, VCT routing
Router	250MHz, pipelined
Pin−to−Pin Latency	16ns
Endpoint (un)Marshaling	16ns
Configuration	16(64) processors

Table 2: Architecture modeled in the simulations. In the table, RT stands for minimum round trip latency from the processor. The number of processors and the L2 cache size in parentheses correspond to the SPLASH-2 applications.

5.2 Applications Executed

We use five parallel applications from three suites that have different characteristics. They are: one compiler-parallelized SPECfp95 application (APPLU); two Olden [3] codes annotated for parallelization (Bisort and MST); and two hand-parallelized SPLASH-2 [34] applications (Ocean and Barnes). These applications synchronize using locks and barriers. In particular, APPLU and Bisort are barrier-only codes, while all others use both locks and barriers. Table 3 summarizes the characteristics of these applications.

The parallel APPLU code was generated by Polaris, a state-of-the-art parallelizing compiler [2]. The Olden codes are pointer-based applications that operate on graphs and trees. They are annotated so that the compiler or the programmer can easily parallelize them. We follow these annotations faithfully. The SPLASH-2 applications are fine-tuned, hand-parallelized codes. Barnes uses hashing to synchronize over a limited number of locks. In our experiments, we use two configurations of Barnes with a different number of hashed locks: one with 512 locks (Barnes-Coarse) and one with 2048 locks (Barnes-Fine). The code in the original suite is Barnes-Fine.

We execute the SPLASH-2 applications on 64 processors because these applications scale particularly well; the rest of the applications are executed on 16 processors because they are less scalable. Under conventional synchronization, the average efficiency of the parallel execution of these applications for the chosen numbers of processors is 49%. In all cases, we warm up the cache hierarchy before starting to collect execution statistics. Finally, we simulate all the applications to completion except for APPLU, where we reduce the number of iterations because they all exhibit a similar behavior.

6 EVALUATION

In this section, we evaluate the applications under conventional synchronization and Speculative Synchronization (*Base* and *Spec*, respectively). We first assess the overall effectiveness of Speculative Synchronization, and then analyze the factors that contribute to it.

6.1 Overall Effectiveness

Figure 5 shows the execution time of the applications on the *Base* and *Spec* systems. The bars are normalized to *Base* and broken down into five categories. *Useful* is the time spent in computation that is ultimately profitable. It includes processor busy time, as well as stall

Application	*Description*	*Parallelization*	*Data Size*	*Processors*	*Barriers/Locks*
APPLU	LU factorization	Compiler	Reference	16	Yes/No
Bisort	Bitonic sort	Annotations	16K nodes	16	Yes/No
MST	Minimum spanning tree	Annotations	512 nodes	16	Yes/Yes
Ocean	Ocean simulation	Hand	258x258	64	Yes/Yes
Barnes−Fine	N−body problem	Hand	16K particles	64	Yes/Yes(2048)
Barnes−Coarse					Yes/Yes(512)

Table 3: Applications used in the experiments.

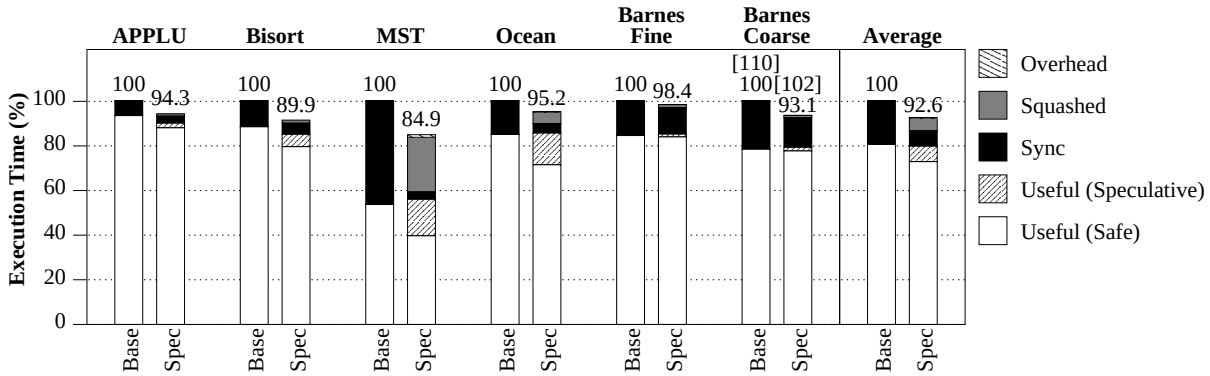

Figure 5: Execution time of the applications under conventional (*Base*) and speculative (*Spec*) synchronization. Bars are normalized to *Base*. Ocean and Barnes run with 64 processors, while the other applications run with 16 processors. Barnes-Coarse also shows in brackets the execution times normalized to Barnes-Fine's *Base*.

due to memory and pipeline hazards. It is subdivided into *Useful Safe* and *Useful Speculative*, depending on whether execution was safe or speculative, respectively. Of course, *Useful Speculative* only appears in *Spec*. *Sync* is the time spent spinning at synchronization points. *Squashed* is the time wasted by speculative threads on execution that gets ultimately squashed. Finally, *Overhead* is the time taken to handle squash operations, including draining the processor pipeline and load/store buffers, and restoring the processor's checkpointed register state.

Under *Base*, applications spend on average 19.4% of their time spinning at synchronization points. Naturally, the impact of Speculative Synchronization is largely bounded by this figure. In practice, Speculative Synchronization reduces the original synchronization time by an average of 34%, if we combine the residual synchronization time (*Sync*), the squashed execution time (*Squashed*), and the squash overhead (*Overhead*) of *Spec*. As a result, the average execution time of the applications in Figure 5 decreases by 7.4%. Across applications, the reduction in execution time ranges from a small value in Barnes-Fine to a significant 15% in MST.

The fraction of the code executed speculatively is the sum of *Useful Speculative* plus *Squashed*. The combined size of these two categories is necessarily small in applications with little synchronization, such as APPLU. On the other hand, frequent synchronization provides an opportunity to speculate more; this is the case for MST and, to a lesser extent, Ocean, where *Useful Speculative* plus *Squashed* account for about 50% and 22% of the total *Spec* time, respectively. On average for all applications, about half of this time proves to be useful (*Useful Speculative*).

Ideally, the total useful time in an application should remain constant as we go from *Base* (*Useful Safe* only) to *Spec* (*Useful Safe* plus *Useful Speculative*). In practice, however, we see that the total useful time changes slightly. The reason is constructive or destructive memory effects by speculative computation. On the one hand, speculative execution that gets ultimately squashed can have the positive effect of prefetching useful clean data into the caches. This is the effect observed in APPLU and Bisort. On the other hand, squashes involve invalidating cache lines modified speculatively. The result may be the negative effect of destroying locality that was originally present in the caches. This is the effect observed in MST. In general, it is hard to predict whether the effect will be constructive or destructive.

Focusing on the non-useful execution, we see that it is largely composed of residual synchronization and squashed execution time; the overhead of the squash mechanism (*Overhead*) accounts for little. Residual sychronization is due to a *speculative* thread bumping into a barrier (case of exposed SSU in Section 3.4), or into a lock that is busy (case of multiple locks, Section 3.3). Both residual synchronization and squashed execution time represent areas for further improvement of Speculative Synchronization. Residual synchronization is relatively large in both Barnes-Fine and Barnes-Coarse; squashed execution time is sizable in Ocean and MST. These two categories are discussed in Section 6.2 in detail.

Finally, the numbers in brackets on top of the Barnes-Coarse bars show the execution time of this application normalized to Barnes-Fine's *Base*. We can see that Barnes-Coarse *Base* takes 10% longer to execute than Barnes-Fine *Base*. This is largely due to the coarser synchronization. However, if we apply Speculative Synchronization (Barnes-Coarse *Spec*), its execution time comes down to only 2% longer than Barnes-Fine *Base*. This shows that Speculative Synchronization can indeed compensate for conservative synchronization.

6.2 Contributing Factors

We now focus on the time lost to synchronization and related overheads. We compare the synchronization time in *Base* with the time lost to residual synchronization, squashed computation, and squash overhead in *Spec*. The results are shown in Figure 6. We break down synchronization time into barrier (*Barrier Sync*) and lock (*Lock Sync*) time—there are no speculative flags *per se* in these codes. We break down the time lost to squashed computation into three categories, depending on the reason for the squash: *True Data*, *False*

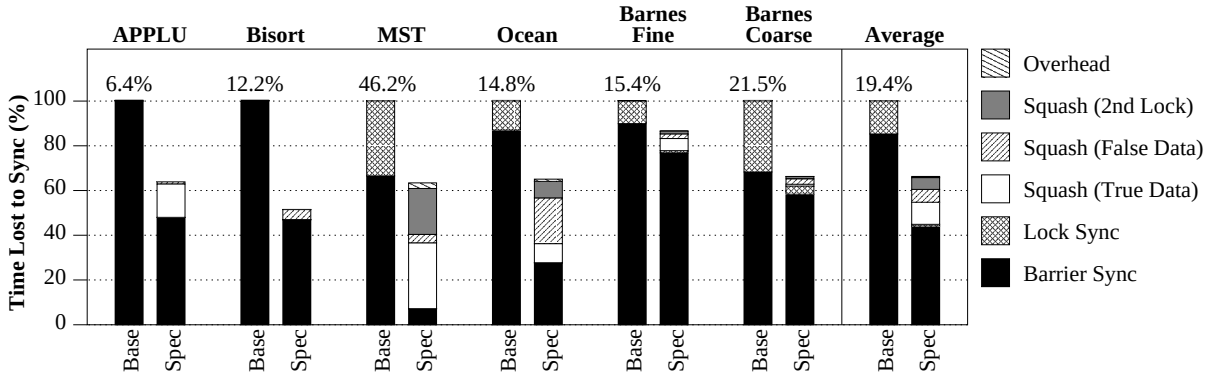

Figure 6: Factors contributing to synchronization time and related overheads for the *Base* and *Spec* systems. The results are normalized to *Base*. Ocean and Barnes run with 64 processors, while the other applications run with 16 processors. The percentages at the top of the bars reflect the fraction of synchronization in *Base*.

Data, and *2nd Lock*. *True Data* and *False Data* are computation squashed due to conflicts induced by same-word accesses and false data sharing, respectively. Recall that the Speculative bits are kept on a per-line basis and, therefore, false sharing causes conflicts. *2nd Lock* is computation squashed due to a speculative thread conflicting in a second synchronization variable. Such a variable is necessarily a lock, since the SSU is exposed on barriers (Section 3.4).

Figure 6 shows that, in general, the synchronization time of these applications in *Base* is dominated by barriers—in fact, APPLU and Bisort synchronize exclusively through barriers. Nevertheless, MST and Barnes-Coarse exhibit significant lock activity. This makes it important to attack synchronization due to both locks and barriers.

As shown in Figure 6, Speculative Synchronization significantly reduces the contribution of both lock and barriers. Still, the *Spec* bars show some residual synchronization time. This is caused by speculative threads stalling at a second synchronization point. The large majority of this residual synchronization time is spent in barriers, which can never be crossed by threads that are already speculative (exposed SSU). In Bisort, for example, residual barrier time appears at the top part of a tree. As processors move toward the root, the number of processors with work to do decreases. Idle processors start synchronizing at every level, with nothing to do but to wait for others. As they cross one barrier speculatively, they immediately bump into the next one, where they stall. In Barnes, speculative threads also stall at barriers. However, contention on the locks is significantly reduced. Often times, by the time a lock is released by its owner, several speculative threads have already completed their critical sections concurrently (Release While Speculative, Section 3.2.3). At that point, they all commit without competing for the lock. This reduces *Lock Sync* time (Figure 6).

Focusing on the squash time, we see that only Ocean and MST exhibit a relatively large fraction of squashed work. In Ocean, the main source of squashes is false sharing (*False Data*). In MST, the main contributors to the squash time are same-word access conflicts (*True Data*) and access conflicts on second-lock variables (*2nd Lock*), in that order. It is important to devise techniques to minimize all these sources of squashes. We address this issue next.

6.3 Eliminating Remaining Overheads

The main four overheads that remain in the *Spec* bars of Figure 6 can be attacked with changes to the SSU and the system. We examine each case in turn.

False Sharing. In general, techniques that reduce false sharing in shared memory multiprocessors also benefit Speculative Synchronization. Some of these techniques would require keeping per-word Speculative bits. We have to be careful, however, not to hurt performance in other ways. For example, while data padding in Ocean may reduce false sharing, it may also give up spatial locality present in the application.

True Sharing. A more sophisticated Speculative Synchronization protocol can reduce the cases where same-word conflicts cause squashes. For example, out-of-order WAR and WAW conflicts need not cause squashes if the system supports multiple versions of a variable across processors. Many systems for TLS incorporate forms of multiple version support. The cost is more hardware support and a more complicated protocol.

2nd-Lock Squash. To avoid second-lock squashes, we can expose the SSU before each lock as in barriers, so that a speculative thread waits to become safe. Then, it can use its SSU to enter the second lock speculatively. This avoids second-lock squashes at the expense of disallowing a processor to speculate simultaneously in multiple critical sections. For our particular applications, this approach causes most of the *2nd Lock* time to to simply mutate into residual synchronization. In fact, the overall execution time is slightly higher.

Residual Synchronization. To minimize residual synchronization, we can design a more sophisticated SSU that handles multiple speculative epochs with multiple sets of Speculative bits. Incidentally, this support can also solve the problem of *2nd Lock*. This multi-epoch support, which complicates the SSU in a nontrivial way, resembles that of TLS.

Interestingly, by adding all these enhancements to Speculative Synchronization, namely support for multiple epochs, multiple data versions, and per-word speculative state, we would obtain a system that comes close to current proposals for TLS. Understanding the full interaction between Speculative Synchronization and TLS is the subject of our current research.

7 ADAPTIVE SPECULATIVE SYNCHRONIZATION

There are a few proposals for hardware lock-free optimistic synchronization in critical sections. These schemes have some good features that complement those of Speculative Synchronization. In this section, we describe two such schemes that have similar hardware simplicity as ours; Section 8 describes other related work. Then, we outline an adaptive scheme that extends Speculative Synchronization to capture the positive aspects of lock-free synchronization, while preserving all the advantages of our original solution. We call the scheme *Adaptive Speculative Synchronization*.

Characteristic	*TM*	*SLE*	*Speculative Synchronization*	
			Basic	*Adaptive*
Applicability	Critical sections	Locks	Locks, flags, barriers	
Commit without lock acquire	Yes	Yes (if successful)	Release While Speculative (RWS)	Yes (if no conflicts/overflow or if RWS)
Convoying	No	No (if successful)	Possible	No (if no conflicts/overflow)
Safe thread	No		Yes	As needed
Action on overflow	Not handled	Grab lock, squash all contenders	Safe thread: not affected Speculative thread: compete until acquire; then continue	
Action on conflict inside critical section	Squash	Squash receiver	Receiver is safe: continue Receiver is speculative: squash receiver	
Squashes in critical path	Possible		No	
Programming effort	Yes	No	No	
On−the−fly rollback	No	Yes	Yes	

Table 4: Comparing the speculative mechanisms of TM and SLE to Speculative Synchronization. *Receiver* denotes a thread that receives a coherence message due to an access conflict inside the critical section.

7.1 Lock-Free Optimistic Synchronization

7.1.1 Transactional Memory

Herlihy and Moss's Transactional Memory (TM for short) [16] proposes lock-free optimistic synchronization in critical sections via special transactional instructions and support in the cache hierarchy to hold speculative data. In general, TM requires that the code be written in a lock-free manner [15]. All threads execute critical sections speculatively, and the coherence protocol helps detect conflicts. The code can check whether conflicts have been flagged for a thread. If so, the thread discards all changes and jumps back to the beginning of the critical section. If at the time a thread completes the critical secion no conflicts have been flagged, the thread can commit.

7.1.2 Speculative Lock Elision

Concurrently to our work [25], Rajwar and Goodman propose Speculative Lock Elision (SLE) [27]. SLE dynamically converts lock-based into lock-free codes, by removing acquire and release operations in the instruction stream from the processor pipeline. As in TM, all threads execute critical sections speculatively. SLE also leverages the coherence protocol to detect conflicts. Compared to TM, SLE presents some important advantages: SLE requires no programming effort, since codes are lock-based and the elision mechanism is transparent; SLE can fall back to conventional synchronization if needed (see discussion below); lastly, SLE features on-the-fly thread rollback and restart (as in Speculative Synchronization).

7.1.3 Discussion

We limit our discussion to speculation in critical sections, since neither TM nor SLE supports speculation on barriers or flags. As lock-free proposals, TM and SLE allow, under the right conditions, to execute critical sections without the need to secure a lock. In Speculative Synchronization, this behavior is possible after a Release While Speculative operation (Section 3.2.3). Still, Speculative Synchronization requires one thread to grab and own the lock in every active critical section. As a result, speculative threads must wait for the lock to be freed by the owner before they can commit. This makes Speculative Synchronization subject to convoying. In conventional locks, convoying occurs when a lock owner is preempted by the scheduler, and other threads are left spinning for the lock. In Speculative Synchronization, preempting a lock owner prevents speculative threads from committing. A number of techniques have been proposed to avoid preempting a lock owner [6, 17, 24]. Nevertheless, in general, convoying is a concern.

On the other hand, TM and SLE share a shortcoming—in the presence of conflicts, their speculative mechanisms do not embed a forward progress guarantee. Indeed, since *all* threads execute the critical section speculatively, *any* such thread that receives a coherence message due to an access conflict inside the critical section is squashed. As a result, repetitive conflicts may cause threads to livelock unless special action is taken. Specifically, TM relies on software-level adaptive backoff to increase the probability of eventual success. In SLE, after a certain number of failed retries, one of the speculative threads abandons lock-free mode and explicitly acquires the lock. Unfortunately, SLE and lock-based synchronization are mutually exclusive; therefore, once a thread explicitly grabs the lock, all other threads in that critical section get squashed and start spinning on the (now busy) lock.

Another problem occurs if speculative threads overflow their speculative buffers. In SLE, a speculative thread that is about to overflow its speculative buffer can abandon lock-free mode and explicitly acquire the lock. As before, all other threads in that critical section get squashed and start spinning on the (now busy) lock. TM does not provide a solution for the problem of overflow.

Meanwhile, in the presence of conflicts or overflow, the existence of a lock owner at all times gives Speculative Synchronization two advantages. First, the lock owner can receive coherence messages due to access conflicts inside the critical section without getting squashed; in the meantime, speculative threads can all execute inside the critical section without being concerned about forward progress, which is guaranteed by the lock owner. The second advantage is that any number of speculative threads that are about to overflow their caches can stall, compete for the lock (which is held by the owner) and, upon acquiring it, continue; no squashing of any thread is involved.

The first four columns of Table 4 summarize this discussion. In the table, *Receiver* denotes a thread that receives a coherence message due to an access conflict inside the critical section.

7.2 Proposed Adaptive Extension

We now extend the SSU to also implement lock-free synchronization in critical sections, and thus show that an adaptive protocol that combines the best of both worlds is possible. The basic idea behind Adaptive Speculative Synchronization is to operate in a lock-free manner, but fall back to Speculative Synchronization by producing a lock owner if forward progress is compromised. Speculation is never disabled. Barriers and flags are still handled speculatively as in our base mechanism.

Upon a speculative lock request, the SSU reads in the lock variable, but it does not try to secure ownership. Therefore, all threads that access the critical section do so speculatively. This is similar to SLE, except that a thread can venture into the critical section speculatively *regardless of whether the lock is busy or free*. As the speculative thread completes execution of the critical section, the SSU tests the value of the lock. If free, the thread commits, lock-free style. If not, the SSU falls into Release While Speculative mode

(Section 3.2.3), and commits only when the lock is freed by the owner. At no time is an acquire operation attempted on the lock variable.

If a speculative thread detects a conflict during the execution, or if the speculative thread is about to overflow its cache, its SSU proceeds to compete for the lock. There are two possible outcomes to this situation. If the SSU secures ownership, the thread becomes safe, guaranteeing forward progress. If, instead, another thread owns the lock, forward progress is guaranteed by that other thread; in this case, the speculative thread rolls back (case of conflict, Section 3.2.4) or stalls while its SSU keeps competing for lock ownership (case of cache overflow, Section 3.2.5).

Therefore, in the absence of conflicts or cache overflow, the SSU implements lock-free Speculative Synchronization in critical sections. If conflicts or cache overflow do occur, SSUs smoothly fall back to the original lock-based Speculative Synchronization. Notice that, even when the lock is grabbed by a thread, the other threads (which are speculative) are allowed to continue. Overall, the SSU operates in lock-free mode except as needed, while preserving the advantages of Speculative Synchronization. The last column of Table 4 summarizes Adaptive Speculative Synchronization.

8 RELATED WORK

Optimistic Concurrency Control (OCC) [20] sets the foundation for optimistic synchronization, based on the notion of "apologizing versus asking permission" [14]. Transactions execute without synchronizing, after which they undergo a validation phase, and then commit (if atomicity is preserved), or abort and restart.

Herlihy uses optimistic synchronization to construct lock-free and wait-free data objects [15]. While the technique may work well for small, simple structures, it is unclear how to deal efficiently with larger, complex objects with high copy overhead. Rinard uses fine-grain optimistic synchronization in compiler-driven parallelization [29]. Still, conventional synchronization is necessary under multiple, interdependent updates to different objects. In general, optimistic synchronization requires nontrivial programming effort.

Several hardware proposals exist for lock-free optimistic synchronization. Two important proposals that relate closely to ours are Herlihy and Moss's Transactional Memory [16] and Rajwar and Goodman's Speculative Lock Elision (SLE) [27]. We address these extensively in Section 7.

Stone et al. [32] propose a hardware optimistic synchronization mechanism called Oklahoma Update. Speculative state is limited to specialized reservation registers within the processor. Requests for exclusive access to speculative data are deferred to the commit phase, called Oklahoma Update, making such an operation potentially slow and traffic-intensive. True conflicts at this phase are resolved by buffering external requests (e.g. invalidations), and selectively delaying responses. Progress is guaranteed only if enough such buffering is provided. Conflicts due to false sharing need a backoff mechanism to guarantee forward progress. None of these problems affects our proposal.

Building on SLE [27], Rajwar and Goodman recently propose Transactional Lock Removal [28], to preserve lock-free behavior even in the presence of conflicts. They use timestamps to dynamically order threads, buffering external requests and selectively delaying responses on the fly based on this order. Special messages are used to avoid deadlocks. Enough buffering resources must be provided to handle conflicts. Cache overflows are handled by disabling the mechanism and falling back to conventional lock-based synchronization.

Sato et al. [30] address speculation across barriers. They discuss how to modify caches and the coherence protocol to support speculation. However, they do not propose any concrete implementation of the speculative barrier itself. Their evaluation assumes a conservative consistency model, in-order processors, and a constant-delay model of processor and memory operations.

Gupta's Fuzzy Barrier [11] attacks barrier imbalance by decoupling barriers into two phases, moving between them nonconflicting code originally after the barrier. This approach requires dependence information at compile time.

Gharachorloo et al. [8] propose allowing loads to execute speculatively ahead of incomplete stores that precede them in program order. They do not allow reordering of store operations, utilizing hardware exclusive prefetches instead. Speculation is limited by the maximum number of uncommitted instructions, and by the processor's buffering capacity. The behavior of the branch predictor in an acquire loop may adversely affect the effectiveness of the scheme.

Pai et al. [26] first propose a synchronization buffer to offload an acquire loop, to improve the behavior of the branch predictor in [8]. They also propose Fuzzy and Speculative Acquires to achieve fine-grain synchronization, which require compiler support to identify the (non)conflicting accesses in critical sections.

Gniady et al. [9] propose SC++, an aggressive implementation of SC that allows reordering of load and store operations by maintaining an in-order history queue of the speculatively retired instructions. Consistency violations trigger a recovery procedure that uses the history queue to reconstruct the state at the instruction at fault. The cost of this recovery grows with the amount of speculative work, and may result in slowdowns. SC++ is shown to match the performance of RC for "well-behaved applications."

Rather than attacking reordering in general, we choose to specialize on speculative execution of widely used synchronization primitives, namely barriers, locks, and flags. This allows our hardware to remain simple, yet effective. Our checkpoint/recovery mechanism, combined with the cache support, allows us to retire a large number of speculative instructions, and quickly roll back in the event of a misprediction—independently of the amount of speculative work. Furthermore, under the right conditions, speculative threads can commit a critical section without ever acquiring the associated lock.

Gharachorloo and Gibbons [7] propose hardware that leverages the coherence protocol to detect violations of sequential consistency. Adve and Hill [1] use explicit synchronization to order contending threads in a critical section, but leverage the coherence protocol to achieve fine-grain synchronization of memory accesses. They employ reserve buffers to selectively defer coherence messages on conflicting addresses.

9 CONCLUSIONS

We have presented *Speculative Synchronization*, which applies the philosophy behind Thread-Level Speculation to explicitly parallel applications. Threads speculatively execute past active barriers, busy locks, and unset flags instead of waiting. The hardware checks for conflicting accesses and, if a violation is detected, the offending speculative thread is rolled back to the synchronization point and restarted on the fly. In any speculative barrier, lock, or flag, the existence of one or more safe threads at all times guarantees forward progress, even in the presence of conflicts or speculative buffer overflow. All in-order conflicts from safe to speculative threads are tolerated without causing squashes. Speculative Synchronization requires simple register checkpointing and cache hardware, can be made transparent to programmers and parallelizing compilers, and can coexist with conventional synchronization at run time.

For critical sections, we have extended our scheme into Adaptive Speculative Synchronization, which captures the positive aspects of lock-free synchronization. Threads operate in a lock-free manner, but the system falls back to Speculative Synchronization by producing a lock owner if forward progress is compromised.

We have evaluated 5 compiler- and hand-parallelized applications under Speculative Synchronization. The results are promising: the time lost to synchronization is reduced by 34% on average, while the overall execution time of the applications is reduced by 7.4% on average. We have also identified ways of further improving Speculative Synchronization.

We are currently extending this work in several directions. Specifically, we are analyzing how the SSU best supports other types of synchronization primitives. We are also evaluating Adaptive Speculative Synchronization. Finally, we are analyzing the full interaction between Speculative Synchronization and TLS.

ACKNOWLEDGMENTS

The authors would like to thank Sarita Adve, Marcelo Cintra, María Jesús Garzarán, Jim Goodman, Milos Prvulovic, and the anonymous reviewers for useful feedback.

REFERENCES

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Advanced program restructuring for high-performance computers with Polaris. *IEEE Computer*, 29(12):78–82, Dec. 1996.

[3] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. In *Symposium on Principles and Practice of Parallel Programming*, pages 29–38, Santa Barbara, CA, July 1995.

[4] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *International Symposium on Computer Architecture*, pages 13–24, Vancouver, Canada, June 2000.

[5] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, Jan.–Mar. 1998.

[6] J. Edler, J. Lipkis, and E. Schonberg. Process management for highly parallel UNIX systems. In *USENIX Workshop on Unix and Supercomputers*, San Francisco, CA, Sept. 1988.

[7] K. Gharachorloo and P. B. Gibbons. Detecting violations of sequential consistency. In *Symposium on Parallel Algorithms and Architectures*, pages 316–326, Hilton Head, SC, July 1991.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *International Conference on Parallel Processing*, pages I355–I364, St. Charles, IL, Aug. 1991.

[9] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC+ILP=RC? In *International Symposium on Computer Architecture*, pages 162–171, Atlanta, GA, May 1999.

[10] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative versioning cache. In *International Symposium on High-Performance Computer Architecture*, pages 195–205, Las Vegas, NV, Jan.–Feb. 1998.

[11] R. Gupta. The Fuzzy Barrier: A mechanism for high-speed synchronization of processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 54–63, Boston, MA, Apr. 1989.

[12] L. Hammond, M. Wiley, and K. Olukotun. Data speculation support for a chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, San Jose, CA, Oct. 1998.

[13] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, second edition, 1996.

[14] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, Mar. 1990.

[15] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Parallel Languages and Systems*, 15(5):745–770, Nov. 1993.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993.

[17] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-conscious synchronization. *ACM Transactions on Computer Systems*, 15(1):3–40, Feb. 1997.

[18] V. Krishnan and J. Torrellas. A direct-execution framework for fast and accurate simulation of superscalar processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, Paris, France, Oct. 1998.

[19] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, Sept. 1999.

[20] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[21] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, May 1990.

[22] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, Inc., New York, NY, 1996.

[23] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *International Conference on Supercomputing*, pages 365–372, Rhodes, Greece, June 1999.

[24] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class user-level threads. In *Symposium on Operating System Principles*, pages 110–121, Pacific Grove, CA, Oct. 1991.

[25] J. F. Martínez and J. Torrellas. Speculative Locks for concurrent execution of critical sections in shared-memory multiprocessors. In *Workshop on Memory Performance Issues*, Gothenburg, Sweden, June 2001.

[26] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Cambridge, MA, Oct. 1996.

[27] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling highly concurrent multithreaded execution. In *International Symposium on Microarchitecture*, pages 294–305, Austin, TX, Dec. 2001.

[28] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based codes. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[29] M. C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization. *ACM Transactions on Computer Systems*, 17(4):337–371, Nov. 1999.

[30] T. Sato, K. Ohno, and H. Nakashima. A mechanism for speculative memory accesses following synchronizing operations. In *International Parallel and Distributed Processing Symposium*, pages 145–154, Cancun, Mexico, May 2000.

[31] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *International Symposium on Computer Architecture*, pages 1–12, Vancouver, Canada, June 2000.

[32] J. M. Stone, H. S. Stone, P. Heidelberg, and J. Turek. Multiple reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993.

[33] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual*. PTR Prentice Hall, 1994.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[35] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 6(2):28–40, Apr. 1996.