SPECULATIVE SYNCHRONIZATION: PROGRAMMABILITY AND PERFORMANCE FOR PARALLEL CODES

PROPER SYNCHRONIZATION IS VITAL TO ENSURING THAT PARALLEL APPLICATIONS EXECUTE CORRECTLY. A COMMON PRACTICE IS TO PLACE SYNCHRONIZATION CONSERVATIVELY SO AS TO PRODUCE SIMPLER CODE IN LESS TIME. UNFORTUNATELY, THIS PRACTICE FREQUENTLY RESULTS IN SUBOPTIMAL PERFORMANCE BECAUSE IT STALLS THREADS UNNECESSARILY. SPECULATIVE SYNCHRONIZATION OVERCOMES THIS PROBLEM BY ALLOWING THREADS TO SPECULATIVELY EXECUTE PAST ACTIVE BARRIERS, BUSY LOCKS, AND UNSET FLAGS. THE RESULT IS HIGH PERFORMANCE.

José F. Martínez
Cornell University

Josep Torrellas
University of Illinois at Urbana-Champaign

•••••• Parallel applications require carefully synchronized threads for execute correctly. To achieve this, programmers and compilers typically resort to widely used synchronization operations such as barriers, locks, and flags.[1] Often, however, the placement of synchronization operations is conservative: Sometimes, the programmer or the compiler cannot determine whether code sections will be race-free at runtime. Other times, disambiguation is possible, but the required effort is too costly. In either case, conservative synchronization degrades performance because it stalls threads unnecessarily.

Recent research in thread-level speculation (TLS) describes a mechanism for optimistically executing unanalyzable serial code in parallel.[2] TLS extracts threads from a serial program and submits them for speculative execution in parallel with a safe thread. The goal is to extract parallelism from the code. Under TLS, special hardware checks for cross-thread dependence violations at runtime, and forces offending speculative threads to squash and restart on the fly. At all times, at least one safe thread exists. While speculative threads venture into unsafe program sections, the safe thread executes code without speculation. Consequently, even if all the speculative work is useless, the safe thread still guarantees that execution moves forward.

Speculative synchronization applies the philosophy behind TLS to explicitly parallel

applications. Application threads execute speculatively past active barriers, busy locks, and unset flags instead of waiting. A speculative thread uses its processor's caches to buffer speculatively accessed data, which cannot be displaced to main memory until the thread becomes safe. The hardware looks for conflicting accesses—accesses from two threads to the same location that include at least one write and are not explicitly synchronized. If two conflicting accesses cause a dependency violation, the hardware rolls the offending speculative thread back to the synchronization point and restarts it on the fly.

Key to speculative synchronization is TLS's principle of always keeping one or more safe threads. In any speculative barrier, lock, or flag, safe threads guarantee forward progress at all times—even in the presence of access conflicts and insufficient cache space for speculative data. Always keeping a safe thread and providing unified support for speculative locks, barriers, and flags are two characteristics that set speculative synchronization apart from lock-free optimistic synchronization schemes with similar hardware simplicity.[3,4]

The complexity of the speculative synchronization hardware is modest—one bit per cache line and some simple logic in the caches, plus support for checkpointing the architectural registers. Moreover, by retargeting high-level synchronization constructs (M4 macros[5] or OpenMP directives,[6] for example) to use this hardware, speculative synchronization becomes transparent to application programmers and parallelizing compilers. Finally, conventional synchronization is compatible with speculative synchronization. In fact, the two can coexist at runtime, even for the same synchronization variables.

We evaluated speculative synchronization for a set of five compiler- and hand-parallelized applications[7] and found that it reduced the time lost to synchronization by 34 percent on average.

Concept

TLS extracts threads from a serial program and submits them for speculative execution in parallel with a safe thread. The goal is to extract parallelism from the code. In TLS, the hardware is aware of the (total) order in which the extracted threads would run in the original serial code. Consequently, the hardware assigns each thread an epoch number, with the lowest number going to the safe thread.

Speculative synchronization deals with explicitly parallel application threads that compete to access a synchronized region. Speculative threads execute past active barriers, busy locks, and unset flags. Under conventional synchronization, these threads would be waiting instead. Contrary to conventional TLS, speculative synchronization does not support ordering among speculative threads; thus, the hardware simply assigns one epoch number for all speculative threads. Every active lock, flag, and barrier has one or more safe threads, however: In a lock, the lock owner is safe. In a flag, the producer is safe. In a barrier, the lagging threads are safe. Because safe threads cannot get squashed or stall, the application always moves forward.

Both safe and speculative threads concurrently execute a synchronized region, and the hardware checks for cross-thread dependency violations. As in TLS, as long as the threads do not violate dependencies, the hardware lets them proceed concurrently. Conflicting accesses (one of which must necessarily be a write) between safe and speculative threads cause no violations if they happen *in order*—that is, the access from the safe thread happens before the access from the speculative one. Any *out-of-order* conflict between a safe thread and a speculative one causes the squash of the speculative thread, and the hardware rolls it back to the synchronization point. Speculative threads are unordered; therefore, if two speculative threads issue conflicting accesses, the hardware always squashes one of them and rolls it back to the synchronization point. However, because safe threads make progress whether or not speculative threads succeed, performance in the worst case is still on the order of conventional synchronization's performance.

A speculative thread keeps its (speculative) memory state in a cache until it becomes safe. At that time, it *commits* (makes visible) its memory state to the system. If a speculative thread's cache is about to overflow, the thread stalls and waits to become safe. The circumstances under which a speculative thread becomes safe are different for locks, flags, and barriers, as we explain next.

Speculative locks
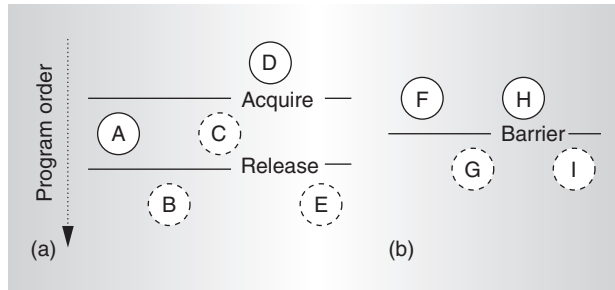
Every contended speculative lock features

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

one safe thread—the lock owner. All other contenders venture into the critical section speculatively. Figure 1a shows an example of a speculative lock with five threads. Thread *A* found the lock free and acquired it, becoming the owner and therefore remaining 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.

Supporting a lock owner has several implications. The final outcome must 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, these speculative threads would wait at the acquire point.) On the one hand, this implies that it is correct for speculative threads to consume values that the lock owner produces. On the other hand, speculative threads cannot commit while the owner is in the critical section. In Figure 1a, threads *B* and *E* have completed the critical section and are speculatively executing code past the release point, and thread *C* is still in the critical section. (The hardware remembers that threads *B* and *E* have completed the critical section, as we describe later.) All three threads remain speculative as long as thread *A* owns the lock.

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—without even acquiring the lock. This environment is race-free because these threads 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 the example) compete for lock ownership. One of them acquires the lock (inevitably *C* in the example), becomes safe, and commits its speculative memory state. The losers remain speculative.

The post-release action is semantically equivalent to the following scenario under a conventional lock: After the owner releases the lock, all the speculative threads beyond 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 1a, for example, this scenario corresponds to a conventional lock whose critical section is traversed in (*A,B,E,C*) or (*A,E,B,C*) order.

Speculative flags and barriers

Flags are synchronization variables that one thread produces and that multiple threads consume, making them one-to-many operations. 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 lets them proceed speculatively. Such threads remain speculative until the "pass" flag value is produced, at which point they all become safe and commit their state. The producer thread, which remains safe throughout, guarantees forward progress.

Figure 1b shows an example of a speculative barrier, which is a many-to-many operation. Conceptually, a barrier is similar to a flag in which the producer is the last thread to arrive.[1] Under speculative synchronization, threads arriving at a barrier become speculative and continue (threads *G* and *I* in Figure 1b). 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.

Implementation

Figure 2 shows the hardware support for speculative synchronization. The main supporting module is the speculative synchronization unit (SSU), which 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. Its function is to offload from the processor the operations on a single 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. Both local and remote requests can access this extra cache line, but only the SSU can allocate it. The local cache hierarchy (L1 + L2 in Figure 2) serves as the buffer for speculative data. To distinguish data accessed speculatively, the SSU keeps one Speculative bit (S) per line in the local cache hierarchy. The SSU sets the Speculative bit of a line when the processor reads or writes the line speculatively. Lines whose Speculative bit is set cannot be displaced beyond the local cache hierarchy.

The SSU also has two state bits, Acquire (A) and Release (R). The SSU sets the Acquire bit if it has a pending acquire operation on the synchronization variable. Likewise, it sets the Release bit if it has a pending release operation on that variable. The SSU can set Speculative, Acquire, or Release bits only when it is active, which is when it is handling a synchronization variable. When the SSU is idle, all these bits remain at zero.

Supporting speculative locks

In an earlier article,[8] we described how our hardware supports speculative locks. We assume the use of the test&test&set (T&T&S) primitive to implement a lock-acquire operation, although other primitives are possible. Figure 3a shows a T&T&S operation. In a conventional lock, the competing threads spin locally on a cached copy of the lock variable (**S1** and **S2**) until the owner releases the lock (a zero value means that the lock is free). The competing threads then attempt an atomic test&set (T&S) operation (**S3**). Register R1 recalls the result of the test; a zero value means success. Only one thread succeeds, becoming the new owner; the rest go back to
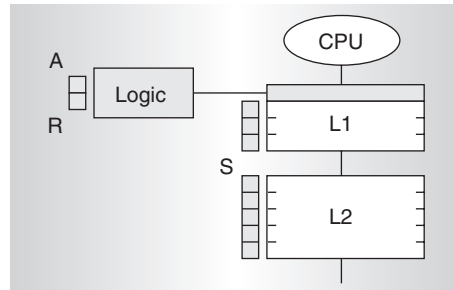

Figure 2. Hardware support for speculative synchronization. The shaded areas are the speculative synchronization unit (SSU), which resides in the local cache hierarchy, typically L1 + L2. The SSU consists of one Speculative bit (S) per cache line, one Acquire bit (A), and one Release bit (R), plus one extra cache line and some logic.

the spin loop (**S4**). In speculative synchronization, the lock-acquire operation is quite different, as the following subsections describe.

Lock request. When a processor reaches an acquire point, it invokes a library procedure that issues a request with the lock address to the SSU. At this point, the SSU and processor proceed independently. 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. If the lock is busy, the SSU keeps "spinning" on it until the lock owner updates the lock, and the SSU receives a coherence message. (In practice, the SSU does not actually spin. Because it sits in the cache controller, it can simply wait for the coherence message before retrying.)

Meanwhile, the processor makes a backup copy of the architectural register state in hard-

```
                              pflg = !pflg;
                              lock (barlck);
                              cnt++;              //increment count
                              if (cnt == tot) {   //last one
                                 cnt = 0;         //reset count
                                 flg = pflg;      //toggle (S5)
                                 unlock (barlck);
                              }
L: ld  R1,lck1 ; (S1)         else {              //not last one
   bnz R1,L    ; (S2)            unlock (barlck);
   t&s R1,lck1 ; (S3)            while (flg != pflg); //spin (S6)
   bnz R1,L    ; (S4)         }
  (a)                         (b)
```

Figure 3. Example of conventional test&test&set lock-acquire (a) and barrier (b) code.

ware so that it can quickly roll back the speculative thread on a squash. To enable this, the library procedure for acquire includes a special checkpoint instruction, right after the request to the SSU. There is no need to flush the pipeline.

The processor continues execution into the critical section. As long as the Acquire bit is set, the SSU deems speculative all the processor's memory accesses after the acquire point in program order. The SSU must be able to distinguish these accesses from those that precede the acquire point in program order. This problem is nontrivial in processors that implement relaxed memory-consistency models, but we have successfully addressed it, as we describe elsewhere.[7] The SSU uses the Speculative bit to locally mark cache lines that the processor accesses speculatively. 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 ensures that a safe copy of the line stays in main memory. It also allows us to use the conventional Dirty bit in the caches (in combination with the Speculative bit) to mark cache lines that the processor has speculatively written. This is useful at the time the SSU squashes the thread, as we explain later.

Access conflict. The underlying cache coherence protocol naturally detects access conflicts. Such conflicts manifest as a thread receiving an external invalidation to a cached line, or an external intervention (read) to a dirty cached line.

If lines not marked speculative receive such external messages, the cache controller services them normally. In particular, messages to the lock owner or to any other safe thread never result in squashes, since none of their cache lines are marked speculative. The originator thread of such a message could be speculative. If so, normally servicing the request effectively supports *in-order* conflicts from a safe to a speculative thread without squashing.

If a speculative thread receives an external message for a line marked speculative, the SSU at the receiving node squashes the local thread. The originator thread can be safe or speculative. If it is safe, an *out-of-order* conflict has taken place, and the squash is warranted. If the thread is speculative, the squash is also warranted, since speculative synchronization

does not define ordering among speculative threads. In either case, the operation never squashes the originator thread.

Once the SSU triggers a squash, it

- gang-invalidates all dirty cache lines with the Speculative bit set;
- gang-clears all Speculative bits and, if the speculative thread has passed the release point, sets the Release bit (see the later discussion on lock release); and
- forces the processor to restore its checkpointed register state, which results in the thread's rapid rollback to the acquire point.

The gang-invalidation is a gang-clear of the Valid bit for lines whose Speculative and Dirty bits are set. Cache lines that the processor has speculatively read but not modified are coherent with main memory and can thus survive the squash.

If an external read to a dirty speculative line in the cache triggered the squash, 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 (modified-exclusive-shared-invalid) protocols, in which the directory queries a node for a line in Exclusive state that was silently displaced from the node's cache.

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 the controller finds no such candidate for eviction, the node stalls until the thread becomes safe or the SSU squashes it. Stalling does not jeopardize forward progress, since a lock owner always exists and will eventually release the lock. When the stalled thread becomes safe, it will be able to resume. Safe threads do not have lines marked speculative and, therefore, replace cache lines on misses as usual.

Lock acquire. The SSU keeps spinning on the lock variable until it reads a zero. At this point, it attempts a T&S operation (as **S3** in Figure 3

does for a conventional lock). If the operation fails, the SSU goes back to the spin test. If the T&S succeeds, the local processor becomes the lock owner. In the example in Figure 1a, this will happen to thread *C* after thread *A* releases the lock. The SSU then completes the action by resetting the Acquire bit and gang-clearing all Speculative bits, which makes the thread safe and commits all cached values. At this point, the SSU becomes idle. Other SSUs trying to acquire the lock will read that the lock is owned.

Lock release. The one exception to this lock-acquire procedure is when the speculative thread has already completed its critical section at the time the lock owner frees the lock. In general, once all the memory operations inside the critical section have completed, the processor executes a release store to the synchronization variable. If the SSU has already acquired the lock and is idle, the release store completes normally. However, if the SSU is still trying to acquire lock ownership, it intercepts the release store and clears the Release bit. By so doing, the SSU is able to remember that the speculative thread has fully executed the critical section. We call this event *release-while-speculative.* The SSU then keeps spinning because the Acquire bit is still set, and the thread remains speculative.

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 that bit is still set, the SSU issues the T&S operation to compete for the lock (lock-acquire procedure). If the Release 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 in the critical section. The SSU then *pretends* that the local thread has become the lock owner and released the lock instantly. It clears the Acquire bit, gang-clears all Speculative bits, and becomes idle. Thus, the thread becomes safe without the SSU ever performing the T&S operation. In Figure 1a, this is the action that threads *B* and *E* would take after thread *A* releases the lock.

Again, this is race-free for two reasons: First, the SSU of the speculative thread clears the Release bit only after *all* memory operations in the critical section have completed without conflict. Second, a free lock value indicates that the previous lock owner has completed the critical section as well.

Exposed SSU. At times, the programmer or parallelizing compiler might not want threads to speculate beyond a certain point—for example, if a certain access is irreversible, as in I/O, or will definitely cause conflicts. In these cases, the programmer or parallelizing compiler can force the speculative thread to spin-wait on the SSU state until the SSU becomes idle. Thus, the thread will wait until it either becomes safe or the SSU squashes it. (Naturally if the SSU is already idle, the thread will not spin-wait.) We call this action *exposing the SSU* to the local thread. In general, although we envision speculative synchronization to be transparent to application programmers and the compiler in practically all cases, we felt it was important to have the software accommodate the ability to expose the SSU.

Supporting multiple locks

When the SSU receives a lock request from the processor, it checks its Acquire bit. If that bit is set, the SSU faces a second acquire, which might be for a subsequent or nested lock, depending on whether or not the processor has executed a release store for the first lock.

If the request is to a lock that is different from the one the SSU is already handling, the SSU rejects the request, the processor does not perform a register checkpoint, and the speculative thread itself handles the second lock using ordinary T&T&S code (Figure 3). No additional support is required. Handling the second lock using ordinary T&T&S code is correct because the thread is speculative, so accesses to that lock variable are also speculative. When the thread reads the value of the lock, the SSU marks the line 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. Lock modification is confined to the local cache hierarchy, however, since the access is speculative. The SSU treats this second lock as speculative data. If the thread is eventually squashed, the squash procedure will roll back the thread to the acquire point of the first lock (the one the SSU handles). The SSU will then discard all updates to speculative data, including any speculative update to the second lock variable. Conversely, if the SSU completes action on the first lock and renders the thread safe, the SSU commits all speculatively accessed data, including the second lock variable itself. If the

thread was originally spinning on this second lock, it will continue to do so safely. Otherwise, any action the thread took speculatively 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.

Another multiple-lock scenario is when the second request goes to the lock the SSU is already handling. We resolve this case by effectively merging both critical sections into one.[7]

Speculative flags and barriers

To implement speculative flags, we leverage the release-while-speculative support in speculative locks. As we described earlier, after a release-while-speculative operation, the SSU is left spinning with the Release bit clear until the lock owner sets the lock to the free value. As soon as the lock is free, the speculative thread becomes safe 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 it does in a speculative lock request, except that it keeps the Release bit clear to avoid a T&S operation. The processor goes past the unset flag speculatively. Unlike in a speculative lock, of course, a squash does not set the Release bit. As part of a speculative flag request, the thread supplies the "pass" value of the flag to the SSU.

Programmers and parallelizing compilers often implement barriers using locks and flags.[1] Figure 3b gives an example. Because the SSU can implement both speculative locks and flags, support for speculative barriers is essentially at no cost.

Under conventional synchronization, a thread arriving early to a barrier updates barrier counter **cnt** and waits spinning on statement **S6**. The counter update is in a critical section protected by lock **barlck**. A thread should not 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. These conflicts, in turn, will force the SSU to roll back the first thread if it is still speculative, all the way to the synchronization point the SSU handles.

Even if the thread arrives at the barrier in a safe state, the critical section is so small that it is best to reserve the SSU for the upcoming flag spin (**S6**). Consequently, threads execute this critical section conventionally and speculate on the flag.

To support this behavior, the library code for barriers exposes the SSU to the thread before the thread attempts to acquire **barlck**, so speculative threads have a chance to become safe and commit their work. The thread then uses conventional synchronization to acquire and release **barlck**. Finally, when the thread reaches the flag spin (**S6**), it issues a speculative flag request and proceeds past the barrier speculatively. Later, as the last thread arrives and toggles the flag (**S5**), all other threads become safe and commit.

Multiprogramming and exceptions

In a multiprogrammed environment, the operating system can preempt a speculative thread. If so, the SSU rolls the preempted speculative thread back to the synchronization point and becomes available to any new thread running on that processor. When the operating system later reschedules the first thread somewhere, the thread resumes from the synchronization point, possibly speculatively. On the other hand, the operating system handles safe threads as it would with conventional synchronization. Because speculative synchronization is ultimately a lock-based technique, it might exhibit convoying under certain scheduling conditions. We address this issue extensively in our description of an adaptive enhancement to our basic mechanism.[7]

When a speculative thread suffers an exception, there is no easy way to know if the cause was legitimate; it could be due to the speculative consumption of incorrect values. Consequently, the SSU rolls back the speculative thread.

Software interface

Both programmers and parallelizing compilers widely use explicit high-level synchronization constructs such as M4 macros[5] and OpenMP directives[6] to produce parallel code. We can retarget these synchronization constructs to encapsulate calls to SSU library procedures, thereby enabling speculative synchronization transparently. Three basic SSU library procedures are sufficient: one to
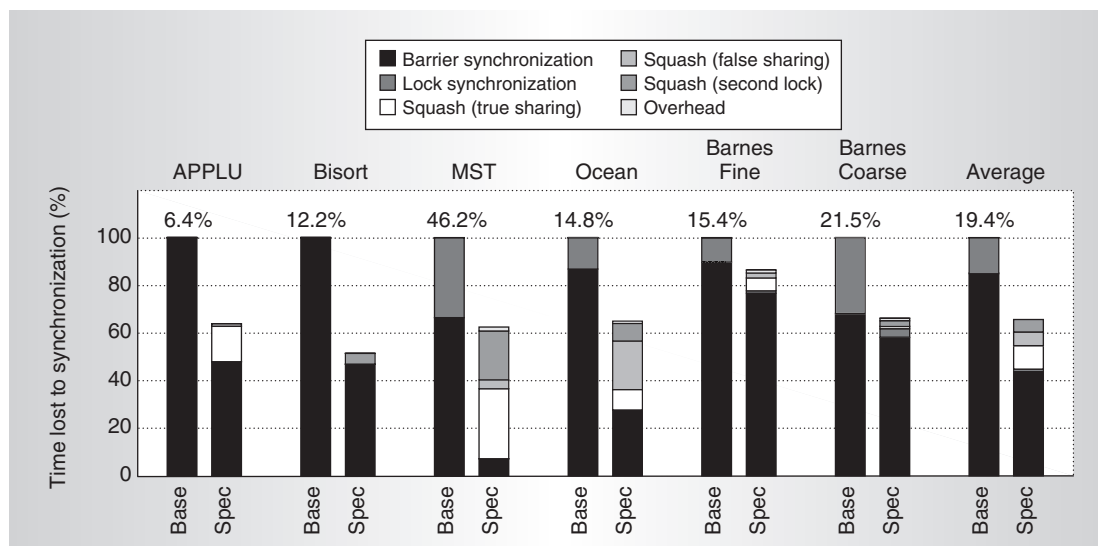
Figure 4. Factors contributing to synchronization time, squashed computation time, and squash overhead for conventional synchronization (Base) and speculative synchronization (Spec). The percentages at the top of the bars are the fraction of synchronization time in Base.

request a lock-acquire operation, one to request a flag spin operation, and one to test if the SSU is idle.

These three library procedures are enough to build macros for speculative locks, flags, and barriers.[7] Programmers and parallelizing compilers can enable speculative synchronization simply by using the modified macros instead of conventional ones. Conventional synchronization primitives are still fully functional and can coexist with speculative synchronization at runtime, even for the same synchronization variables.

Evaluation

To evaluate speculative synchronization, we used execution-driven simulations of a cache-coherent, non-uniform memory access multiprocessor with 16 or 64 processors,[7] and ran five parallel applications: a compiler-parallelized SPECfp95 code (APPLU); two Olden[9] codes (Bisort and MST), hand-annotated for parallelization; and two hand-parallelized Splash-2[10] applications (Ocean and Barnes). We used two configurations of Barnes: Barnes-Coarse has 512 locks and Barnes-Fine, the original configuration, has 2,048. The Splash-2 applications ran on 64 processors because they scale quite well, while the other applications ran on 16 processors.

Figure 4 compares the synchronization time under conventional synchronization (Base) to the residual synchronization time, the squashed computation time, and the squash overhead under speculative synchronization (Spec). The bars are normalized to Base. Figure 4 plots synchronization time separately for barriers and locks. Squashed computation time is due to true sharing, false sharing, and accesses to second-lock variables. The first two represent computation squashed because of conflicts induced by accesses to same and to different words of the same cache line, respectively. (Speculative synchronization keeps per-line Speculative bits and, therefore, false sharing results in conflicts.) The third one represents computation squashed because a speculative thread conflicts in a second-lock variable.

The figure shows that speculative synchronization reduces synchronization time on average 34 percent (including squashed computation time and overhead). Speculative threads stalling at a second synchronization point cause the residual synchronization time. Barriers account for most of this time, since already-speculative threads can never go past them (exposed SSU).

Some applications such as Ocean and MST exhibit a sizable amount of squash time. We discuss techniques to minimize sources of squashes in these applications elsewhere.[7]

Speculative synchronization successfully supports speculative execution past locks, flags, and barriers, in a unified manner using hardware of modest complexity. The presence of one or more safe threads in every active synchronization point guarantees the application's forward progress, even in the presence of access conflicts or insufficient cache space for speculative data. The support for speculative locks, under the right conditions (release-while-speculative case), allows speculative threads to commit the execution of a critical section without ever having to acquire the lock. Conflicting accesses are detected on the fly, and offending threads are squashed and eagerly restarted. Commit and squash operations take approximately constant time, irrespective of the amount of speculative data or the number of processors. Situations involving multiple locks are handled transparently and at no extra cost. Finally, speculative synchronization can easily be transparent to application programmers and parallelizing compilers, and can coexist with uses of conventional synchronization in the same program, even for the same synchronization variables. MICRO

References

1. J.L. Hennessy and D.A. Patterson, *Computer Architecture: A Quantitative Approach,* 3rd ed., Morgan Kaufmann, 2003.
2. G. S. Sohi et al., "Multiscalar Processors," *25 Years (ISCA): Retrospectives and Reprints*, ACM Press, 1998, pp. 521–532.
3. M. Herlihy and J.E.B. Moss, "Transactional Memory: Architectural Support for Lock-free Data Structures," *Int'l Symp. Computer Architecture* (ISCA 93), ACM Press, May 1993, pp. 289-300.
4. R. Rajwar and J.R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," *34th Int'l Symp. Microarchitecture* (MICRO-34), IEEE CS Press, Dec. 2001, pp. 294-305.
5. E. Lusk et al., *Portable Programs for Parallel Processors*, Holt, Rinehart, Winston, 1996.
6. L. Dagum and R. Menon, "OpenMP: An Industry-Standard API for Shared-Memory Programming," *IEEE Computational Science and Eng.*, vol. 5, no. 1, Jan.-Mar. 1998, pp. 46-55.
7. J.F. Martínez and J. Torrellas, "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," *Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, Oct. 2002, pp. 18-29.
8. J.F. Martínez and J. Torrellas, "Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors," *Workshop Memory Performance Issues*, June 2001; http://iacoma.cs.uiuc.edu/wmpi.pdf.
9. M.C. Carlisle and A. Rogers, "Software Caching and Computation Migration in Olden," *Symp. Principles and Practice of Parallel Programming*, ACM Press, July 1995, pp. 294-305.
10. S.C. Woo et al., "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Int'l Symp. Computer Architecture* (ISCA 95), ACM Press, July 1995, pp. 24-36.

José F. Martínez is an assistant professor of electrical and computer engineering at Cornell University. His research interests are in parallel computer architecture, microarchitecture, and hardware-software interaction. Martínez has a PhD in computer science from the University of Illinois at Urbana-Champaign and is a two-time recipient of the Spanish government's national award for academic excellence. He is a member of the IEEE Computer Society and the ACM.

Josep Torrellas is a professor of computer science and Willett Faculty Scholar at the University of Illinois at Urbana-Champaign and vice-chair of the IEEE Technical Committee on Computer Architecture (TCCA). His research interests are in computer architecture. Torrellas has a PhD in electrical engineering from Stanford University and is a recipient of an NSF Young Investigator Award. He is an IEEE Fellow and a member of the ACM.

Direct questions and comments about this article to José F. Martínez at Computer Systems Laboratory, Cornell University, 336 Frank H.T. Rhodes Hall, Ithaca, NY 14853-3801; martinez@csl.cornell.edu.