Wshp. on Memory Performance Issues, Intl. Symp. on Computer Architecture, June 2001.

Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors

José F. Martínez and Josep Torrellas
Department of Computer Science
University of Illinois at Urbana-Champaign
{martinez,torrellas}@cs.uiuc.edu

ABSTRACT

Multi-threaded applications typically use coarse- or fine-grain locks to enforce synchronisation when needed. While fine-grain synchronisation enables higher concurrency, it often involves significantly more programming effort than coarse-grain synchronisation. To address this trade-off, this paper proposes *Speculative Locks*. Speculative Locks are based on the concepts and mechanisms of speculative thread-level parallelisation. Threads access a critical section without synchronising, while the underlying hardware monitors for conflicting accesses. If a conflict is detected, threads are rolled back and restarted on the fly.

Overall, Speculative Locks allow the programmability of coarse-grain synchronisation while enabling the concurrency of fine-grain synchronisation. The presence of a lock owner at all times guarantees forward progress, and all in-order conflicts between owner and speculative thread are tolerated. Under the right conditions, a system with speculative locks and coarse-grain synchronisation performs about as well as one with conventional locks and fine-grain synchronisation.

1 INTRODUCTION

The choice of grain size used to synchronise in multi-threaded applications offers a trade-off between programmability and concurrency. While fine-grain synchronisation allows greater thread concurrency, it often requires greater development effort and, therefore, results in longer time to market. On the other hand, coarse-grain synchronisation, while restricting concurrency, delivers simpler and more stable software.

Furthermore, in some extreme cases, both fine-grain and coarse-grain synchronisation can fail to deliver the levels of performance expected from a program. For example, fine-grain synchronisation may penalise threads with costly overhead, even though the data accessed by different threads rarely overlap.

Ideally, we would like to combine the higher concurrency enabled by fine-grain locking with the higher programmability and lower overhead of coarse-grain locking. As an example from the commercial workload domain, we would like to combine the concurrency of tuple-level locking with the programmability and low overhead of table-level locking.

Interestingly, a recent development in computer systems architecture has been hardware support for speculative thread-level parallelisation [2, 7, 8, 12, 18, 19, 24]. Under such a technique, multiple threads that may have data and control dependences with each other are speculatively run in parallel. Typically, extensions to the cache coherence protocol hardware detect any dependence violation at run time. Any time one such violation is detected, the offending threads are squashed, rolled back to a safe state, and then eagerly restarted.

The concept of safe or *non-speculative* thread used in speculative thread-level parallelisation is very appealing for concurrent execution of critical sections. Indeed, having one such thread that owns the critical section while allowing others to traverse it under some conditions guarantees forward progress in case of an access conflict or other events like cache overflow or context switches.

We present a new scheme, *Speculative Locks*, for concurrent execution of critical sections in shared-memory multiprocessors. We borrow the principle of speculative thread-level parallelisation from existing literature to implement an efficient optimistic concurrency scheme. Speculative Locks allow the programmability of coarse-grain synchronisation while enabling the concurrency of fine-grain synchronisation. State commit and squash operations take constant time. Forward progress is guaranteed by the presence of a lock owner at all times, and all in-order conflicts between owner and speculative thread are tolerated. Under the right conditions, a system with speculative locks and coarse-grain synchronisation performs about as well as one with conventional locks and fine-grain synchronisation.

2 CONCURRENT EXECUTION OF CRITICAL SECTIONS

While proper labelling in Release Consistency guarantees race-free execution, the ordering restrictions it introduces are too strong. This is because the conditions for RC affect not only *conflicting* but *all* ordinary accesses [5]. While it is easy to envision a weaker version of RC in which restrictions apply only to conflicting ordinary accesses, a hardware implementation of such is by no means obvious [1].

Fig. 1 shows an example of two threads competing for access to a critical section. Suppose that the leftmost thread is granted access first. In the figure, arrows show some of the ordering restrictions imposed by RC. Conflicting accesses to location *X* are successfully made non-competing by ordering them through release and acquire operations of threads 1 and 2, respectively, as solid arrows show. However, many other unnecessary restrictions are introduced, represented by dotted arrows in the figure. To eliminate these, the programmer could implement the lock at a finer grain, provided that he can determine that only the store and load accesses to *X* are conflicting. In practice, this may prove impossible, or it may require a non-trivial effort that the programmer is unwilling to invest.

We want to eliminate all competing pairs within the critical section, while avoiding superfluous ordering restrictions. One way of doing this is to proceed optimistically as if no conflicting accesses exist, and use hardware to correct inconsistencies on the fly and eagerly restart offending threads. In the absence of conflicts at run time, several threads can effectively execute the critical section concurrently. Further, they may achieve partial concurrency in the presence of some conflicting accesses, since offending threads are restarted eagerly. We propose one such mechanism in the next section.

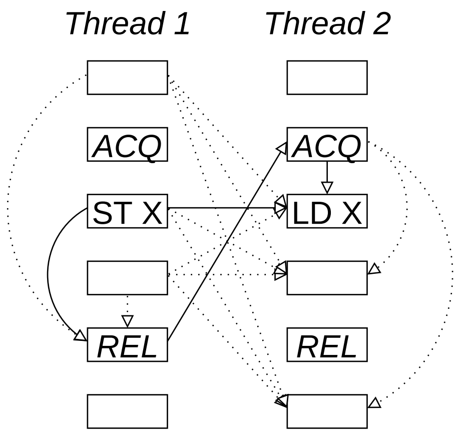

Figure 1: Critical section under RC, in which Thread 1 has gained access first. Empty boxes represent non-conflicting ordinary accesses. Ordering restrictions introduced by the release-acquire pair are shown, and dotted lines represent superfluous restrictions.

3 SPECULATIVE LOCKS

In this section we present Speculative Locks. First, we give a high-level description, and then propose an implementation using a device that we call *Speculative Concurrency Unit* (*SCU*). Our discussion assumes a properly labelled program under RC.

3.1 High-Level Description

A speculative lock allows threads to execute memory operations that belong to the critical section without actually owning it. Threads only need to *request* the speculative lock in order to enter its critical section. Only one thread at a time can own the lock. A thread granted access to a critical section without ownership of its lock is deemed *speculative*. Owner and speculative threads are free to execute memory operations inside the critical section. However speculative threads may not *commit* their results to the rest of the system until they are granted ownership. As in conventional locks that follow RC, threads in the critical section may also execute memory operations past the release point, but speculative threads may not commit these until ownership is granted, either.

When a thread is granted ownership of a speculative lock we say that the thread has *acquired* the lock, and the thread is considered *non-speculative*. Threads not competing for the critical section are also regarded as non-speculative. Non-speculative threads commit their memory operations as they complete. A thread may *release* a speculative lock only when (1) it has been granted ownership, and (2) it has issued and completed all its memory operations within the critical section.

Speculative locks must preserve the semantics of conventional locks, i.e. the final outcome must be as if the threads had gained access to the critical section one at a time. In particular, a speculative lock must ensure that conflicting accesses by a speculative thread and the owner are executed in order, i.e. the owner's first, followed by the speculative's. If the system detects an *out-of-order* conflict between a speculative and a non-speculative thread, it must *squash* the former and force a roll-back to the acquire point. Ordering is undefined among speculative threads, thus if two speculative threads issue conflicting accesses, one of them is squashed and restarted. Overall, therefore, speculative threads operate optimistically, assuming no out-of-order conflicting accesses with non-speculative threads and no conflicting accesses at all with other speculative threads. In all cases, non-speculative threads, in particular the owner, must not be disturbed in order to guarantee *forward progress*.

Eventually, a speculative lock is released by its owner. Speculative threads that have already completed all accesses within the critical section without being squashed due to conflicts with other threads can commit their memory operations and become non-speculative at once. Then, one of the speculative threads inside the critical section, if any, will acquire the lock, also committing its memory operations. This is equivalent to a conventional lock in which the owner releases the lock; then, all speculative threads past the release point, in some order, acquire the lock, execute the critical section, and release the lock; and finally, another thread competing for the lock acquires ownership and enters the critical section.

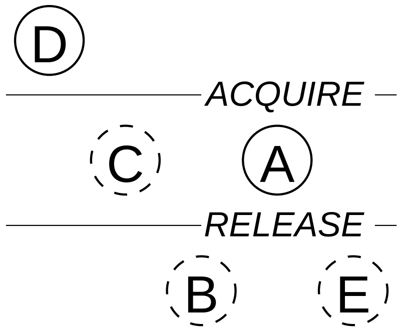

Figure 2: Example of a speculative lock. Dashed and solid circles denote speculative and non-speculative threads, respectively. Thread A is currently the owner of the lock.

Fig. 2 shows an example of a speculative lock with five threads. Threads A and D are non-speculative, while threads B, C, and E are speculative. Thread A is currently the owner of the lock. Thread C is also in the critical section, but it does not own the lock and hence its speculative state. Threads B and E have executed the critical section completely and are now executing the code after the release point, but they remain speculative since they still have not acquired the lock. Lastly, thread D is non-speculative since it has not reached the acquire point.

If thread A now leaves the critical section, threads B and E can become non-speculative at once, and thread C becomes the new lock owner, turning non-speculative as well. This is equivalent to a conventional lock whose critical section is traversed in order (A, B, E, C) or (A, E, B, C).

3.2 Implementation

Our implementation of speculative locks builds inexpensively on top of a conventional shared-memory multiprocessor with small modifications to the cache hierarchy and cache coherence protocol. The core of our implementation is the *Speculative Concurrency Unit* (*SCU*). The SCU sits next to the cache and is comprised by an array of *Speculative* bits with as many entries as cache lines, a single *Owner* and a single *Release* bit, one extra cache line,

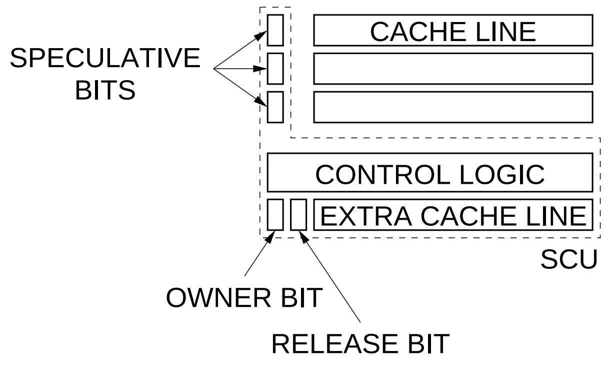

Figure 3: Speculative Concurrency Unit. Notice the Speculative bits by each cache line and the additional logic that comprise the device. Three cache lines are shown.

and some small control logic. Fig. 3 depicts a cache and its SCU. For the sake of simplicity, we will assume a single level of cache; extension to multiple levels is straightforward if inclusion is preserved. In a multi-level cache hierarchy, the SCU would be associated to the most external cache, while the other caches would have simple logic support.

Traditional systems use different operations to implement acquire and release operations, such as Test&Set (T&S), Test&Test&Set (T&T&S), Load Linked/Store Conditional, Compare&Swap, or Fetch&Op [3]. The processor usually spins using one of these operations until the lock becomes available.

We decouple this task from the processor and make the SCU do it instead. Specifically, once the processor reaches the acquire point, it issues a request to its SCU with the address of the lock. The SCU records the address in the extra cache line, resets the Owner and Release bits, and initiates a “spin-locking” loop. To give a concrete example, we will assume that the SCU uses T&T&S.

Meanwhile, the processor checkpoints its internal state and then continues execution past the acquire point. State checkpointing is a feature already present in existing processors, such as the MIPS R10000 [23]. Memory accesses by the processor past the acquire point are deemed speculative by the SCU if the Owner bit is not set.

Handling of the Lock

The SCU remembers the speculative accesses by setting the Speculative bit of the cache lines accessed in speculative mode by the local thread. Aside from this fact, all speculative accesses are handled by the underlying cache coherence protocol. There is one small exception: should a speculative thread perform a first access to a cache line that resides dirty in any cache, including its own, the coherence protocol must force a write-back, in order to keep a safe copy in main memory at all times. Cache lines whose Speculative bit is set may not be evicted.

We now go back to the SCU, at the time it executed T&T&S to compete for ownership. The first access in T&T&S is a read request on the lock variable. If the read reveals that the lock is already set, the SCU keeps spinning locally on it until the copy is updated externally (which typically signals a release operation by the owner). When the lock becomes available, the SCU attempts a T&S operation. If the operation fails, the SCU goes back to the spin-test. Otherwise, the Owner bit is set and all Speculative bits are cleared in one shot, effectively committing all values to the system in an instant.

Early Release

As in conventional locks, a processor issues a lock release only after it has completed all its memory operations within the critical section. Upon detecting the lock release, typically a store operation on the lock variable, the SCU checks the Owner bit. If it is already set, all the SCU does is reset the value of the lock variable, which effectively releases the lock in the system. If, instead, the Owner bit is still not set, the SCU does not reset the lock variable, but sets the Release bit. The SCU then keeps waiting for ownership. In neither case is the execution of the speculative thread interrupted. Remember that threads in the critical section can execute instructions after the release point.

When a SCU that does not yet own the lock receives an external invalidation to the lock variable[1],

[1]We assume that the only exclusive request to the lock variable that can take place is the lock release by the lock owner. If other exclusive accesses (e.g. T&S operations, exclusive prefetches, etc.) are allowed, the system must differentiate this invalidation message from others.

the Release bit is first checked. If the bit is set, the SCU knows that the thread has completed all memory operations prior to the release, and can aggressively assume that ownership is acquired and released instantly. Therefore, all Speculative bits are cleared in one shot and the SCU regards the critical section as successfully traversed. In this case, the T&S operation will never take place, allowing other speculative threads to compete for the lock.

Conflicting Accesses

If a node receives an external invalidation for a cache line marked Speculative, or an external read for a dirty cache line marked Speculative, the SCU realises that a conflict has taken place. It then forces a squash by invalidating in one shot all dirty lines with the Speculative bit set, resetting the Release bit and all Speculative bits, and forcing the processor to roll back to its shadow state. Notice that we do not invalidate lines that have been speculatively read but not modified, since they are consistent with main memory. If the requested speculative line was itself dirty, the node will reply to the home node without supplying any data, forcing the home node to regard the state for that line as stale and supply a clean copy from memory to the requestor. 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 replaced. External requests to cache lines not marked Speculative are processed normally. Non-speculative threads, in particular the owner, can never be squashed, since none of their cache lines is ever marked. This allows all in-order conflicting accesses between owner and speculative threads to be tolerated without causing squashes.

Notice that the SCU does not differentiate between external requests triggered by speculative or non-speculative threads. This could potentially lead to a ping-pong effect involving multiple, recalcitrant speculative threads that issue conflicting memory accesses. Fortunately, the owner of the critical section is never affected by this, thus forward progress is always guaranteed. The ping-pong effect between speculative threads can be easily limited by turning off speculation in a lock for a thread exceeding a certain number of restarts.

Cache Overflow

To preserve data integrity, cache lines whose Speculative bit is set cannot be selected as replacement victims upon a cache miss. In the event of a memory access not being serviceable due to lack of evictable entries (cache overflow), the node stalls until ownership of the lock is granted or a squash is triggered. Stalling does not jeopardise correctness because, again, the fact that there is an owner thread at all times that cannot stall guarantees forward progress: the lock will be eventually release, handing ownership to another thread. To reduce the chances of a stall due to overflow, a victim cache extended with Speculative bits can be provided in a manner similar to [2].

3.3 Multiple Locks

Only one lock at a time can be handled by the SCU. Subsequent locks must be handled as conventional synchronisation accesses beyond the first one. We envision a lock acquire procedure to be programmed so that it reads the state of the SCU. If the SCU is available, the program then uses it to run the lock speculatively as we have seen so far. If, instead, the SCU is busy, the program deals with the lock using conventional T&T&S code. Locks so handled will expose speculative threads to squashes due to conflicts on the lock variables themselves. When squashed, they will roll back to the acquire point of the first lock, controlled by the SCU. Further discussion of this situation is beyond the scope of this paper.

3.4 Summary

The SCU is a simple hardware device that implements speculative locks at a modest cost and requires few modifications to the cache hierarchy and coherence protocol. The most salient features of our scheme are:

– Threads operate optimistically, assuming no conflicting accesses with other threads. Conflicting accesses are detected on the fly by the coherence protocol, and offending threads are squashed and eagerly restarted.

– Commit and squash operations take constant time, irrespective of the amount of speculative data or the number of processors.

– Forward progress is guaranteed by forcing one lock owner to exist at all times. The owner can never be squashed due to conflicts, or stall due to cache overflow.

– All in-order conflicting accesses between the owner and a speculative thread are tolerated and thus do not cause squashes.

4 EVALUATION

4.1 Experimental Set-Up

We evaluate the potential of speculative locks by simulating a simplified synthetic version of the TPC-C OLTP benchmark. The results show the advantages of a system featuring speculative locks over a conventional configuration.

We use a simplified synthetic model of TPC-C composed of five branches, five tellers per branch, and a number of accounts per teller that ranges from five to one thousand. Locks can be placed at any level: account, teller, branch, or even global. Fig. 4 depicts such a system.

A 64-way multiprocessor hosting the OLTP system processes one million transactions. Each transaction is modelled as follows. First, it computes the branch, teller, and account to access using three random numbers. Then, it secures whatever lock is necessary. After some internal pre-processing, it reads the balance from the account, performs some internal manipulation, and writes the updated balance to the account. Finally, it does some internal post-processing, and then it releases the lock. The duration of the pre-processing, balance manipulation, and post-processing is chosen randomly. We use a uniform distribution with a range of one to seven time units.

We model configurations with conventional and speculative locks. In the case of conventional locks, a transaction whose lock is already taken blocks. When the owner frees it, a new owner is chosen randomly among the contendants. In the case of speculative locks, a transaction starts processing whether its lock is available or not. The hardware monitors for conflicting accesses to the accounts, and squashes and restarts offending transactions on the fly as needed.

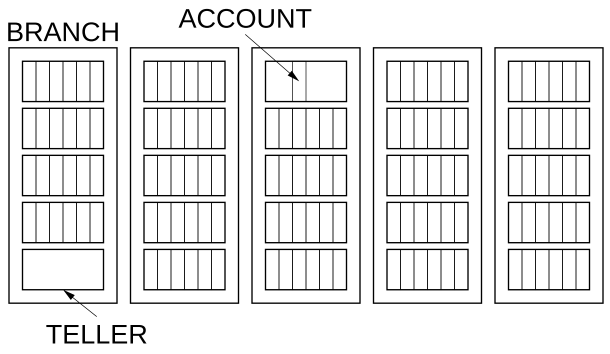

Figure 4: Organisation of the synthetic on-line transaction processing system used in the evaluation.

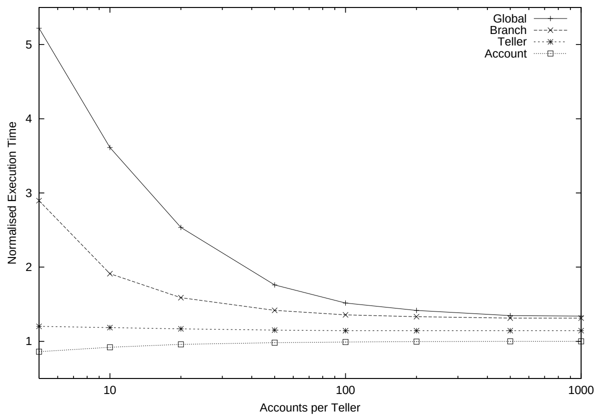

Figure 5: Execution time of different systems featuring speculative locks, normalised to that of a system that uses fine-grain (per account) conventional locks. Curves for speculative locks at all possible grain sizes (account, teller, branch, and global) are shown.

A processor is assigned transactions dynamically, but it cannot initiate a new transaction until the resident one has graduated. Speculative transactions must wait for ownership of the lock before graduating. To the disadvantage of our scheme, transactions do nothing beyond the release point. This will stall the processor if the lock is still unavailable by the time its transaction hits the release point. If transactions had work to do past the release point, they could continue processing while pending ownership.

4.2 Results

In our experiment we evaluate the performance of speculative locks at all grain levels: account, teller, branch, and global. In all four cases, we measure the execution time considering different number of accounts per teller, from five to one thousand. We compare the times obtained against those obtained for a conventional system that features locks at the account level, the finest possible grain. Fig. 5 shows the results.

The results of this experiment show that systems using speculative locks coarser than the baseline quickly approach the performance of the latter, as the number of accounts per teller increases. This is because the larger number of accounts in the system makes the probability of conflict smaller, which the systems with speculative locks exploit. That conventional systems can be closely followed in performance by systems featuring speculative locks and much coarser grain locking is very good news. It means that we can reduce the programming complexity significantly with coarse-grain locks without compromising the code performance.

We also observe that systems using speculative locks perform better the finer the grain is. This is because, with coarser-grain locks, the competition for ownership is fiercer, making speculative transactions wait longer to commit before they can be retired. The chances of conflict for each transaction increase with the time that it spends in speculative state. Of course, this effect quickly loses relevance as the number of accounts increases.

Finally, we note that the system that uses speculative locks at the account level slightly (but consistently) outperforms the baseline system. This is despite the fact that, since transactions access a single account, all transactions using the same lock are guaranteed to conflict. Speculative locks outperform the baseline because they *tolerate conflicts* if they happen in order with the owner, something that the conventional system cannot exploit.

5 CONCLUSIONS

We have presented Speculative Locks, a hardware mechanism that allows the programmability of coarse-grain synchronisation while enabling fine-grain concurrency. We have borrowed the principle of speculative thread-level parallelisation from existing literature to implement an efficient scheme of optimistic concurrency control. Threads access a critical section concurrently without synchronising, and the system uses the underlying coherence protocol to continuously monitor for conflicting accesses on the fly, rolling back and eagerly restarting offending threads. Commit and squash operations take constant time, irrespective of the amount of speculative data or the number of processors.

We maintain a legitimate owner of the critical section at all times in order to guarantee forward progress. Owners can neither get squashed due to conflicts, nor stall due to cache overflow, and all in-order conflicting accesses between owner and speculative thread are tolerated without squash.

In the absence of conflicts, the system allows multiple threads to execute the critical section concurrently. Further, partial concurrency can be achieved in the presence of some conflicting accesses, due to the eager restart mechanism. Under the right conditions, a system with speculative locks and coarse-grain synchronisation performs about as well as one with conventional locks and fine-grain synchronisation.

REFERENCES

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. thesis, Univ. of Wisconsin at Madison, December 1993.

[2] M. H. Cintra, J. F. Martínez, and J. Torrellas. “Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors”. *Intl. Symp. on Computer Architecture*, June 2000.

[3] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: a Hardware/Software Approach*. Morgan Kaufmann Publishers, 1999.

[4] K. Gharachorloo, A. Gupta, and J. L. Hennessy. “Two Techniques to Enhance the Performance of Memory Consistency Models”. *Intl. Conf. on Parallel Processing*, August 1991.

[5] K. Gharachorloo, D. E. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. “Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors”. *Intl. Symp. on Computer Architecture*, May 1990.

[6] C. Gniady, B. Falsafi, and T. N. Vijaykumar. “Is SC+ILP=RC?”. *Intl. Symp. on Computer Architecture*, June 1999.

[7] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. “Speculative Versioning Cache”. *Intl. Symp. on High Performance Computer Architecture*, February 1998.

[8] L. Hammond, M. Wiley, and K. Olukotun. “Data Speculation Support for a Chip Multiprocessor”. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[9] M. Herlihy and J. E. B. Moss. “Transactional Memory: Architectural Support for Lock-Free Data Structures”. *Intl. Symp. on Computer Architecture*, May 1993.

[10] M. Herlihy. “A Methodology for Implementing Highly Concurrent Data Objects”. *ACM Trans. on Programming Languages and Systems*, Vol. 15, No. 5, November 1993.

[11] M. Herlihy. “Wait-Free Synchronization”. *ACM Trans. on Programming Languages and Systems*, Vol. 11, No. 1, January 1991.

[12] V. Krishnan and J. Torrellas. “A Chip-Multiprocessor Architecture with Speculative Multithreading”. *IEEE Trans. on Computers, Special Issue on Multithreaded Architectures*, September 1999.

[13] H. T. Kung and J. T. Robinson. “On Optimistic Methods for Concurrency Control”. *ACM Trans. on Database Systems*, Vol. 2, No. 6, June 1981.

[14] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. “An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors”. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[15] P. Ranganathan, V. S. Pai, and S. V. Adve. “Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models”. *Symp. on Parallel Algorithms and Architectures*, June 1997.

[16] M. C. Rinard. “Effective Fine-Grain Synchronisation for Automatically Parallelised Programs Using Optimistic Synchronisation Primitives”. *ACM Trans. on Computers and Systems*, Vol. 17, No. 4, November 1999.

[17] D. Shasha and M. Snir. “Efficient and Correct Execution of Parallel Programs that Share Memory”, *ACM Trans. on Programming Languages and Systems*, Vol. 10, No. 2, April 1988.

[18] G. Sohi, S. Breach, and T. Vijaykumar. “Multiscalar Processors.” *Intl. Symp. on Computer Architecture*, pages 414-425, June 1995.

[19] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. “A Scalable Approach to Thread-Level Speculation”. *Intl. Symp. on Computer Architecture*, June 2000.

[20] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. “Multiple Reservations and the Oklahoma Update”. *IEEE Parallel and Distributed Technology*, Vol. 1, No. 4, November 1993.

[21] A. Thomasian. “Concurrency Control: Methods, Performance, and Analysis”. *ACM Computing Surveys*, Vol. 30, No. 1, March 1998.

[22] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. “Supporting Fine-Grain Synchronisation on a Simultaneous Multithreading Processor”. *Intl. Symp. on High-Performance Computer Architecture*, January 1999.

[23] K. Yeager. “The MIPS R10000 Superscalar Microprocessor”. *IEEE Micro*, Vol. 6, No. 2, April 1996.

[24] Y. Zhang, L. Rauchwerger, and J. Torrellas. “Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors”. *Intl. Symp. on High-Performance Computer Architecture*, February 1998.