Speculative Lock Reordering: Optimistic Out-of-Order Execution of Critical Sections

Peter Rundberg and Per Stenström

Department of Computer Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
{*biff,pers*}*@ce.chalmers.se*

Abstract

We present a new method – Speculative Lock Reordering *(SLR) – that enables multiple threads to concurrently and speculatively execute within a critical section. Its novel feature is to exploit that there is no a priori execution order between separate invocations of a critical section that speculatively executed threads must respect. In contrast to previously proposed speculative synchronization schemes, we show that since an execution order can be selected that removes as many data dependences as possible, SLR can expose more concurrency. Additionally, it is shown that SLR can be implemented in a chip-multiprocessor by only modest extensions to already published thread-level data dependence speculation systems.*

1. Introduction

Critical sections are a key mechanism to implement atomicity of compound modifications to data structures in a parallel program. A well-known performance problem with critical sections, however, is that they may limit the amount of concurrency by serializing concurrent accesses to the protected data structures. While limiting the scope of the critical sections could potentially expose more concurrency, it requires more detailed analysis by the programmer to avoid violating the correctness of the program. In addition, finer-grain critical sections lead to more synchronization overhead. Overall, this leads to a delicate trade off between concurrency on one hand, and programming effort and synchronization overhead on the other.

Thread-level speculation systems can be used to expose more concurrency, without any additional programming effort. As long as there are no data dependences between the concurrently executed speculative threads, speculation will succeed; otherwise some or all threads must be re-executed serially. Inspired by the quite extensive body of work on thread-level data dependence speculation systems [2, 3, 10], there have been attempts to apply this approach to synchronization primitives such as barriers [11, 7], and critical sections [7, 8, 9].

Independently of our investigations, two recent schemes that enable speculative execution of threads within a critical section have been reported [7, 8]. While these schemes allow threads to enter and execute critical sections concurrently, the order by which they enter or exit dictates the execution order. However, adhering to such a strict execution order restricts the opportunities to remove data dependence violations. To see how, imagine that the first thread modifies data whereas the second one reads the data within a critical section. If they execute concurrently, and the modification happens after the read operation, the speculation will fail. Since sequential consistency [4] does not state any a priori assumption about the order by which the threads execute a critical section, data dependence violations could have been avoided if the speculation system picked the reverse execution order.

In this paper, we contribute with a new scheme for speculative execution of critical sections that exploits this fundamental ordering property with the aim of eliminating more data dependence violations than previous schemes. Our new scheme does this by not assigning an execution order between the speculatively executed threads until all of them have reached the exit (or unlock) point of the critical sections. The execution order is then defined in such a way that it minimizes the number of dependence violations. However, to do that, speculative modifications need to be buffered. As a result, the bypassing feature of the method proposed by Martínez *et al.* [7] can not be exploited. In addition, since the detection of misspeculations is postponed until the time of the commitment, and not when they happen

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)
IEEE COMPUTER SOCIETY

```
LOCK(locks->error_lock)
if (local_error > multi->err_multi)
    multi->err_multi = local_err;
UNLOCK(locks->error_lock)
```

Figure 1. Critical section in Ocean.

[7, 8, 9], it cannot perform an *eager restart*, i.e., squashing and restarting a conflicting thread immediately.

Our analysis, based on micro-benchmark analysis of the critical section executions in SPLASH2 [13] reveals that our method can extract more concurrency. On the other hand, this is partly compensated by the more aggressive strategy to cut down on the misspeculation penalty in the competing schemes. Unfortunately, we find that the bypassing capability of Martínez's method does not prove to be very successful. In addition, while the eager restart capability can increase the concurrency to some extent, the large number of eager restarts may also cause severe overhead. Finally, we sketch how our scheme can be implemented in a typical thread-level dependence speculation system for a chip-multiprocessor with fairly modest extensions of such an infrastructure.

We first review other methods and identify their weaknesses in Section 2. Our method is then presented in Section 3 The experimental approach and the results obtained are discussed in Sections 4 and 5. Finally, an implementation of our scheme is sketched in Section 6 after which we conclude in Section 7.

2. Previous Approaches

We focus on a chip-multiprocessor in which an L1 cache is attached to each processor. A snoopy-cache protocol maintains consistency across the L1 caches through a shared-bus to which an L2 shared cache is attached. Thread-level speculation systems are typically embedded in the cache coherence schemes. How this is done, and how it can support thread-level parallelism within critical sections is delegated to Section 6. We now review previous methods to support speculative concurrent execution of critical sections.

2.1. Martínez's method

Martínez's method works as follows. When a thread reaches a critical section whose lock is free, it enters the critical section in a conventional non-speculative manner and is referred to as the safe thread. However, subsequent threads enter without acquiring the lock and executes speculatively and keep all modifications in a local buffer; the local processor cache. Potential dependence violations are monitored through the cache coherence mechanism.

```
LOCK(ACell->space_lock)
space_value = ACell->space;
if (space_value < 0) {
  UNLOCK(ACell->space_lock)
  .
  .
  LOCK(ACell->space_lock)
}
.
.
space_value = ACell->space;
if (space_value == 0 {
  ACell->space = pn;
} else {
  if (space_value > 0) {
    ACell->space = 0;
.
.
UNLOCK(ACell->space_lock)
```

Figure 2. Critical section in Mp3d.

When the safe thread eventually releases the lock, all speculative threads compete for the lock and the one that succeeds will first commit its data and then become the new safe thread. As a result, in absence of misspeculations, all speculative threads will commit their data at the exit point after which their execution can return to non-speculative mode.

Let's now see how this method resolves data dependence violations by considering the code segment in Figure 1 that shows one of the critical sections in `ocean` from the SPLASH2 benchmark suite [13]. Note that while the variable `multi->err_multi` is always read, it is sometimes subsequently updated conditionally on its value. As a result, when the variable is only read, multiple threads can successfully execute the critical section concurrently.

Assuming that two threads simultaneously execute the critical section under Martínez's method, there are three cases: (1) no thread writes to the shared variable, (2) exactly one thread writes to the shared variable, and (3) both threads write to the shared variable. While speculation will succeed in the first case, the second and third case will fail if the safe thread performs any operation on the shared data after the speculative threads touches the data. This is quite likely as the write happens late in the critical section.

Figure 2 shows a critical section in the `mp3d` code from the SPLASH benchmark suite [12]. Here, atomic updates to the shared variable `ACell->space` is ensured through the use of a lock. If `ACell->space` is less than zero this critical section will be split into two protected by the same lock; only one involving reading the variable and one involving both reading and writing. By allowing speculative execution of critical sections, some unnecessary synchronization can be avoided. If only read operations are involved, Martínez's method can allow concurrent execution of this critical section. If, however, one thread executes the then-clause of the first if-statement whereas another exe-

IEEE COMPUTER SOCIETY

cutes the else-clause, Martínez's method can cause a dependence violation depending on which thread that will arrive first. The only rescue is if the first thread can bypass data to the second thread. Another way to cut down on the penalty of misspecultion is to enable 'eager restart' which can potentially cut the misspeculation penalty.

In summary, while this method can allow more concurrency, a data dependence violation can hurt performance if bypassing and/or eager restart are not effective. This tradeoff will be later explored in the micro-benchmark analysis.

2.2. Rajwar's method

Rajwar's method [9] , called *Speculative Lock Elision* or *SLE*, allows threads to enter a critical section without acquiring the lock. This optimization leads to less memory traffic because on a successful speculative execution of the critical section, the lock variable is not touched. Threads in the critical section are speculative and their results are buffered in a local write buffer. By keeping speculative state in the write buffer, speculative writes can be merged, removing unnecessary writes in the memory system and also removing silent stores [5].

The execution order between speculative threads conforms to the order by which they exit the critical section. As a result, the bypassing capability of a safe thread in Martínez's method cannot be used as the execution order is not defined for speculative threads. Dependence violations are detected when a thread commits its data and the conflicting thread is then rolled back. Recalling the examples in Figures 1 and 2, Rajwar's method will misspeculate in all cases involving a write operation.

3. Speculative Lock Reordering (SLR)

Sequential consistency [4] means that the result of the execution should be as if all operations are executed in *some* sequential order and the operations of each individual thread appear in the order specified by its program. An operation can be a single load or store but can also be a critical section because of its atomic execution of loads and stores.

A chief observation is that individual operations by different threads can appear in any order as long as all threads have a consistent view of that sequential order. So, if three threads execute a critical section, the result of their execution should conform to some sequential order. As we will see, this property of sequential consistency is exploited by SLR to establish an order by which the number of dependence violations is minimized.

SLR allows multiple threads to enter a critical section but no one gets ownership of the lock at this time. Instead, all of them execute speculatively and buffer modifications locally to remove name dependences through renaming. While data

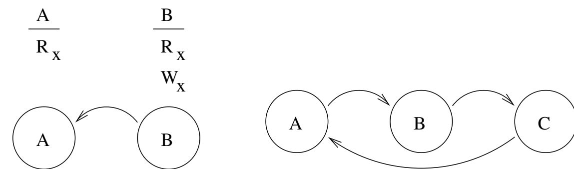

Figure 3. Left: Thread dependency graph for Ocean. Right: An example of a cyclic thread dependency graph.

dependence violations can potentially happen, they are only logged (to be explained later) and do not result in squashing and re-execution of the threads. When a thread reaches the release point, it will become non-speculative first when all other threads in the critical section reaches the release point. Meanwhile, it continues to execute past the release, in speculative mode, until the next lock acquire operation, where it stalls awaiting all threads to reach the release point. This approach conforms to Release Consistency [1] that allows a release to be postponed until the next acquire. An important consequence is that it guarantees that a thread cannot re-enter the same critical section before all threads have become non-speculative.

When all speculative threads have exited the critical section, an execution order that aims at eliminating all dependence violations is established. A directed dependence graph called *thread dependency graph* is constructed. In this graph each vertex represents a thread identity and arcs represent data dependences between them. With this graph, it is possible to identify (1) which threads are independent of each other, (2) a *commit order* between threads depending on each other that resolves data dependences between them, and (3) which threads that have to be re-executed to resolve any remaining dependences.

Consider again the example from `ocean` in Figure 1, where we assume that thread A does only a read operation whereas thread B does a read-modify-write. Now by executing them in parallel and committing thread A first, a data dependency will be converted into a name dependency. Since name dependences are eliminated through local buffering, the SLR scheme in this case avoids the costly violation that previous methods would suffer from. The same opportunity can be exploited in the example from `mp3d` in Figure 2 assuming that one thread does a read whereas the other does a read-modify-write.

The left part of Figure 3 shows the thread dependency graph of the `ocean` example. As all writes in the critical sections are kept in local buffers for each thread, a valid execution order of these critical section invocations can be constructed by committing the state of the threads in the opposite dependence order, i.e., A and then B. In general, for an acyclic thread dependency graph, if we commit threads starting from the sink and work backwards to the source, all

IEEE COMPUTER SOCIETY

data dependences can be resolved. If two threads depend on a third but not on each other, they can be committed in an arbitrary order.

In the right part of Figure 3 there is a cyclic dependency between three threads. It is impossible to find a global commit order that would conform to a serial execution order of the threads. However, if a thread is re-executed, all its dependences with other threads will be eliminated. The circular dependency chain can fortunately be broken by selecting any of the three threads for re-execution. Then one removes all arcs to and from this thread in the graph. For example, if thread C is selected for re-execution, we can find a commit order between A and B that is consistent with a correct serial execution order in which B is executed followed by A. In general, a strategy that will expose as much concurrency as possible is to remove as few threads as possible so that the thread dependency graph becomes acyclic.

3.1. Qualitative Comparison of the Schemes

Table 1 summarizes the key features of the methods presented above. We note that the methods mainly differ in two respects: (1) what mechanism is used to eliminate data and name dependence violations and (2) whether they can reduce the cost of the remaining data or name dependences and how.

	SLR	SLE	Martínez
Data dep.	Reorder	No	Bypass
Name dep.	Reorder	No	No
Misspec. penalty reduction	Cycle removals	Eager restart	Eager restart

Table 1. Comparison between SLR, SLE (Rajwar), and Martínez's method.

First, SLE will cause a misspeculation in all cases. Martínez's method can resolve some data dependences through bypassing. By contrast, through reordering SLR can convert data dependences into name dependences. Because SLR provides local buffering, name dependences are eliminated through renaming in case an execution order is chosen that eliminates data dependences. Second, Martínez's method and SLE both adopt eager restart of the offending thread whereas SLR can select a minimum set of threads to be re-executed so that circular dependency chains are broken.

Finally, there is an interesting tradeoff between how especially Martínez's method and SLR manage to eliminate and reduce the penalty of dependence violations. In the next section, we analyze this tradeoff quantitatively.

4. Experimental Methodology

We have used the lock intensive programs from the SPLASH-2 suite [13]. While the programs in this suite are well-tuned, and critical sections are not critical to the performance, we use the critical sections in this suite as microbenchmarks. The programs we have picked include **Barnes**, **Cholesky**, **FMM**, **Ocean**, **Radiosity**, **Raytrace**, **Volrend** and **Water-Nsq**. The programs not included in the evaluation, **FFT**, **LU**, and **Radix**, are all synchronized using other methods (mainly barriers). In Table 2 we show the problem sizes used for each application.

Program	Problem Size	Unique Lock Paths
Barnes	16K particles	68815
Cholesky	tk15.O	617
FMM	2K particles	1059
Ocean	130 x 130 ocean	10
Radiosity	room, -ae 5000.0 -en 0.05 -bf 0.1	67408
Raytrace	car.env	3099
Volrend	head-scaleddown4	32
Water-Nsq	512 molecules	2216

Table 2. Application characteristics.

We have constructed the microbenchmarks by extracting execution traces from the critical sections contained in the applications. For each application, we collected an execution trace that contains memory references to shared data and the time between consecutive memory references for each critical section invocation, including the initial lock acquisition and the final lock release.

To derive a parallel execution trace for two processors, we sorted the execution traces in such a way that all traces associated with the same lock variable were inserted into the same bin. Each such unique trace is a thread. For all threads in a bin, we constructed all distinct pairs of threads. Given a particular lock variable bin that contains N distinct traces, we could thus construct $M = \sum_{i=1}^{N-1}(i) = \frac{N(N-1)}{2}$ pairs of execution traces for that lock variable. The parallel execution trace thus contains M pairs of critical section memory reference traces that are assumed to execute one after the other. For L lock variables, and assuming M_k pairs of traces for lock variable k, the parallel execution trace is constructed by concatenating L traces. The parallel execution trace of one application thus contains all possible combinations of critical section invocations, assuming two threads, that result from the given input data with a barrier in between each pair of traces.

The sequential execution trace is simply the concatenation of all critical section invocation traces in the parallel

IEEE COMPUTER SOCIETY

trace, one after the other.

To generate the execution trade, we used Simics [6] to generate memory reference traces for all invocations of the critical sections in the application suite by turning on the tracing facility at the point of a lock acquisition and turning it off at the release of a lock. Each item in the execution trace is a memory reference and the time to the next one assuming that memory references take unity time. The traces can then be used to derive the execution time assuming no penalty in the memory system.

In the ideal case when all execution traces associated with a particular lock variable are equally long, the speedup assuming two threads is two. The deviation from this ideal number is attributed to load imbalance and the ability by which each of the two speculation methods can resolve data dependences. For Martínez's method, it is dictated by the ability to bypass data or the ability to salvage some of the lost execution opportunities by eager restart when a data dependence violation is signaled. For the SLR method, it is dictated by the extent by which reordering of the invocation of each thread can eliminate data dependence violations. The results of our investigations are presented in the next section.

5. Microbenchmark Analysis Results

In Section 5.1, we first focus on how big a fraction of all of the concurrent thread pairs that succeeds in executing critical sections in parallel in both methods. Then in Section 5.2, we focus on the concurrency exploited by the methods. We end this section by analyzing the relative number of restarts carried out the schemes. This is the theme of Section 5.3.

5.1. Amount of Concurrency in Critical Sections

Recall that the workload we use for our analysis is constructed by a sequence of critical section invocations such that critical section invocations protected by the same lock happen one after each other. In the first experiment, we let two threads execute this program and count the number of critical section invocation pairs that successfully execute the critical sections in parallel. We show the percentage of such fully parallel pairs out of all pairs in Figure 4.

A first observation is that four out of the eight microbenchmarks (Cholesky, FMM, Ocean, and Volrend) have a fair amount of parallelism; more than 40% of all pairs do not encounter any misspeculations as a result of data dependence violations. On the other hand, Barnes and Radiosity exhibit more modest amounts of parallelism and Raytrace and Water seem to have almost no parallelism. A second observation is that SLR consistently manages to discover more concurrency. Overall, SLR manages to let

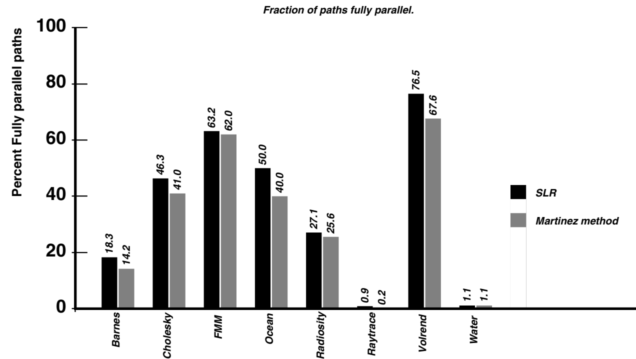

Figure 4. Percentage of thread pairs that are fully parallel using our method and Martínez's method.

about 20% more pairs to execute in parallel compared to Martínez's method.

While measuring the number of thread pairs that manage to execute in parallel provide some insight, the key question is what impact it is expected to have on the execution time of the critical sections. We study this issue next.

5.2. Impact of Concurrency on Execution Time

We now investigate the speedup obtained by executing the threads speculatively in parallel. Since we assume two threads, the maximum speedup we can obtain is two. In Figure 5, we show the execution time of the critical section invocations normalized to the serial execution time for each microbenchmark. For each microbenchmark, the left bar shows the normalized execution time for SLR whereas the right bar shows the normalized execution time for Martínez's method.

An overall observation is that there is a decent speedup in four of the eight microbenchmarks (Cholesky, FMM, Radiosity, and Water). For Radiosity, it is very close to optimum. It is interesting to note that Water enjoys a speedup of about 1.6 despite the fact that only 1% of the execution pairs did not cause any misspeculation as shown in Figure 4. The reason is that these paths are long enough to constitute a major part of the execution. For the other microbenchmarks (Barnes, Ocean, Raytrace, and Volrend), the main reason for the limited speedup is either because of severe load imbalance or because of data dependence violations. In order to understand the exact cause, we have broken down the execution time into four categories: parallel execution time, load imbalance, bypass, and serial/eager restart time.

The first category is the time spent in executing the threads fully in parallel. However, since two threads that run in parallel may execute different critical sections protected by the same lock, they may have different amounts

IEEE COMPUTER SOCIETY

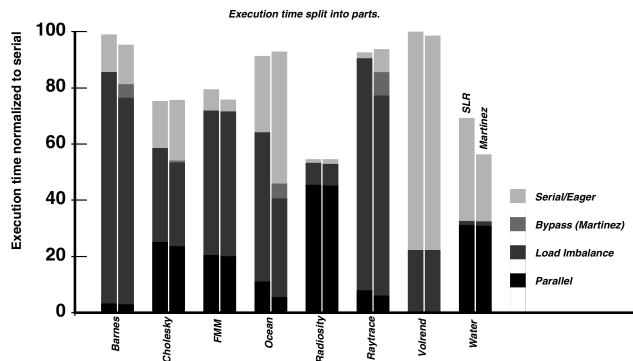

Figure 5. Execution time of the critical sections normalized to the serial execution time.

of work to be done. The one that finishes first will be idle until the second one finishes; it is this idle time that goes into the section denoted load imbalance. Therefore, load imbalance may prevent them from fully exploiting the concurrency. Note that this will not be the case in a real implementation of these schemes since both allow execution to continue after the release point, as noted in section 3. Considering Barnes, the limiting factor is load imbalance. This also explains the poor speedup for Ocean and Raytrace. Yet, these microbenchmarks enjoy some speedup in contrast to Volrend in which there is virtually no concurrency. The load imbalance problem is not the target of the techniques we study. Therefore, let us now focus in detail how the schemes resolve data dependences.

The other two categories used to break down the execution time explain the overhead in resolving data dependence violations. The middle section – bypass – tells us how much of the execution time is spent by thread pairs that resolve data dependence violations using bypassing in Martínez's method. Overall, this feature is not very successful. The last category – serial/eager restart – depicts how big a fraction of the parallel execution time that is used to resolve data dependences by re-executing misspeculated threads. In SLR, a misspeculation results in serial re-execution after the thread has been completed whereas in Martínez's method, the misspeculated thread is aborted at the time the data dependence violation is detected and restarted immediately. Thus, in Martínez's method, it is expected that less work gets lost upon a misspeculation.

Let us now compare how well the two schemes fare as far as resolving data dependences by first considering the four microbenchmarks for which Martínez's method performs better than SLR (Barnes, FMM, Volrend, and Water). Starting with Barnes, both methods spend about the same time on serial re-execution. For this microbenchmark, eager restart does not appear successful but some of the penalty is saved through bypassing. However for FMM, Volrend and Water, the parallel execution times are the same for both methods and the eager restart facility manages to execute the misspeculated threads faster than in SLR. Continuing with the microbenchmarks where SLR does better (Cholesky, Ocean, Radiosity, and Raytrace), it can be clearly seen that the time Martínez's method spent in doing eager restart is considerably longer than the time SLR spends in serially re-executing the threads. When an eager restart takes place it may happen that the thread will experience a data dependence violation again. In contrast, since SLR re-executes a thread when all threads have finished their work, re-executing the thread is guaranteed to succeed. In addition, since restarting a thread may impose a considerable overhead, eagerly restarting threads may offset the gains of speculative execution. We look into this issue more closely next.

5.3. Effects on Number of Misspeculations

Figure 6 shows the number of misspeculations for SLR normalized to Martínez's method. As we can see, SLR consistently suffers from fewer misspeculations than Martínez's method. For three microbenchmarks (Ocean, Radiosity, and Volrend) Martínez's method results in more than twice as many misspeculations. For the other microbenchmarks, Martínez's method results on average in about 15% more misspeculations than SLR. In a chip-multiprocessor, a misspeculation results in flushing the buffered state, restoring the state, and restarting a new thread. All these actions are associated with non-negligible overheads. Thus, the higher number of misspeculations can severely offset the gains of speculative execution of critical sections.

In summary, we have found that there is a fair amount of concurrency that can be exploited in critical sections. SLR manages to expose more such concurrency than Martínez's method. Yet, the speedup is about the same for both systems. This is because Martínez's method sometimes manages to overlap some of the lost work due to data dependence misspeculations by its eager restart mechanism. However, we also showed that the same mechanism can also lead to a much higher number of misspeculations. If we would have taken into account the cost of such misspeculations, the gains of this feature could be outweighed by the overhead in carrying out the eager restart. In addition, we have also seen that the bypassing capability of Martínez's method did not show any significant gains.

6. An Implementation Sketch

Many of the previously proposed thread-level dependence speculation mechanisms for chip-multiprocessors leverage on the cache coherence mechanism to detect data

IEEE COMPUTER SOCIETY

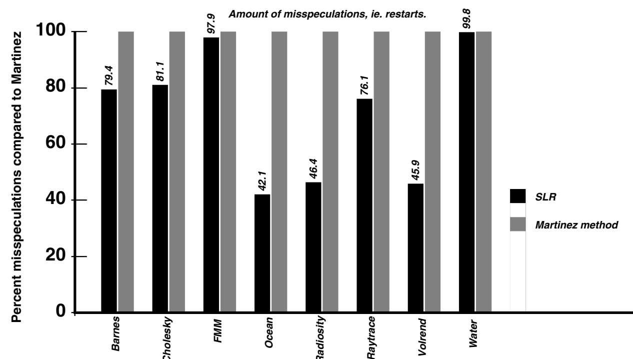

Figure 6. Number of misspeculations for SLR normalized to Martínez's method.

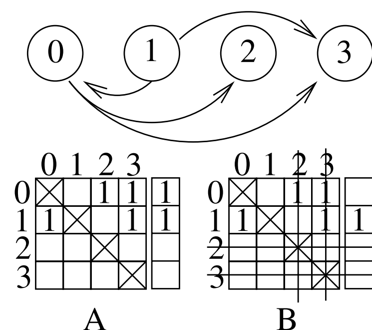

Figure 7. Dependence graph and its corresponding dependence matrix. Matrix A shows that threads 2 and 3 can commit whereas matrix B shows that thread 0 can commit.

dependence violations. We will sketch what additional support is needed to implement SLR in such an infrastructure.

The baseline system we consider is the Stanford Hydra CMP [2] which consists of a number of processors attached to private L1 caches and a shared L2 cache. The L1 data cache for each processor uses a write-through policy to simplify cache coherence. Speculation support consists of extra state associated with the L1 data cache and a number of write buffers between the L1 and L2 caches that are responsible for keeping speculative state. A speculation coprocessor allows control of the speculation hardware from software.

Briefly, a traditional write-through invalidation-based protocol is extended with extra bits associated with each cache line that detects data dependence violations between threads. For example, if a less speculative thread writes to a word that a more speculative thread has read from, a violation is signaled. Additionally, the write buffers are used to keep speculative state for later commitment to memory once the speculative thread is completed.

By contrast, in SLR no a priori execution order between speculative threads is established until all threads are completed. Therefore, the basic speculation support in Hydra is used to detect data dependence conflicts between threads. The basic support must be extended with a mechanism that constructs the thread dependency graph. This mechanism is described next.

The mechanism used to determine the commit order is a unit called the *Commit Order Generator*, or the COG. The COG is connected to each processor listening for dependences found by the speculation extensions to the L1 data cache. When a processor detects a dependence with another processor with the basic support in Hydra, it sends a signal to the COG indicating which processor it has a conflict with. The COG records this information in a structure called the *dependence matrix*. The dependence matrix keeps track of dependencies between processors so that when all threads have exited the critical section a commit order removing the dependencies can be found.

Horizontal rows in the matrix represent producer threads whereas vertical columns represent consumer threads. If processor `n` has produced a value consumed by processor `m` the bit in position `[n,m]` of the matrix will be set. At the time of the commit, the COG will generate a commit mask. The commit mask is a vector of the same length as the number of processors. If a bit in a row of the matrix is set the same bit in the commit mask will be set. All processors without the bit in the commit mask set will be sent a commit signal by the COG to initiate a drain of their write buffers. As the committing processors are now non-speculative they must allow other non-speculative processors to read from their write buffers as long as they are not emptied to the second-level cache. In this way, the commit operation is instantaneous.

As soon as the COG has sent the first commit signals, it removes both the rows and the columns of the committed processors from the dependence matrix. It then generates a new commit mask, sends commit signals to the committable processors, and removes them from the dependence matrix. The COG continues doing this until all processors are committed or a circular dependence is detected. As previously explained in Section 3, a circular dependence requires that one of the involved threads are restarted after all other processors are committed.

At the top of Figure 7 is an example graph of a four processor system. Thread 0 produces data consumed by threads 2 and 3, and thread 1 produces data consumed by threads 0 and 3. The matrices below the graph is the dependence matrix, representing the dependences in the graph. The crossed elements in the diagonal of the matrix are not stored since intra-thread dependences are not able to cause dependence violations. The commit mask is generated by applying a

IEEE COMPUTER SOCIETY

logical OR operation on the elements of each row in the matrix. The empty bits in the commit mask indicate threads that no other threads depend on. In matrix A of Figure 7 we can see that no other thread depends on threads 2 and 3 because their bits in the ORed vector are not set. This means that they can be committed simultaneously as they will not cause any inter-thread dependences. Now that we have committed thread 2 and 3, we can remove arcs pointing to and from these threads from the graph. This is done by removing rows 2 and 3 as well as columns 2 and 3 of the matrix. Matrix B of Figure 7 shows the result of this operation, as well as that thread 0 can now commit since no thread depends on it anymore. Committing thread 0 leaves us with only thread 1, and thus it is the last one to commit.

7. Conclusion

We have introduced the Speculative Lock Reordering Scheme (SLR) to extract more concurrency in critical sections by letting threads speculatively execute them in parallel. A unique feature of SLR is that it exploits the fact that multiple threads can execute a critical section in an arbitrary order as long as modifications inside it are carried out atomically. If threads that speculatively execute the critical section concurrently form a dependency chain that can be represented by an acyclic dependency graph, an execution order that commits threads in the reverse dependency order will convert true data dependences into name dependences and will thus not cause misspeculations.

We carried out a microbenchmark analysis based on the critical sections in applications in the SPLASH-2 suite and found that a fair number of them could be executed in parallel. We also compared the data dependence resolution strategy in SLR with that of previously published lock speculation systems in which the execution order is dictated by the order by which the threads arrive or exit from the critical section. While such methods can resolve data dependences through bypassing and eager restart, these capabilities were shown to only partly compensate for the improved concurrency by SLR. We finally showed that SLR can be supported by thread-level data dependence speculation systems with a quite simple extension. This extension makes decisions on the order by which threads should be committed based on the dependences between them.

Acknowledgments

This research has been supported by a grant from Swedish Research Council on Engineering Sciences (TFR) under contract 221-98-443.

References

[1] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," in *Proc. of 17th Intl. Symp. on Computer Architecture*, pp. 15-26, May 1990.

[2] L. Hammond, M. Willey, and K. Olukotun. "Data Speculation Support for a Chip Multiprocessor", in *Proc. of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 58-69, Oct. 1998.

[3] V. Krishnan and J. Torrellas. "Hardware and Software Support for Speculative Execution of Binaries on a Chip-Multiprocessor," in *Proc. of 1998 Int. Conf. on Supercomputing*, July 1998.

[4] L. Lamport. "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," in *IEEE Trans. on Computers* C-28(9) pp. 690-691. 1979.

[5] K. M. Lepak and M. H. Lipasti. "On the Value Locality of Store Instructions", in *Proc. of the 27th Annual International Symposium of Computer Architecture*, pp. 182-191, June 2000.

[6] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, B. Werner, "SIMICS/sun4m: A Virtual Workstation", in *Proc. of Usenix Annual Technical Conference*, June 1998.

[7] J. Martínez and J. Torrellas. "Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications," in *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002.

[8] R. Rajwar and J. R. Goodman. "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution", in *Proc. of 34th Intl. Symp. on Microarchitecture (MICRO)*, pp. 294-305, Dec. 2001.

[9] R. Rajwar and J. R. Goodman. "Transactional Lock-Free Execution of Lock-Based Programs", in *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct 2002.

[10] P. Rundberg and P. Stenström. "Low-Cost Thread-Level Data Dependence Speculation on Multiprocessors," in *Proc. of the workshop on Multithreading Execution and Compilation at MICRO-33*, pp. 1-9, Dec. 2000.

[11] T. Sato, K. Ohno, and H. Nakashima. "A Mechanism for Speculative Memory Accesses Following Synchronizing Operations," in *Proc. of Intl. Parallel and Distributed Processing Symp. IPDPS00*, pp. 145-154, May 2000.

[12] J. P. Singh, W. D. Weber, and A. Gupta. "SPLASH: Stanford Parallel Applications for Shared Memory", *Computer Architecture News*, 20(1):5-44, March 1992.

[13] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. of 22nd Intl. Symp. on Computer Architecture*, June 1995.

IEEE COMPUTER SOCIETY