Predicting Misspeculations Encountered During Speculative Lock Elision:

Ayswarya "ash" Sundaram and Jason Rickwald
Cal Poly State University
SLO, CA 93405 USA
{ asundara, jrickwal } @calpoly.edu

1. Abstract:

Critical sections cause serialization of threads and thus degrade the performance of parallel processing. Normally, a thread acquires a lock, executes the critical section, and then releases the lock. Meanwhile, other threads wait for the first thread to release the lock so that they can then acquire it and enter the critical section. This serialization may not always be necessary since critical sections can often be safely executed without acquiring any locks. A variety of techniques like Speculative Lock Elision (SLE) [1] and Transactional Lock-free Execution of lock based programs (TLR) [2] have been developed to dynamically remove unnecessary lock-induced serialization; thus enabling concurrent multithreaded execution of critical section. However, these studies face the limitation of increased costs whenever a misspeculation is encountered due to inter-thread data conflicts. We propose Predicting Misspeculations Encountered during Speculative Lock Elision (PME) to address this problem. PME introduces a novel prediction mechanism that predicts the outcome of eliding a lock and thus helps to reduce the number of misspeculations encountered.

2. Introduction:

Locks ensure mutual exclusion of the critical section, which is essential to avoid race conditions. This results in serialization of the execution of the critical section, which often might not be necessary. For example, consider *n* shared processors operating on a shared data structure. If all the *n* processors operate on *n* independent elements of this data structure, then the processors need not have acquired the locks in the first place. Speculative Lock Elision and Transactional Lock-free Execution are two mechanisms that introduce the concept of dynamic removal of unnecessary lock-induced serialization. They thus enable concurrent multi-threaded execution of the critical section.

Speculative Lock Elision is a hardware mechanism that proposes to dynamically elide the locks while observing the shared memory operations. When a processor encounters a lock, it ignores it and continues with the execution of the critical section speculatively. The speculative updates to the memory and registers are managed using the write buffer and the reorder buffer, respectively. When inter-thread data conflicts are encountered, a mis-speculation is triggered. The threads then restart the execution of the critical section, this time explicitly acquiring the locks. Misspeculations in SLE increase the latency inside the critical section, and finally result in the serialization of the critical section.

The TLR concept has been introduced as

a sequel to SLE, and it requires the processors to be able to support SLE. TLR was proposed to tackle the problem of increased latency with regards to critical section when conflicts are encountered. TLR introduces a time-stamp based conflict resolution scheme to enable multiple threads to concurrently execute inside the critical section even in the presence of conflicts. A single globally unique timestamp is attached to all memory requests generated within the critical section. On a conflict, the thread whose transactions have lower priority (the higher value time stamp) will have to restart the execution of critical section while the thread with higher priority (lower value time stamp) can continue with the execution uninterrupted. If a processor receives a request from a lower priority processor for a shared memory location, then the request is buffered. The request is satisfied after the higher priority processor completes the execution of the critical section. On the other hand, if a processor with lower priority receives a similar request from a processor with higher priority, then the request has to be satisfied immediately. As a result, the thread in the lower priority processor will have to restart the execution of the critical section. The value of a processor's time stamp is determined before the execution of the critical section begins, and the same time stamp is used for subsequent re-executions until the execution of the critical section is successful. On successful completion of the execution of the critical section the time stamp is updated. This ensures that no processor will have to wait indefinitely to execute the critical section, and improves forward progress.

A problem with both of the mechanisms described above is that when inter-thread data dependencies are encountered, the cost of rolling back execution or recovering from the conflict outweighs the performance benefit of concurrent execution of the critical section. Misspeculations cause an increase in costs both due to additional resources required to buffer requests for shared data and flushing the speculative data when rolling back execution to a safe state. Synchronization, leading to serialization of access to the critical section, is actually preferable when very frequent data conflicts are encountered. PME has been proposed to address this concern.

PME introduces a prediction mechanism to predict if eliding a particular lock would be beneficial to performance or not. To make the prediction, it uses a history table that has information about the outcomes of eliding the locks in the past. PME implements a simple 1-bit prediction algorithm. PME can be implemented easily since it does not require any complex software or hardware design support. The buffer to maintain history information about the locks can be shared with the existing buffer used for branch prediction.

PME is implemented in hardware, and is thus transparent to software, requiring no extra programming effort. It enables programmers to write correct and fast code with the ease of conservative synchronization (large critical sections). It reduces the burden of the programmers to improve performance.

Key features of PME are:

1) PME complements currently researched speculative techniques that enable concurrent multithreaded

execution of critical section by helping them reduce costs.

2) PME allows easier code development by reducing the burden on the programmer to minimize critical sections.
3) PME is a hardware mechanism that is very simple to implement.

In the next section we will discuss how PME works. Following that, in section 4, we will describe our simulation environment and micro benchmarks used to evaluate PME. Section 5 presents results of our evaluation. Section 6 discusses work related to PME. Finally, in section 7 we go over some future work and improvements to PME, and section 8 concludes.

3. Implementing PME:

We have explained how PME can be used to improve the performance of any speculative mechanism that enables concurrent execution of critical sections. Currently, the implementation of PME supplements the base implementation of SLE with a prediction mechanism. PME's prediction mechanism is similar to a 1-bit branch predictor. The outcome of eliding a lock at time *t* is used to predict the outcome of eliding the lock at time *t+1*. PME exploits the idea that when frequent data dependence is encountered then explicitly acquiring the lock is a better idea than speculation.

We now describe two examples with varying conflicting situations. Example 1 in Figure 1 depicts a relatively lighter conflict situation, whereas Example 2 describes a higher conflict situation. PME aims to avoid unnecessary misspeculations that would be encountered if more than one thread elides the lock at the same time. Some of the implementation aspects of PME include handling speculative register updates, handling speculative memory updates, detecting and recovering from misspeculations and, finally, predicting misspeculations to assist the processor ih deciding whether to speculate past locks or not.

1: light conflict situation

```
lock_acquire operation on shared object
{
conditional update on the shared object
}
lock_release operation on shared object

-----------------------------------------------

```

2: high conflict situation

```
lock_acquire operation on file pointer{
 write to file
 }
 lock_release operation on file pointer
```

Figure 1: Varying conflict situations

Initially, when the processor enters the speculative state its program counter value is saved. When a misspeculation is encountered, then a roll back is executed utilizing this saved program counter value. When a process encounters a conflict while executing through a critical section, then all the other processes currently active in the same critical section are forced to restart the execution of the critical section, and every thread must now explicitly acquire the lock. We incur latency due to the speculative execution and rollback, but it will only happen this once, because the predictor will guard against speculative execution at a later time. In this way, PME can complement any mechanism that utilizes speculation to enable concurrent execution of the critical section.

PME utilizes the Reorder Buffer to manage speculative register updates. Before entering the speculative state the contents of actual registers are copied to a set of speculative registers. While in speculative state all register updates are done to the speculative registers. On successful completion of the execution of the critical section the values of the speculative registers are copied back to the actual registers. On misspeculation the contents of the speculative registers are flushed out.

Every processor also has a local speculative store buffer for memory updates while in a speculative state. The store buffer is modeled as a hash table. Also, when the processor is speculating past locks all the reads are first sourced from this store buffer. Read requests are serviced by the cache if and only if the value is not present in the store buffer. When the processor successfully completes the execution of the critical section without encountering any conflicts, then the contents of the store buffer are committed to the memory system.

For conflict detection, each cache block is augmented with a state bit per processor. When a processor updates or reads a shared memory location, then the state bit for the respective processor is set. When any processor attempts to access a shared memory location, then the state bits of that cache block are first checked. If the state bit indicates that the corresponding cache block has been viewed or modified by another processor, then the essential condition of atomicity has been violated. Atomicity guarantees that updates done by a thread in a critical section must be readable by other threads only when the said thread completes execution of the critical section. Hence, when atomicity is violated, then misspeculations are generated forcing all the threads active in the corresponding critical section to restart. The state bits in the cache are flushed for a given processor when it commits at the end of a critical section or when it is forced to restart.

A history table has been included in the interrupt trap handler routine to maintain information about the outcomes of eliding the respective locks. The table is implemented in the interrupt trap handler routine since, in the simulator we use, the lock acquires and releases are modeled as system calls. Our work does not currently address resource limitations, so our history table has been modeled as a linked list. The table contains a mapping of the processor number, program counter of the instruction that causes lock acquire, and the past result of eliding the lock. A simplification made here is to require the program counters of different lock acquire operations to be distinct. When the interrupt handler routine receives a system call for lock acquire it looks up the corresponding mapping in the history table. If a mapping is not present, then a new entry is added to the table and the lock is speculatively elided. If an entry is present, then past result is checked. A past conflict causes the lock to be acquired in the normal way. In contrast, a past result indicating a success causes the lock to be elided speculatively. Though this concept can be applied to nested locks, they are currently not addressed here. PME thus helps to reduce repeated misspeculations in a high conflict situation.

4. Evaluation Methodology:

4.1. Simulation Environment:

We use PolyScalar, a modular simulator based on SimpleScalar for running multithreaded binaries. PolyScalar enables the easy creation of a multi-programmed simulator due to its object-oriented code. Our simulator models multiple out-of-order processors. All the processors share a common L1 data cache and L2 cache. All processors have independent L1 instruction caches. The processors have independent store buffers to hold the speculative updates to memory. The reorder buffer is used to handle speculative register updates. Each cache block is augmented with state bits to detect misspeculations.

4.2 Micro-benchmarks:

We evaluate PME using a simple set of 4 micro-benchmarks. The primary goal is to capture two different types of conflict situations [2]:

1) No-conflicts: The threads speculating past locks do not encounter any misspeculations.

2) High-conflicts: The threads speculating past locks almost always encounter misspeculations.

Performance of benchmark1 is unaffected by PME. Benchmark1 does not give rise to any conflicts and hence no misspeculations are encountered. Though benchmark2 gives rise to a high conflict situation, each of those conflicts are caused by eliding different locks. A particular lock is not encountered more than once by a processor. PME does not make predictions for the first occurrence of a lock but, rather, uses the information about the result of eliding the lock in the past to make a prediction. Hence, the performance of benchmark2 is not affected by PME. PME however, helps to improve the performance of benchmark3. It helps to reduce the repeated generation of misspeculations, thus bringing down the execution cycle count.

	Features	Conflict Situation
benchmark1	The processors increment independent counters protected by a common lock – a no conflict situation.	None
benchmark2	The processors repeatedly make the same function call where the same lock is used to protect updates to a shared memory region.	High
benchmark3	The processors repeatedly make different function calls where different locks are used to protect updates to shared memory regions.	High
benchmark4	The processors repeatedly make different function calls to the same functions where the same locks are used to protect updates to a shared memory region, but at different times.	Varies

Table 2: Micro-benchmarks

Processor	32 general registers, 32 floating point registers, instruction window size 128, instruction fetch queue size 8, decode width 8, issue width 8, load store queue size 64, 4 ALUs, 1 multiplier, ROB of size 64, out of order issue.
L1 data cache	Shared data cache with number of ports equal to number of processors, line size 64, bank size 512, associativity 10, write buffer size 10, request latency 0, response latency 0, coherence policy –update, write policy –write around.
L1 instruction cache	Independent instruction caches, line size 64, bank size 512, request latency 0, response latency 0, coherence policy –update, write policy –update.
L2 cache	Shared, line size 64, bank size 4096, associativity 4, write buffer size 10, request latency 1, response latency 1, coherence policy –invalidate, write policy –write allocate.
DRAM	Access latency 60, transfer rate 10.

Table 1: Simulated Machine Parameters

5. Results:

PME contributed to significant performance improvements in bench-marks where repeated misspeculations are encountered.

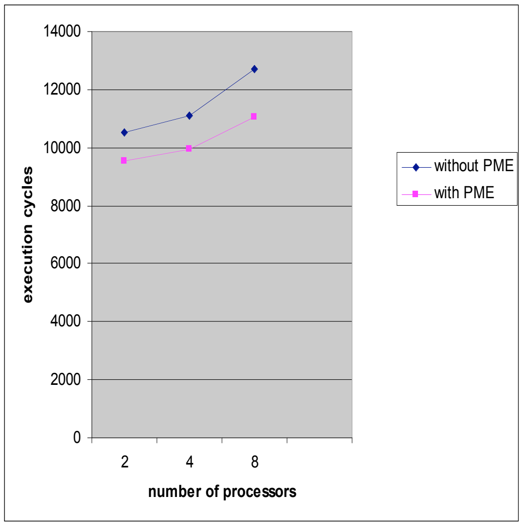

Figure2: Micro-benchmark result

Figure 2 plots execution cycles of benchmark 3 (described in Table 2) on the y-axis with number of processors on the x-axis. The performance degrades as the number of processors increase since this benchmark leads to a conflict situation every time a lock is elided, thus serializing the execution of the critical section. PME improves performance by reducing the execution cycles that would have otherwise been wasted on recovering from misspeculations every time a conflict was encountered.

Figure 3 depicts the percentage of the critical section in the benchmarks listed in Table 2.

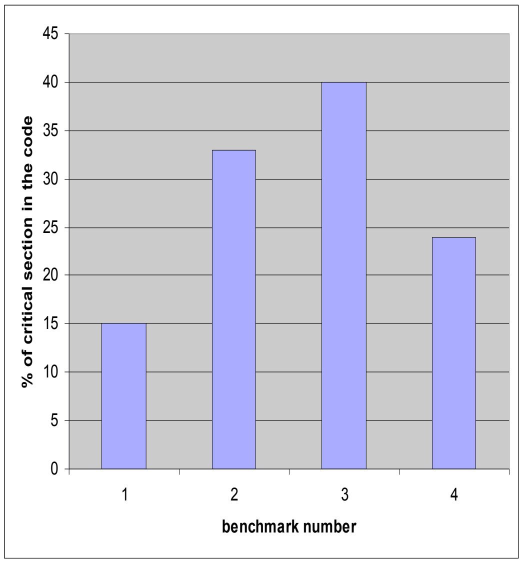

Figure3: Percentage of critical section

6. Related Work:

Speculative Lock Elision introduced a hardware technique that allows for a processor to dynamically elide a lock [1] and hence allowed multiple threads to concurrently execute inside a critical section. The motivation for this was that locking is often used to guard against unusual interactions between threads that share data that do not typically happen during dynamic execution, and the serialization introduced by the locks can make critical sections into bottlenecks. SLE locates locks in a program without the aid of the software by looking for typical lock acquire and lock release code sequences. A processor that encounters a lock will checkpoint its current state, then execute past the lock speculatively. Conflicts between threads are caught using existing cache coherency mechanisms.

Techniques that speculatively execute inside the critical section often lead to an increase in the latency inside critical section. TLR was introduced to address this problem. TLR allowed the earliest conflicting thread to continue with the execution inside the critical section while the others were either required to either wait or restart [2]. TLR improves SLE by preventing an increase in the latency inside critical section when misspeculations are encountered, thus enabling it to make better forward progress.

Thread Level Speculation (TLS) [4] forms another research area that is different from the ones mentioned above. It is a hardware technique that breaks a single thread of execution into multiple parallel threads in order to further exploit the performance benefits of parallelization in multi-core or multi-processor machines. TLS however, requires the use of value prediction and the placement of split points in a program, both of which are complex features that can have an adverse effect on program execution time if done poorly.

Speculative Synchronization is a technique applying TLS concepts to explicitly parallel applications [3], which is highly relevant to our work. Speculative Synchronization does not require the use of complex features like value prediction and split points. In Speculative Synchronization, a thread initially tries to acquire a lock. If it succeeds, then it continues with the execution and is considered to be "safe," and it will not ever be forced to rollback to the beginning of the critical section. On the other hand, if it cannot acquire the lock, then it speculatively executes past the lock. If a dependence violation is encountered during speculative execution, then it is forced to roll back to the start of the critical section. If no such violation is encountered and the thread successfully reaches the end of the critical section, then it is required to wait to become safe before it is allowed to commit. Hardware support is required to allow threads to spin on the lock to become safe while simultaneously speculatively executing past the locks. Moreover, the implementation of Speculative Synchronization requires some software support to utilize the hardware.

Compiler Optimization of Scalar Value Communication between Optimized Threads (COSV) introduced the concepts of conservative scheduling and aggressive scheduling [5] to reduce the number of instructions inside the critical section and hence the overall impact of serialization of the critical section.

Though the above proposals regarding speculative execution of critical sections are conceptually very powerful, they face the limitation of increased costs in terms of critical time, work, and power whenever misspeculations are encountered. PME reduces the number of misspeculations encountered and can thus bring down the costs incurred. PME is technique that can complement any of the speculative techniques discussed above and can help them enhance their performance.

7. Future Work:

As a next step we aim to evaluate the performance of PME with more complex Spec benchmarks. Also, the current prediction algorithm used by PME has a limitation that once a conflict is encountered while eliding a lock, then any time in the future, if the same lock is encountered, then a conflict situation is assumed. This may not necessarily be the case. The next step is to improve the prediction algorithm by evaluating the applicability of other predictor types and techniques. Predictors that take prior control flow information into account, such as global history predictors [6], may do a better job of predicting the likelihood of inter-thread dependence violations. The execution of the thread leading up to the critical section may have a strong correlation with the behavior of the thread execution inside the critical section.

8. Concluding Remarks:

We have proposed a hardware-based predictor as a technique to improve the performance of any speculative technique that enables concurrent multithreaded execution of the critical section. This technique is very simple to implement. It does not require any instruction level support or any involvement of the programmers. The key insight is that misspeculations waste processor cycles and power and should be avoided whenever possible. PME can reduce the number of repeated unnecessary misspeculations that would have otherwise been caused. Thus, PME reduces costs associated with concurrent multithreaded execution and thereby helps to improve performance.

Acknowledgements:

We would like to especially thank Dr. Diana Franklin for guiding us through the implementation and for extensive discussions to answer our questions.

References:

[1] Ravi Rajwar, and James R. Goodman: Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *proceedings of 10th Internatational Confrence on ASPLOS*, Oct. 6- Oct. 9, 2002, San Jose, California, USA.

[2] Ravi Rajwar, and James R. Goodman: Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *proceedings of 34th National Symposium on Microarchitecture*, Dec. 3- Dec. 5, 2001, Austin, Texas, USA.

[3] José Martínez, and Josep Torrellas: Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems*, 10/02, San Jose, CA, USA.

[4] J. Greggory Steffan, Christopher Colohan, Antonia Zhai, and Todd Mowry: A Scalable Approach to Thread-Level Speculation. In *proceedings of the 27th annual international symposium on computer architecture*, 2000, Vancouver, British Colombia, Canada.

[5] Antonia Zhai, Christopher B. Colohan, J.Gregory Steffan and Todd C. Mowry: Compiler Optimization of Scalar Value Communication between Speculative Threads. In *proceedings of the 10th international conference on Architectural Support for Programming Languages*, 10/02 San Jose, CA, USA.

[6] Daniel Jiménez: Reconsidering Complex Branch Predictors. In *proceedings of the 9th international symposium on high-performance computer architecture*, 2/03.