Detection of Synchronization Errors through Speculative Lock Elision

Steven S. Lumetta
University of Illinois at Urbana-Champaign, ECE and CSL, lumetta@uiuc.edu

The idea of executing sections of code optimistically in parallel in order to increase performance has recently regained popularity in the computer architecture literature. Two papers [6, 7] suggest the speculative elision of lock acquisitions in threaded programs to eliminate unnecessary serialization and to allow the use of more coarse-grained locks without sacrificing performance. These schemes rely primarily on extensions to processor caching mechanisms, an idea introduced in [4].

This paper suggests a minor extension to such a protocol [7] to enable production-time detection and localization of synchronization errors in threaded applications. Although the idea may also apply to some earlier techniques [3], its value has not been previously recognized. The closest related work in terms of detecting synchronization errors is Eraser [8], which seems more likely to detect errors, but slows applications down by about a factor of ten, and is thus not practical outside of the application development environment. Similar techniques have also been proposed for data race detection with release consistency [2].

Background: Databases require concurrent execution of transactions to achieve high performance, but transactions must be executed in a mutually atomic fashion to avoid introducing inconsistencies in the form of non-serializable data dependencies between transactions. The need for transaction rollback in the face of deadlock led to the view of a transaction as a lock acquisition phase followed by a lock release phase [1]. Later work, which coined the term "optimistic concurrency" [5], proposed a three-phase view consisting of reads, validation of reads, and writes. The latter two phases in this model must be performed atomically, with execution of the write phase (commit) being predicated on successful completion of the validation phase. Notions of read and write sets were introduced to track accesses by each transaction for use in detecting conflicts.

The application of processor caching mechanisms to optimistic concurrency first appeared in the context of automatic parallelization of LISP code [4]. The sequential code was split into a totally ordered set of fragments, with each fragment containing only a few writes to memory. Unlike database transactions, such code fragments must obey the ordering implied by sequential execution. Two caches associated with each processor held the read and write sets for currently executing fragments. When a fragment completed in sequential order, its write set was broadcast to other processors across a bus, restarting later fragments that had used any previous value optimistically.

 NSF grants ACI 99-84492 CAREER and ANI 01-21662 ITR supported part of this work.

One of the first computer architecture schemes to support explicitly parallel transactions was Transactional Memory [3], which extended the instruction set to support transaction semantics. New instructions included transactional reads and writes, a validate instruction to confirm the continued accuracy of all transactional reads, and a commit instruction to validate and complete a transaction when possible. The paper also outlined a possible implementation, which for practical reasons added a second cache to hold the read and write sets apart from the normal cache data. The design also extended the cache coherence protocol to convey the difference between transactional and non-transactional accesses, allowing a processor to restart a remote transaction by refusing to share transaction data.

Speculative Lock Elision: Many modern, high-performance processors support speculative execution of instructions in hardware to enable branch prediction. Recently, two papers [6, 7] suggested the use of this speculative mode to support optimistic execution of critical sections protected by locks, a technique termed speculative lock elision, or SLE. We build on [7], which allows execution to pass acquisition of an unowned lock without actually claiming the lock by placing the processor into speculative execution mode. The approach is logically equivalent to that used with Transactional Memory [3], but uses hardware speculation for rollback and restart rather than explicit software control.

The support of mutual exclusion (lock) semantics rather than atomicity (transaction) semantics has implications for both programmers and for the system. Atomicity is more attractive than mutual exclusion from a programming point of view, as atomicity is a more local phenomenon that need not be rethought—or, in many cases, even understood—by programmers extending an application. In contrast, use of mutual exclusion must be managed carefully to avoid deadlock, and the strategy used for each abstraction must generally be understood by every programmer that extends or builds upon the abstraction. Outside of the database community, however, mutual exclusion is more common in practice because locks are more widely and directly supported by both hardware and software than are transactions.

From a hardware implementor's point of view, SLE offers advantages over optimistic transactions. With SLE, a natural backoff mechanism is available to eliminate the possibility of starvation. In particular, repeated failures of speculative elision cause the processor to acquire the lock non-speculatively [7], guaranteeing forward progress for correct

programs. This benefit is also a drawback, of course, in that the responsibility for understanding the need for synchronization between all critical sections in the code is forced back onto the programmer.

SLE Fault Tolerance: The complexity of applying mutual exclusion also implies that lock-based synchronization in threaded programs is likely to contain errors. An application may not acquire the proper set of locks before manipulating a datum. Many executions of the application may succeed, with the fault lying dormant for years, and rare system failures caused by this fault may be nearly impossible to trace back to the fault itself.

The mechanisms suggested in [7] do provide some protection against such errors through a transaction-like implementation. In particular, critical sections incorrectly protected by different locks are executed in a mutually atomic fashion despite this fault, as termination of speculation is based on detection of potential conflicts, not on the locks chosen to protect against them. Similarly, speculatively executed critical sections are restarted when any code not protected by a lock conflicts with atomic execution. If the processor backs off to non-speculative lock acquisition, however, the faults may still lead to failures.

Detecting and Reporting Errors: We now describe a mechanism that allows errors arising from synchronization faults to be detected and the two conflicting sections of code to be identified. We first introduce an intermediate backoff stage in which a lock is acquired, and the lock acquisition is made visible to other processors, but the execution remains speculative. The lock value is the only speculatively written data made visible to other processors; should speculation be terminated after this value is shared, the processor reacquires exclusive ownership of the lock cache line and releases the speculatively acquired lock. As the semantics of the resulting operation are slightly different than that of the usual synchronization primitive, a new instruction should be added to differentiate the two.

Once a lock is acquired, potentially conflicting accesses (other than those to the lock itself) can only be made from sections of code not protected by the lock. Speculative mode is necessary for tracking the critical section's read and write sets, which allows conflicts to be differentiated from accesses to data cached prior to the critical section. When a conflicting access is detected, speculative state is committed (the processor owns the lock) and an exception generated, which is then delivered to the application in the form of a signal, allowing production-time identification of the offending program state. In order to identify the second code section, we can follow [3] in extending the cache coherence protocol to allow a processor to refuse to provide data to any coherence transaction that causes a synchronization exception. Receipt of such a response generates an exception at the appropriate instruction in the second section of code.

False negatives are only possible with the mechanism just outlined if both sections of code are completely unsynchronized. Unfortunately, speculation can also terminate due to resource limitations, in which case the processor must commit the speculative data and sacrifice its error detection capabilities.

False positives can also occur, as unsynchronized accesses may be intended. Consider a concurrent FIFO queue based on a cyclic array with head and tail indices (see [8] for examples from applications). Let a single thread write to the queue and the tail index, while multiple threads read from the queue under the protection of a lock. When trying to dequeue, readers obtain the lock and read both the head and the tail indices. The writer, however, need not obtain the lock before writing the tail index during an enqueue operation, leading to a false positive with our mechanism. An analogous false positive is found in the case of multiple writers with a single reader. False positives of this form are fairly unlikely in code written in high-level languages such as Java, although libraries may provide more sophisticated synchronization. False positives can be suppressed in software at the expense of an occasional signal, or the instruction set and cache coherence protocol can be extended to allow suppression of exceptions when desired.

An additional benefit of our approach is that processor interrupts, such as those used to transfer control to the operating system scheduler for time sharing, end speculation and release the lock, thereby extending the non-blocking behavior of SLE to times of high contention.

The coherence protocol distinction between transactional and non-transactional accesses in Transactional Memory [3] can be extended to support fault detection and reporting in a manner similar to that suggested here, but no mention of this capability appears in the paper.

References

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–33, 1976.

[2] K. Gharachorloo and P. B. Gibbons. Detecting Violations of Sequential Consistency. In *Symposium on Parallel Algorithms and Architectures*, pages 316–26, 1991.

[3] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *International Symposium on Computer Architecture*, pages 289–300, 1993.

[4] T. Knight. An Architecture for Mostly Functional Languages. In *ACM Conf. on LISP and Functional Programming*, pages 105–12, 1986.

[5] H. T. Kung and J. T. Robinson. On Optimistic Methods for Concurrency Control. *Transactions on Database Systems*, 6(2):213–26, 1981.

[6] J. F. Martínez and J. Torrellas. Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. In *Workshop on Memory Performance Issues*, 2001.

[7] R. Rajwar and J. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *International Symposium on Microarchitecture*, pages 294–305, 2001.

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In *Symposium on Operating Systems Principles*, pages 27–37, 1997.