Atomic Transactional Execution in Hardware: A New High-Performance Abstraction for Databases?

Ravi Rajwar
Intel Labs-MRL
ravi.rajwar@intel.com

Philip A. Bernstein
Microsoft Research
philbe@microsoft.com

April 9, 2003

1.0 Introduction

Advances in hardware technology have deep implications on future database system designs [1]. Increasing transistor densities, faster processors, larger memories, and low inter-chip communication latencies have enabled new hardware mechanisms such as processors supporting speculative execution, large on-chip buffering, and aggressive multiprocessor system organizations with cache coherence. As a result, much of the support required to implement hardware transactions is now either present in modern processors or their implementations are well understood. This paper discusses one such proposal. It is based on a hardware mechanism called Transactional Lock Removal [2] (TLR), which was originally designed to support the atomic execution of critical sections by a lock-based multithreaded program in a lock-free manner. In this paper, we explain the mechanism and suggest how it could be used to control the atomic execution of transactions in a database system.

The TLR hardware identifies, at runtime, lock-protected critical sections in the program and executes these sections without acquiring the lock. TLR maintains correct semantics of the program in the absence of locks by executing and committing all operations in the now lock-free critical section "atomically". Any updates performed during the critical section execution are locally buffered in processor caches. They are made visible to other threads instantaneously at the end of the critical section. By not acquiring locks, the hardware can extract inherent parallelism in the program independent of locking granularity.

TLR embodies a core transaction mechanism: a notion of an "all-or-nothing serializable execution of a sequence of operations." The ability to implement transactions in hardware could have major implications for future database systems. In section 2, we discuss the new hardware abstraction and in section 3 we discuss how to apply it to database transactions.

2.0 Critical section execution: A new hardware abstraction

2.1 Critical section semantics

A critical section is associated with a named lock. The goal is to execute the section so that only one thread can execute it at a time and the thread’s updates made during a critical section are only made visible to other threads at the end of the section. For lack of a better term, we'll call this an atomic execution (strictly speaking, this covers atomicity and isolation in ACID terminology). To attain atomic execution, the programming convention is that a thread should bracket a critical section by acquire_lock(LOCK) and release_lock(LOCK) operations, on the LOCK that protects the section. The semantics of acquire_lock is that only one thread at a time can own the lock and a thread waiting for the lock will acquire the lock only after all updates made by previous executions of the critical section have been made visible to all threads.

Clearly, the acquire_lock operation requires that a programmer decide when and what to lock. In principle, the programmer also decides the granularity at which to lock, and the locking protocol to avoid deadlocks. Since locks enforce a serial execution of critical sections by different threads, performance can degrade substantially if these decisions are not made well. Further, breaking deadlocks requires detecting them and unrolling any updates performed to memory to achieve the all-or-nothing behavior. These are non-trivial actions. The proposed mechanism automates most of these decisions.

2.2 The TLR mechanism

The TLR hardware treats a lock-based critical section as a lock-free optimistic critical section. It uses a speculative lock elision technique to elide the named lock [3] and allows multiple threads to enter and execute the same critical section speculatively. The lock is read in a shared state and monitored, but ordinarily is not written. The execution of a critical section by a thread is assigned a single globally unique timestamp. All memory operations generated by that execution are associated with that timestamp. Processor cache structures are used to locally buffer any memory updates performed within the optimistic critical section. Speculatively buffered data is never made visible to other threads. Hardware cache coherence mechanisms are already present to ensure copies of a cache block in various processor caches are kept consistent. TLR uses these mechanisms to determine if any data blocks accessed within the optimistic critical sections experienced a conflict.

A conflict occurs if two or more threads access a given data block and at least one of those threads performs a write operation to the block. Timestamps are used to resolve conflicts. The thread with the smallest (earliest) timestamp wins, and the other conflicting threads restart by employing hardware re-execution mechanisms. For these restarting threads, the same timestamp determined at the beginning of their optimistic critical section is retained for subsequent re-executions until the critical section is successfully completed. This guarantees each thread will eventually win any conflict by virtue of having the earliest timestamp in the system and thus succeed in executing its optimistic lock-free critical section. The thread which wins all conflicts and completes its critical section makes all of its updates visible to other threads instantaneously at the end of the critical section.

If the data accessed cannot be locally buffered (i.e. a cache overflow occurs), the elided lock is explicitly acquired by the processor that experiences the overflow. When the lock is acquired, the coherence protocol automatically broadcasts the write of the lock to all other speculating processors that elided this lock. Since each critical section execution reads the lock on entry, subsequent attempts to enter the critical section will be delayed until the processor that wrote the lock flushes its cache at the end of its critical section. Thus, the critical section is not executed lock-free in this case.

If updates in the optimistic critical section can be locally buffered, all non-conflicting critical sections execute and complete concurrently, without being serialized on the lock because the lock is never acquired. Critical sections experiencing data conflicts are also executed without acquiring locks and without interfering with non-conflicting critical sections. Importantly, the decision of when to serialize execution is based on the accessed data experiencing a conflict, rather than a conflict on the named lock.

If anything goes wrong while the critical section is executing, any updates locally buffered are discarded thus providing the all-or-nothing behavior automatically.

While the mechanism sounds complex, much hardware required to implement it is already present in systems today. The ability to recover to an earlier point in an execution and re-execute is used in modern processors and can be performed very fast. Caches retain local copies of memory blocks for fast access and thus can be used to buffer local updates. Cache coherence protocols allow threads to obtain cache blocks containing data in either shared state for reading or exclusive state for writing. They also have the ability to upgrade the cache block from a shared state to an exclusive state if the thread intends to write the block. The protocol also ensures all shared copies of a block are kept consistent. A write on a block by any processor is broadcast to other processors with cached copies of the block. Similarly, a processor with an exclusive copy of the block responds to any future requests from other processors for the block. The coherence protocols serve as a distributed conflict detection and resolution mechanism and can be viewed as a giant distributed conflict manager. Coherence protocols also provide the ability for processors to retain exclusive ownership of cache blocks for some time until the critical section completes. A deadlock avoidance protocol in hardware prevents various threads from deadlocking while accessing these various cache blocks.

2.3 TLR discussion

Since the decision of when to serialize execution is now based on the accessed data experiencing a conflict rather than on the named lock, the programmer can use coarse-grain locking instead of fine-grain locking. Even though with coarse-grain locking one lock protects a large number of data blocks, execution is serialized only for threads with conflicting access to a data block. This behavior is the same as if the programmer had used a lock for each data block. By allowing the use of coarse-grain locks, deadlock issues become trivial because a thread can acquire just one coarse-grain lock instead of multiple fine-grain locks. Thus, deciding on the granularity at which to lock, the performance impact of when and what to lock, and the deadlock avoidance strategy are now shifted from the programmer to the hardware.

To avoid the explicit acquisition of a lock, it is important to minimize the frequency of cache overflows. Hardware advances help here. By profiling common critical sections, sufficient resources can be provided in the processor to capture a large fraction of these critical sections. As an example, if a processor had a 64-entry victim cache [4] for a 64-byte cache block in addition to its local cache hierarchy, in the worst case the processor can buffer a little over 4 kilobytes of data and in the best case the size of the local cache (which may be a few megabytes).

If a non-cacheable operation occurs, TLR explicitly acquires the lock and falls back to a lock-based execution. In this case, explicit deadlock avoidance is needed. For example, one could prohibit nested critical sections, so that a thread never requests a lock while it owns a lock.

The number of instructions executed within the critical section can be made arbitrarily large because TLR requires a subset of the mechanisms required for speculative execution--namely only the ability to restart execution from a single point in the past. Conceptually, at the start of the critical section, the processor creates a checkpoint of its architectural state (a fast operation in modern processors) and restores this checkpoint if required.

3.0 The TLR abstraction and database transactions

A strong relation exists between database concurrency control mechanisms and TLR. Conceptually, TLR is coordinating accesses to cache blocks by using the cache coherence protocol. In effect, it runs each critical section as a transaction using optimistic locking and the wound-wait algorithm [5], with explicit locking as a “backup” strategy in the event of cache overflow.

This is similar to database transactions performing concurrency control by acquiring appropriate locks for data accessed within the transactions. In most database systems, these locks are explicitly stored and associated with the data accessed, and the database software detects deadlocks and breaks them by aborting transactions. However, with TLR, the effect of these locks is automatically captured by virtue of the coherence state of the data block itself. A data cache block in shared or exclusive state is similar to a data block protected by a shared or exclusive lock. However, this information is "implicit" under TLR, not "explicit" as it is in database systems. Given the similarity in concept, it’s not surprising that the mechanisms for concurrency control are also very similar.

We now discuss TLR’s atomic lock-free critical section abstraction in terms of the ACID properties of database transactions.

Atomicity: A transaction must either execute to completion or have no effects at all. TLR attains this by buffering updates performed within the lock-free critical section and writing them to memory only if the lock-free critical section completes. If the critical section fails, the updates are discarded.

Consistency: A transaction must preserve consistency of shared data. This is a property of the transaction and not of the mechanism that implements it.

Isolation: Transactions must execute serializably. TLR achieves this by aborting a transaction whenever it experiences a conflict, employing a wound-wait-type algorithm in hardware. Thus, in effect transactions run serially.

Durability: Durability requires that results of transactions having successfully completed are committed to storage. The transaction state must be restored in the event of any type of failure. TLR does not handle disk writes and thus does not provide durability.

TLR's atomic lock-free critical section provides ACI of the ACID properties for transactions. Thus, TLR's abstraction may provide a promising fit for high-performance database operations for main memory databases. If durability is important, then a log could be introduced as another main memory structure that is written by each transaction and spooled to disk.

Using TLR as a database transaction mechanism significantly reduces locking overhead compared to the standard database approach. Despite its use of coarse-grain locks, TLR provides a similar effect as a system using fine-grain locking because the decision to serialize execution is based on the actual data conflicts, not lock conflicts. The coarse-grain lock is used only to specify the scope of the transaction and the lock itself does not serialize any execution. TLR does not require fine-grain locks to be explicitly specified in the software because a lock is "implicit", as discussed above, with each cached data block accessed by the transaction. As in database transaction mechanisms, these "implicit" locks are associated with the data only for the duration of the transaction.

Direct benefits of the above include reduced space requirements for locks, and simpler locking protocols. By not actually requiring explicit fine-grain locks to achieve their behavior, the memory footprint of locks is significantly reduced. Further, since performance is not impacted by the use of coarse-grain locks, the complex locking hierarchies required for manipulating fine-grain locks are not necessary. At times, TLR with coarse-grain locking can outperform systems with fine-grain locking because the data memory footprint of the transaction is much smaller primarily due to the absence of explicit fine-grain locks in software. This translates to reduced traffic in the hardware memory system.

TLR automatically performs concurrency control in hardware using the cache coherence protocol. Thus, the database system designer does not have to worry about deadlocks due to incorrect manipulation of fine-grain locks. TLR executes a deadlock-free concurrency control algorithm on the data blocks being accessed in the transaction. By doing most concurrency control operations directly in hardware and avoiding software overhead, transactions can execute faster than with database locking.

The atomic transaction abstraction provided by TLR is a powerful primitive for constructing richer operations in software while allowing the continued use of critical sections as a programming model but without the limitations of critical sections. Can databases exploit the new abstraction of atomic transactional execution? If so, this would provide an added incentive to hardware manufacturers to support such a TLR mechanism. If not, then it would be valuable to understand why, to determine whether a variation on the TLR mechanism would provide the synchronization behavior and performance required by database transactions.

Acknowledgement

The TLR work was done by the first author and Prof. James R. Goodman at the University of Wisconsin-Madison.

References

[1] P. Bernstein, M. Brodie, S. Ceri, et al, The Asilomar Report on Database Research, SIGMOD Record, Vol 27. No 4, 1998.
[2] R. Rajwar and JR. Goodman, Transactional Lock Free Execution of Lock-Based Programs, 10th International Symposium on Architectural Support for Programming Languages and Operating Systems, 2002.
[3] R. Rajwar and J.R. Goodman, Speculative Lock Elision: Enabling Highly-Concurrent Multithreaded Execution, 34th International Symposium on Microarchitecture, 2001.
[4] N.P. Jouppi, Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, 17th International Symposium on Computer Architetcure, 1990.
[5] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, System Level Concurrency Control for Distributed Database Systems, ACM Transactions on Database Systems,1978.