The Software Stack for Transactional Memory

Challenges and Opportunities

Brian D. Carlstrom JaeWoong Chung
Christos Kozyrakis Kunle Olukotun

Computer Systems Laboratory
Stanford University
{bdc, jwchung, kozyraki, kunle}@stanford.edu

Abstract

Transactional memory systems apply the experience of the database community to the general problem of parallel programming with the goal of providing a simple parallel programing model that delivers on the performance potential of multi-processor systems. Although initial research into both software-only and hardware-supported transactional memory has shown promising results, there are many challenges to creating a fully transactional software stack. Although today's software stack has some limited use of transactional programming, many parts of the stack from basic data structures to the operating system and program runtimes contain at least some lock-based code. In code with coarse-grained locking, transactions provide an opportunity to improve performance. In code with fine-grained lock, transactions provide an opportunity to simplify code while reducing overhead.

1. Introduction

In 1978 C.A.R Hoare wrote, "developments of processor technology suggest that a multiprocessor machine [...] may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor" [11]. Over a quarter of a century later, even laptop computers are multiprocessors thanks to the arrival of multi-core processors. Although there has been progress in the intervening decades, there is still debate about how to best create software to take advantage of multiprocessor systems.

The most widely used mechanism for explicitly managing parallelism in programs is locks, whether it be through Pthread mutexes in C or C++ or synchronizing on objects in Java. However, the traditional problems of locks such as priority inversion, convoying, and deadlock, have led to the development of alternatives such as non-blocking data structures as well as the use of ACID transactions.

Although locks may be the most common method for managing parallelism explicitly, more programmers probably use parallel systems implicitly through the use of transactional database systems. We believe that there is much to be gained from the transactional alternative to locking. We believe that transactions make it easier to write correct parallel programs and, just as important, to write parallel programs that have good performance. Our beliefs are not just based on our own experience with the programming model provided by transactional memory, but also on the experience of the database community. To give a seasoned opinion from that community, here is a quote from Jim Gray:

> There are many examples of systems that tried and failed to implement fault-tolerant or distributed computations using ad hoc techniques rather than a transaction concept. Subsequently, some of these systems were successfully implemented using transaction techniques. After the fact, the implementers confessed that they simply hit a complexity barrier and could not debug the ad hoc system without a simple unifying concept to deal with exceptions [Borr 1986; Hagmann 1987]. Perhaps even more surprising, the subsequent transaction oriented systems had better performance than the ad hoc incorrect systems, because transaction logging tends to convert many small messages and random disk inputs and outputs (I/Os) into a few larger messages and sequential disk I/Os. [9]

While this paper does not focus exclusively on hardware transactional memory, the last sentence of this quote has particular relevance to multi-core processors, where exchanging many small, latency sensitive messages for fewer, larger messages can be advantageous, as previously demonstrated [13].

Despite their short comings, locks can be found supporting parallelism in all layers of today's software stack including basic data structure libraries, operating systems, databases, programming languages, and distributed systems. This creates opportunities for transactions to improve parallel performance while reducing software complexity throughout the stack. However, with these opportunities come challenges, particularly in handling I/O and other operations that don't always fit the transactional semantics, which are common in system software.

2. Data Structures

Parallel access to data structures using locks is a trade off between simplicity and efficiency. Usually a programmer will start with a single lock to protect a structure, which can lead to idle time due to contention. A programmer can then move to finer grained locks, with the cost of implementation complexity, increasing the likelihood of deadlock.

There has been work on non-blocking data structures to avoid using locks and the common problems they lead to such as deadlocks, convoying, and priority inversion. However, these data structures can be more complex than even their fine-grained locking counterparts.

STMCS '06 March 26, 2006, Manhattan, New York, NY

Transactional memory allows simple lock-free data structures to be used with concurrent access without performance impact. However, similar to non-blocking data structures, contention can still be a problem. For example, a simple hash table will have little contention even on parallel update. However, if the table tries to maintain the current number of elements, it creates a source of contention for any update that adds or removes elements, which for most uses is the common case.

A few extensions to the basic transactional semantics have been proposed to handle such cases [12]. The concept of open nested transaction can help by minimizing the impact of contention, by only rolling back the update of element counter and not the entire transaction. Another approach is to use violation handlers to respond to contention in a data structure specific way; in the case of the counter, the transaction can discard its changes and simply re-increment the counter. One area of future work is to create a transaction friendly set of standard collection classes for C++ and Java.

3. Program Composition

Building software systems involves the composition of many building blocks, such as data structures, to create a working whole. Simple nesting of transactions allows basic libraries to be built and integrated into larger components, similar to the nesting of synchronized blocks in Java. Transactions can also offer alternatives to threads for parallelism, such as speculative loops that use transactions to maintain sequential semantics while taking advantage of additional processors for performance. Parallel looping constructs can dynamically detect the number of processors to use, running sequentially if higher levels of parallelism are already using other processors or using all processors if their is a sequential caller.

One challenge to program composition is the inclusion of I/O calls within a transaction. A general solution is to use buffering of input and output to remove any non-transactional operations from within transactions. In practice, this seems reasonable in many cases. For example, in a web server, a request can be read from the network, then a transaction can process the request buffering its output. If the transaction completes successfully, the buffer can be flushed across the network. If the transaction is aborted, it can be restarted with the same input. If there is a compelling reason, the I/O could be allowed from within the transaction, with some compensating code to be run on failure. For example, a transaction may want to use a private temporary file within a transaction, which could simply be deleted if the transaction is later aborted.

Another challenge to program composition is waiting on conditional variables within nested transactions. There is no transactional interpretation that is guaranteed to preserve the semantics of all lock based programs [3]. Fortunately, alternatives such as conditional critical regions fit well with transactional memory [10].

4. Operating Systems

Operating system kernels show great potential to benefit from Transactional memory. The Linux kernel's evolution from uniprocessor to multiprocessor support has shown many of the difficulties of locks. Over time there has been a move from a "one big lock" lock approach to finer grained locks for high contention data structures. The increased code complexity due to locking has led a significant number of bugs. For example, many drivers have had issues working in a multiprocessor environment because of requirements locking places on all accesses to shared data structures. Transactional memory can simplify kernel construction and potentially even provide performance improvements by speculating through former lock waiting points.

Transactional memory can also be used to build more reliable and secure operating systems. The atomicity of transactions prevents unexpected hardware or software faults from leaving shared system structures in an inconsistent state. The isolation provided by transactions can be useful for a secure system in that latent attacks from malicious code are always wrapped by transactions and remain ineffective to systems until they are filtered by security conformance test before the commit of the transactions.

The opportunities provided by transactional memory come with some challenges. Operating systems typically consider processors to have low preemption cost. Even though there is much talk of the cost of context switching, in practice the cost of save and restore the register set is negligible. Unfortunately, hardware transactional memory complicates preemption by extending the processor state to include the current state of transactional memory in addition to the register set. Operating system schedulers may need to take the transactional status of a processor into consideration, favoring the preemption of code not currently in a transaction. Forward progress can become a serious issue if transactions need to run longer than the scheduling quantum and are rolled back on preemption, although studies of common case transactional behavior of programs seems to indicate this is not likely to be a problem[7].

Interrupts handling creates similar problems, because we do not want the handlers state to affect an unrelated transaction that was already running. One potential solution is to run interrupt handlers as a sort of open nested transaction that runs and commits its changes without committing the parent that was already on the processor.

Physical device I/O operations are inherently non-transactional because they cannot be rolled back. Several approaches should be considered. First, some buffering techniques described above might work for non-block devices. Second, I/O operations could be delayed until commit, a typical approach taken in other transactional systems. Finally, perhaps simply prohibiting device I/O within a transaction would not be an undue burden on the kernel programmer.

Finally, virtual memory management needs to cooperate with transactional memory when updating virtual to physical address mappings so that changes in mapping do not cause the system to miss any data conflicts.

It is worth noting that the challenges of transaction preemption on context switch or interrupt are limited to hardware transactional memory systems. Software transactional memory systems do not share these issues, since by definition none of their transactional state is maintained within the processor. Hopefully, the challenges presented by hardware transactional memory in the context of operating systems can be explored in parallel with opportunities through the use of software transactional memory.

5. Language Implementation

Just as databases hide complexity in implementation in order to provide a simple interface, programming language runtimes often have to deal with details so the programmer does not have to. One area of complexity is memory allocation. For example, some software transactional memory systems internally differentiate between heap allocated and stack allocated objects in C/C++ [15]. Java garbage collectors need to be made transaction aware so parallel collection can avoid unnecessary overhead.

While there are challenges in implementing programming languages in a transactional environment, there are opportunities as well. For example, one can imagine taking advantage of violation detection to allow a garbage collector to run in parallel with mutator threads, where a violation works as event trigger for concurrent updates of mutators and collectors to heap and stack memory. By having violation handlers to do suitable adjustments at violations due

to conflicts between collectors and mutators, the burden of playing with fine-grain locking and synchronization schemes can be greatly mitigated [12]. In addition, one could imagine a JIT compiler making unsafe optimization with respect to reachability of a location, counting on violations to catch any problems in practice.

6. Programming Models

There seem to be as many different semantics for transactional memory as there are transactional memory proposals. While most provide the basic concept of begin transaction, commit transaction, and simple nesting, from this starting point there are departures for many competing destinations. One source of concern is that often proposals use the same name for differing semantics, with one example being the usage of open nesting.

A larger problem is when certain behavior is not defined at all. For example, there are two general forms of atomicity: strong and weak. Strong atomicity implies that a transaction's results are not visible to non-transactional code and that changes made to shared state by non-transactional code violate transactions that depend on that state. However some systems offer only weak atomicity, which means atomicity is preserved only if all accesses are from within transactions, but no guarantees are made for simultaneous read or write access from non-transactional code. This means that reads from outside transactions may read partially update state and that writes may break the atomicity assumptions of code that is running within a transaction.

We believe that strong atomicity should be a requirement of all transactional memory proposals, otherwise a programmer accidentally reading or writing shared state outside a transaction will destroy atomicity expectations in other parts of the system. While almost all hardware transactional memory proposals provide strong atomicity, until recently most software transactional memory proposals did not. We are encouraged to see that this has become a topic of interest in recent software transactional memory research. However, we are concerned when some hardware proposals offer extensions that create environments where strong and weak atomicity might be mixed within a program, especially when neither programming model is a functional subset of the other [2].

One design challenge is how to provide transactional features to the programmer in a structured way, particularly in choosing which features to expose to high level programmers and which to reserve for system programming. A large number of language proposals that were otherwise unrelated to transactions have recently added atomic constructs, although some are more completely specified than others [1, 5, 8]. Our own proposal for the Atomos programming language tries to address the boundary between transactional constructs for the programming language user and those programming language implementer, focusing on providing the basics like open and closed nesting and conditional waiting to the programmer and reserving the full power of violation handlers for the use under the hood [4].

7. Distributed Transactions

Large enterprise applications are typically built with distributed transactions. The programmer in this environment is already used to the concept of transactions, which have been used for decades as a vehicle to build reliable and efficient business systems. In these environments, a single transaction can span more than one system. Each system acts as a *resource manager* and a *transaction manager* employees a two phase commit protocol to ensure that the ACID properties of transactions are preserved in the event of the failure of any of the resource managers. It is important for the transactional memory community to be aware of the requirements placed on resource managers to allow future integration with external transaction managers.

Distributed transaction systems typically provide their own interface for coordinating transactions across the system, including such basic operations as committing a transaction, so that the requests are directed to the transaction manager. Some examples of these interfaces include Microsoft .Net's Enterprise Services for its Distributed Transaction Coordinator (DTC), Java 2 Enterprise Edition (J2EE)'s Java Transaction Service (JTS), and CORBA's Object Transaction Service (OTS). Any transactional memory interfaces, including transactional language proposals, need to consider how they could delegate their commit operations to use these interfaces if needed.

Distributed transaction systems often provide a declarative definition of the transaction proprieties of interfaces, separate from the code that implements those interfaces. For example, Enterprise Java Beans (EJBs), the distributed object system in J2EE, requires programmers to annotate interface methods with transactional properties such as TX_REQUIRED or TX_BEAN_MANAGED to indicate if a transaction is expected to already be present or should be internal to the bean. When calling these interfaces, transactional memory may be able to extract parallelism by using the interface annotations as hints about what calls might prove useful sources of method-level parallelism.

8. Legacy Code

No matter how successful transactional memory systems become, they will always have to integrate with existing legacy code. There are several approaches to taking advantage of transactions with existing code. For existing parallel code with locks, proposals such as TLR have explored converting critical sections into transactions, falling back to locks on rollback [14]. Even sequential code can take advantage of transactions with techniques to convert ordered loops into speculative transactions such as the Jrpm system for Java[6]. Similar techniques could be applied directly to native code with binary translation. Together these two approaches promise to smooth the transition for both existing parallel and sequential code to parallel transactional execution.

9. Conclusion

For transactional memory to deliver on its promise, transactions must be integrated across the software stack. While there are several challenges along that way, revisiting the software stack for transactions provides a great opportunity to improve performance, reliability, and ease-of-use in future systems based on multicore chips. It also provides an opportunity to re-energize collaboration between the architecture, programming language, compiler, operating systems, and distributed systems communities. Such synergistic work is necessary to create practical and truly scalable parallel systems.

Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant number NBCH104009. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

Additional support was also available through NSF grant 0444470. Brian D. Carlstrom is supported by an Intel Foundation PhD Fellowship.

References

[1] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, 2005.

[2] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[3] B. D. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Cao Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional Execution of Java Programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

[4] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The Atomos Transactional Programming Language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2006. ACM Press.

[5] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[6] M. K. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing java programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 434–445, June 2003.

[7] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The Common Case Transactional Behavior of Multithreaded Programs. In *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, 2006.

[8] Cray. *Chapel Specification*. February 2005.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[10] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402. ACM Press, 2003.

[11] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[12] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural Semantics for Practical Transactional Memory. In *Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[13] A. McDonald, J. Chung, H. Chafi, C. Cao Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on Chip-Multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 63–74, Washington, DC, USA, September 2005. IEEE Computer Society.

[14] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 5–17, New York, NY, USA, October 2002. ACM Press.

[15] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.