TRANSACTIONAL MEMORY: THE HARDWARE-SOFTWARE INTERFACE

THIS COMPREHENSIVE ARCHITECTURE SUPPORTS NESTED TRANSACTIONS, TRANSACTIONAL HANDLERS, AND TWO-PHASE COMMIT. THE RESULT IS A SEAMLESS INTEGRATION OF TRANSACTIONAL MEMORY WITH MODERN PROGRAMMING LANGUAGES AND RUNTIME ENVIRONMENTS.

Austen McDonald
Brian D. Carlstrom
JaeWoong Chung
Chi Cao Minh
Hassan Chafi
Christos Kozyrakis
Kunle Olukotun
Stanford University

As multicore chips become ubiquitous, the need to provide architectural support for practical parallel programming is reaching critical. Conventional lock-based concurrency control techniques are difficult to use, requiring the programmer to navigate through the minefield of coarse- versus fine-grained locks, deadlock, livelock, lock convoying, and priority inversion. This explicit management of concurrency is beyond the reach of the average programmer, threatening to waste the additional parallelism available with multicore architectures.

A promising solution to this dilemma is transactional memory (TM).[1] With TM, programmers declare code segments that must execute atomically and in isolation from all other code. Concurrency control becomes largely the system's responsibility, with no additional programming burden. The system monitors transactions that are executing optimistically in parallel, detects accesses to shared data that violate atomicity, and potentially aborts and reexecutes some transactions to restore atomicity.

Several TM implementation proposals advocate using hardware, software, or hybrid techniques,[2] and all propose implementing two key mechanisms. The first, data versioning, manages the new data values that transactions produce until they complete or abort. The second, conflict detection, tracks the addresses each transaction reads and writes to identify concurrent accesses that violate atomicity. Initial evaluations indicate that TM provides scalable performance with simple parallel code and that, with some hardware support, its bookkeeping overhead is low.

Nevertheless, for the programming community to broadly accept TM, it must work with modern programming languages and runtime systems. In current TM systems, the hardware-software interface is just a few instructions that define transaction boundaries. Although such limited semantics have been sufficient for initial demonstrations with simple benchmarks, they fall short of supporting key aspects of a modern programming environment. In real-world applications, transactions must work correctly with library calls, system calls, runtime exceptions, and even distributed application frameworks. TM systems should also support novel languages that build on transac-

0272-1732/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society.

tions using conditional synchronization and contention management.

To address these shortcomings, we defined a comprehensive TM architecture that includes expressive and clean interfaces between transactional hardware and software.[3] The architecture introduces three basic mechanisms:

- nested transactions with independent rollback;
- transactional handlers that support transaction commit, violation, and abort; and
- two-phase transaction commit.

These mechanisms require only a few registers and instructions, since their support is mostly through software conventions. This reduction of hardware resources and reliance on software is similar to the way modern architectures support function calls and interrupt handling: a few special instructions and heavy reliance on well-defined software conventions.

The three proposed mechanisms can support a rich set of functions in programming languages and runtime systems, including composable library calls, conditional synchronization, system calls, I/O, and runtime exceptions within transactions. To demonstrate their effectiveness, we added them to the Transactional Coherence and Consistency (TCC) TM system[4] and used them to implement the Atomos programming language.[5] TCC is a lazy versioning, optimistically concurrent TM system that we and colleagues at Stanford designed to explore the concept of "all transactions all the time," but these mechanisms are equally applicable to all proposed TM implementations. Atomos is a full-featured transactional programming language derived from Java that combines easy parallel programming with scalable performance.[5] This implementation demonstrated that introducing the three mechanisms—nesting, transactional handlers, and two-phase commit—does not introduce significant overhead. We also used nested transactions to improve performance by reducing the cost or frequency of conflicts.

We believe that the semantics of the three mechanisms will provide a solid substrate for future developments in TM software research. Perhaps more important, clean TM interfaces enhance hardware and software interoperability and allow researchers to pursue more cost-effective hardware support and practical programming environments apart from one another.

Addressing current challenges

Each of the three proposed mechanisms addresses the challenges in existing TM systems, including support for nested transactions, input/output (I/O) handling, system calls within transactions, error recovery and contention management, conditional synchronization, and system-level and distributed transactions.

Nested transactions

Modern programs rely on extensive hierarchies of libraries. Such libraries have well-defined interfaces, but their implementations are not transparent to users. Since libraries called within transactions can include atomic blocks, transactions will often be nested. Current TM systems deal with nested transactions by subsuming (or flattening) all inner transactions within the outermost transaction.[4-6] Flattening nested transactions poses both functionality and performance challenges. Several TM languages include constructs such as orElse that specify alternative control-flow paths if a transaction aborts.[7] It is impossible to support such functionality for a nested transaction if, when it aborts, the system always rolls back to the outermost transaction. Flattening can significantly degrade performance, since a conflict in a small, inner transaction can cause a large, outer transaction to reexecute. It then becomes impossible to use separate contention management policies for the inner and the outer transaction—a requirement for properly dealing with nested transactions.

In our proposed architecture, TM systems follow the closed nesting model in Figure 1a to support independent abort for nested transactions. The system tracks reads and writes in the inner transaction separately from those in the outer transaction.

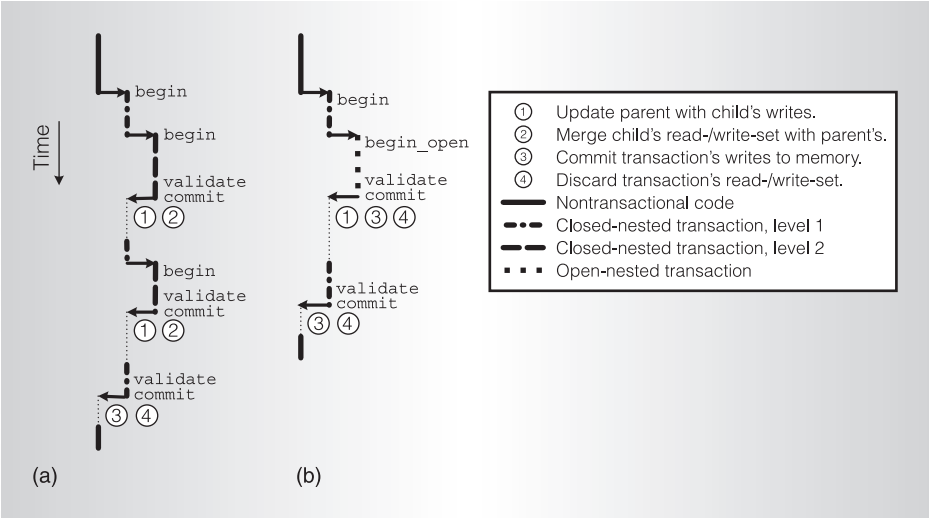

Figure 1. Timeline of nested transactions under the closed-nesting (a) and open-nesting (b) models.

Thus, if a conflict occurs, the nested transaction (child) can roll back to its beginning without aborting the outer one (the parent). At commit, the nested transaction's reads and writes are merged with the parent's (① and ②). However, no value that a child transaction produces becomes visible to the shared memory until the parent transaction commits (③ and ④).

System calls

Current TM systems prohibit system calls within transactions[6] or handle them by reverting to sequential execution.[4] Both approaches are unacceptable, since real programs include system calls that are often hidden in library calls. Simply nesting system call code within a user transaction can lead to undesirable effects, both for the user application and the operating system (OS). A closed-nested call to gettimeofday(), for example, would generate a dependency between the user transaction and the OS's clock. As the system periodically updates the clock, the user code will experience unnecessary and repeated conflicts that might even prevent forward progress.

Similarly, simultaneous calls to mmap() for memory allocation by two transactions will allocate the same chunk to both transactions, causing wasted work when one rolls back. If a transaction aborts after making a system call, such an approach could also (unintentionally) undo important OS bookkeeping and logging performed during the system call.

To support system calls, the system code should be able to read and update its data structures independently of the user transaction that triggered it. Our architecture meets this requirement by allocating a different memory chunk to each transaction and by avoiding unnecessary dependencies through OS data structures. The open-nesting model in Figure 1b illustrates how this works. Open nesting differs from closed nesting only in commit semantics. In an open-nested commit, the child transaction immediately updates the shared memory with its writes (③ and ④), as if it were a parent transaction. If the parent and child transactions overlap because of writes from the child, the parent transaction simply updates its read or write data—it does not generate a conflict. In practice, these differences imply that the atomicity and isolation of a child open-nested transaction are independent of those of its parent.

Thus, a transaction that makes an open-nested call to gettimeofday() will not experience conflicts because of clock updates after the child transaction returns. Similarly, two concurrent transactions that

make an open-nested call to mmap() will allocate different chunks. If the two mmap() calls overlap, the system can handle conflicts by aborting and reexecuting one of the calls as if it were a parent transaction. The dependencies through OS structures do not propagate to the user code when the call returns, regardless of how long the user transaction takes.

Because our architecture lets open-nested code update memory immediately, the effects of the child transaction become visible even if the parent transaction aborts. To undo these effects when the parent transaction fails, the system must take some compensating action. In our architecture, the system executes violation and abort handlers when their effects must be rolled back because of a conflict or an explicit abort. Thus, the open-nested call to mmap() will register a violation handler that calls munmap(). The system will invoke the handler if the parent transaction must be aborted after mmap() returns. A handler can have an arbitrary number of parameters and can define arbitrary code, including nested transactions. At any point, multiple handlers might be associated with a single transaction.

Open nesting can also be a way to enhance user code performance. In the server benchmark SPECjbb2000, processing an order requires first acquiring a unique order ID from a globally shared counter. Without open nesting, all transactions that process new orders experience frequent conflicts on the shared counter. By surrounding the acquisition of the order ID with an open-nested transaction, we atomically acquire a unique ID and are free to discard the dependence before moving on with order processing. In general, open nesting can be useful in eliminating conflicts when some nested code handles separate data from the parent transaction. However, programmers must use it with caution, since it could lead to violations of transactional properties if the open-nested transaction writes to the same shared variables as its parent.

Input/output

I/O calls, such as read() and write(), constitute an important class of system calls. Current TM systems handle I/O within transactions as they do other system calls, either disallowing it or reverting to sequential execution before executing I/O calls to avoid rolling back I/O actions. A better approach is to combine open nesting and transactional handlers to support the transactional execution of I/O calls. It's possible to implement an input call, for example, by using an open-nested transaction to remove data from OS buffers. If the transaction or one of its parents later aborts, the system can register a violation handler to restore the input data. This approach works well even if the input call happens within a deeply nested transaction.

Implementing output calls requires support for commit handlers. Just as violation handlers implement compensating actions, commit handlers provide finalizing actions that complete the I/O call when the parent transaction completes. When the program executes an output call, it places the output data in a temporary buffer and registers a commit handler. When the outermost transaction commits, the commit handler will copy the data to the OS output buffers. Like violation handlers, commit handlers can have an arbitrary number of parameters and can include nested transactions. The system can register multiple commit handlers at any point.

Error recovery and contention management

TM lets systems recover from software errors efficiently. For example, when a mechanism like Java's try/catch discovers an error, aborting the enclosing transaction rolls back all the side effects from the erroneous code. For debugging and monitoring, however, it is useful to expose some information about the error, and violation handlers can provide this function. A registered handler triggered before the transaction aborts can use open nesting to preserve important debugging information for later processing. Violation handlers can also streamline contention management. User code or the runtime environment can register handlers that determine how best to react to conflicts for each transaction. The handler can retry the transaction immediately or use a back-off technique. The

handler might even decide to ignore some conflicts and resume the transaction if it has application-level knowledge that it can do so safely. Overall, violation handlers enable application-, transaction-, and nesting-level-specific contention management that provides the best possible guarantees of performance and fairness. These handlers also support programming constructs that define alternative execution paths when a transaction aborts, such as tryatomic.[8]

Conditional synchronization

Locks not only provide mutual exclusion, but can also implement conditional synchronization primitives such as Java's wait/notify. TM can support conditional critical regions, and therefore efficient conditional synchronization that eliminates even the need for explicit notify statements in the user code.[9] A transactional conflict can act as a tripwire to detect when to reevaluate a waiting condition. The conflict indicates that another transaction has updated a value that the waiting transaction has read.

Several proposed languages include similar constructs for conditional synchronization using transactions, such as conditional atomic,[9] retry and orElse,[7] and when.[10]

Figure 2 gives an example that clarifies the requirements for implementing conditional synchronization.[5]

In Atomos, the consumer uses watch statements to identify the addresses it must wait for (watch set). When the consumer cannot make further progress, it calls retry to roll back the transaction and yield the processor. The implementation of retry uses open nesting to communicate the watch set to a scheduler, which listens for updates to these addresses on behalf of the now-yielded consumer. The open-nested transaction causes a conflict for the scheduler, and its violation handler reads the watch set from some mutually known location. When the producer commits a new value in the watch set, the scheduler receives a conflict. Instead of aborting the scheduler transaction, the violation handler will wake up (reschedule) the consumer. The producer did not have any knowledge about other threads waiting for this data and did not have an explicit notify.

```
public int consumer() {
    atomic {
        if (!available) {
            watch available;  // remember &available

            retry;            // communicate &available
                              // to scheduler; then roll back.
        }
        available = false;
        return contents;
    }
}
public void producer (int value) {
    atomic {
        if (available) {
            watch available;
            retry;
        }
        contents = value;
        available = true;
    }                         // commit new value; no notify needed
}
```

Figure 2. Producer-consumer example in the Atomos programming language.

System-level and distributed transactions

Thus far, we have assumed that conflicts or explicit aborts are the only factors determining whether a transaction completes or rolls back. However, real-world programs can access additional transactional resources within one atomic block, such as a database or a log-based file system. Before committing the transaction, the system must coordinate across all resources to ensure that there are no conflicts. Similarly, in a distributed system, a transaction within one process might communicate with transactions in other processes. Again, systemwide coordination is necessary before a transaction is committed. Finally, the user code might be running in parallel with checker threads that validate its security and integrity; hence, before committing any transaction, the system must consult the checkers.[11]

A two-phase commit makes it easier to handle system-level and distributed transactions. The first phase, *validate*, guarantees no conflicts on previously accessed data. The second phase, the actual *commit*, commits the updates to shared memory. Using open-nested transactions between the two phases enables communication with other system resources or processes. System-

wide coordination can result in continuing with the commit or aborting the transaction.

Complete architecture

To support real-world programming environments, a TM system must accommodate nesting, transactional handling, and two-phase commit. Our architecture combines software and hardware to implement these mechanisms in a way that balances complexity and performance.[3]

Software conventions

The transaction stack is the key software structure. As Figure 3 shows, the stack tracks the information for all nesting levels currently executing in a thread. When a new transaction begins, a new transaction control block (TCB) is pushed onto the stack. In this sense, the transaction stack is similar to a traditional software stack, with transaction creation being analogous to a function call. Each TCB uses thread-local storage to hold transaction metadata (status, nesting level, and so on) and vital information for versioning and conflict detection. Some TM implementations store TCB fields such as the read-set and write-set in hardware caches[4,6] or hardware signatures.[12]

The code pointers and arguments for transactional handlers are in separate stacks to enable a fixed-length TCB format while supporting unlimited handlers with an arbitrary number of arguments. The separate stacks also simplify handler management as nested transactions start, complete, or abort. To preserve undo semantics, when the system rolls back a transaction, violation handlers must execute in reverse of the order in which they were registered. Commit handlers, in contrast, run in the same order in which they were registered after validation of a parent or an open-nested transaction. When a child transaction commits, its handlers merge with those of its parent through a simple manipulation of the pointers in the parent's TCB.

All these rules are software conventions, and programmers can adjust them to match the requirements of a specific programming environment without any hardware changes.

A violation, such as a conflict between concurrent transactions, or an explicit abort can disrupt a transaction. Our architecture handles the two events in a way similar to the handling of user-level interrupts. The system automatically transfers control to a prespecified software monitor that examines the source of the interruption and determines a proper reaction (ignore conflict and resume, roll back, invoke contention-management code, and so on). If need be, the monitor will also invoke the violation and abort handlers registered in the TCB. To avoid repeated monitor invocations, the hardware detects additional conflicts while the current one is processing but does not immediately jump to the monitor. The monitor can decide to handle multiple conflicts with single or multiple invocations.

Because many transactions will be only hundreds of instructions long, the code that manipulates TCB entries as well as the violation and abort monitor should be highly tuned assembly code. This is analogous to compilers using assembly templates to generate function call prologues and epilogues. The transactional handlers themselves, however, can be in a high-level language.

Hardware support

For two-phase commit, the hardware provides separate validate and commit instructions. If the validate instruction executes successfully, the transaction is guaranteed not to roll back if a conflict occurs. Until the transaction commits, the conflict-detection logic gives it priority over any other transactions. The commit instruction atomically changes the transaction status to committed and makes its data visible to shared memory. To differentiate the beginning of open-nested transactions, the hardware provides separate begin and begin_open instructions.

A small set of instructions and registers makes it easier to invoke the monitor on violations and aborts,[3] but we used software for TCB manipulation and handler registration. To support nested transactions, the hardware provides separate versioning and conflict detection for each nesting level. In

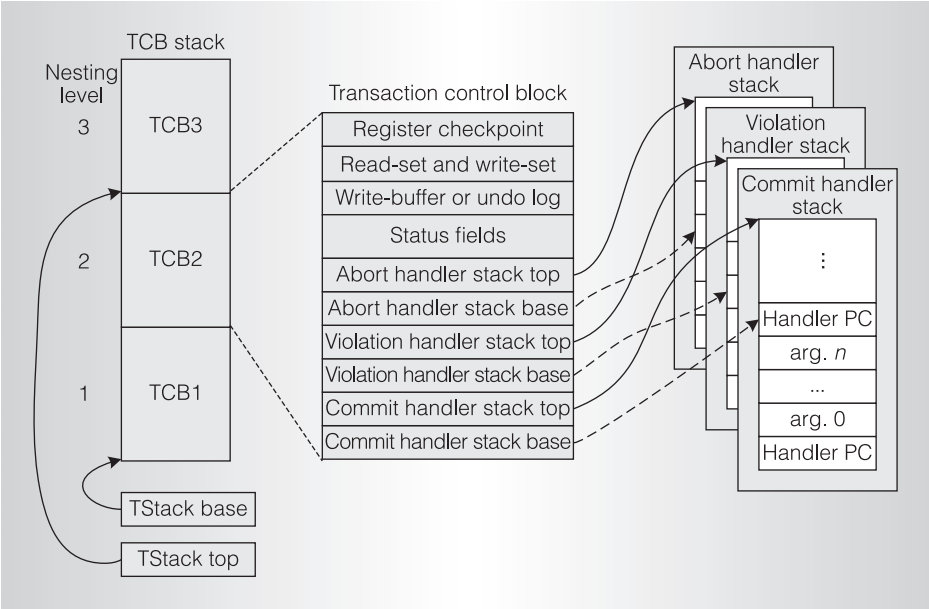

Figure 3. A transaction stack containing three transaction control blocks, one per active nested transaction, and the second entry (TCB2) in detail, complete with commit, violation, and abort handler stacks.

a multicore chip, implementing versioning and conflict detection requires modifying the caches and the coherence protocol. Existing systems use a single read (R) or write (W) bit per cache line to indicate whether the line belongs to the read-set or write-set of the flattened transaction. For nested transactions, our architecture uses multiple read and write bits, one for each transaction level, as in Figure 4a. This approach works well with TM systems that use eager versioning.[6] TM systems with lazy versioning[4] can use separate lines with a nesting level (NL) identifier to track the read-set and write-set of different transactions, as in Figure 4b.

A challenge for both approaches is how to merge overlapping read-sets and write-sets when a child transaction commits. To avoid complex hardware schemes, it is best to do merging in the background.[3] With either scheme, hardware can support only a limited number of nesting levels. After these are expended, a virtualization technique is required to extend the available levels.

An alternative approach is to use hardware signatures[12] to track the read-set and write-set for nested transactions outside the cache. A single set of hardware signatures is sufficient. When a nested transaction begins or aborts, it is possible to save or restore the signature of its parent from its TCB entry.

Sample implementation

To illustrate the expressiveness and efficiency of the proposed mechanisms for TM systems, we added them to the TCC architecture[4] and used them to implement Atomos.[5] Atomos uses transactions to im-

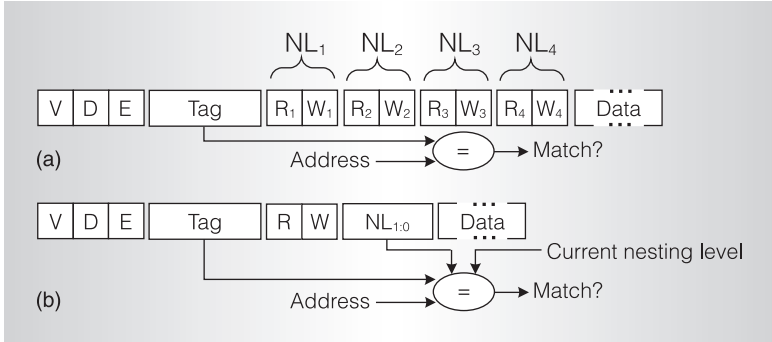

Figure 4. The two cache line designs, single read and write bits per cache line (a) and separate cache lines with a nesting level identifier (b). Both designs support tracking the read-set and write-set for multiple nested transactions.

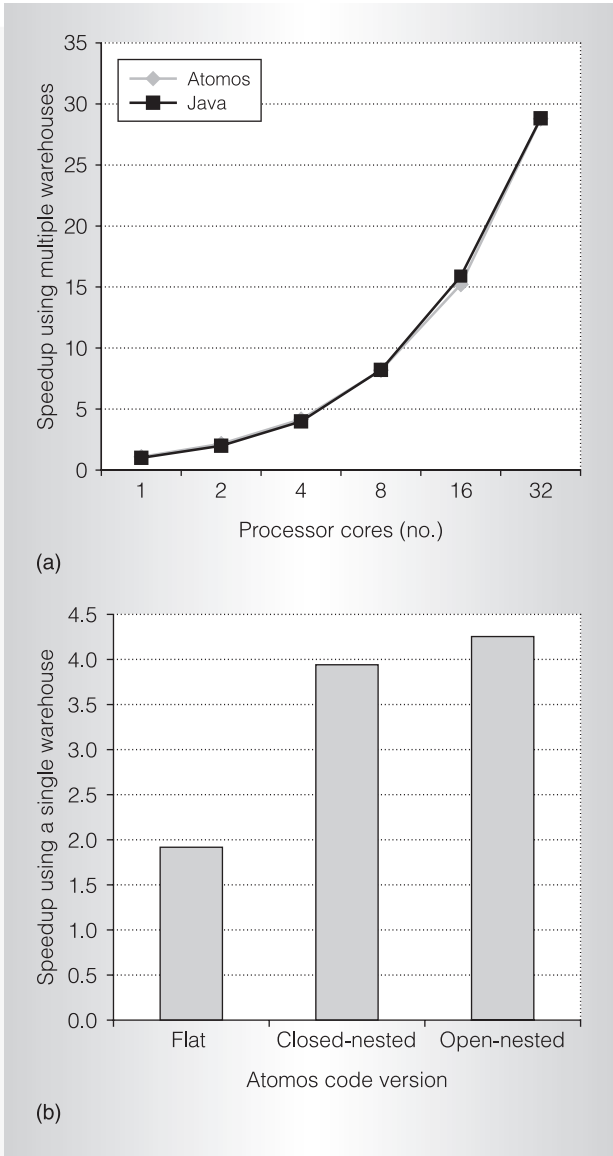

Figure 5. The speedup, over a single-core system, for the Java and Atomos versions of SPECjbb2000 using multiple warehouses and up to 32 processor cores (a) and the speedup of three Atomos versions of SPECjbb2000 using a single warehouse on eight cores (b).

plement conditional synchronization without the need for explicit notify statements. Its runtime system uses open nesting and violation handlers to support this functionality. It also provides transactional collection classes that wrap existing data structure code and allow their use in long transactions without frequent conflicts from data structure bookkeeping and memory allocation. Atomos implements collection classes using open nesting, commit, and violation handlers.[13] In our sample implementation, it used our architecture (the three mechanisms) to implement its I/O library.

Figure 5a compares the scalability of the Java and Atomos versions of the SPECjbb2000 three-tier benchmark on a simulated multicore system. The Atomos version of this benchmark replaces synchronized blocks with atomic blocks (transactions) and converts wait and notify conditional synchronization to Atomos's watch and retry. The two SPECjbb2000 versions scale similarly for up to 32 cores, which means that introducing the three mechanisms does not incur significant overhead. We also used nested transactions to improve performance by reducing the cost or frequency of conflicts.

Figure 5b compares the speedup over a single-core system achieved with three Atomos versions of SPECjbb2000. Unlike the experiment in Figure 5a, in which we made tasks parallel across multiple SPECjbb2000 warehouses (trivial parallelism), the experiment in Figure 5b exploits parallelism within a single warehouse. Conceptually, there is significant concurrency within a single warehouse, since different customers place orders and make payments mostly on different objects. Nevertheless, conflicts are possible because customer tasks manipulate shared data structures (B-trees). These conflicts are hard to predict statically and limit the speedup to 1.9 in the base case with flat transactions. With nesting support, we developed two additional versions of the Atomos code that double the base speedup. Closed nesting reduced the cost of rollbacks from B-tree searches and updates within long transactions. Open nesting reduced the frequency of conflicts on globally shared data, such as the order ID counter.

With the mechanisms we created, we were able to seamlessly integrate TM into modern programming and runtime environments. We also developed full-system TM prototypes that are practical for real-world application development, in which libraries, system calls, I/O and runtime exceptions are essential. Such

prototypes are critically important in evaluating the ease of parallel programming and the performance that TM can deliver.

Our work also opens new directions for research on transactional systems. With system-level transactions, programmers can use multiple transactional resources in a unified manner (memory, file systems, and so on). Distributed transactions can extend the transactional model to client-server applications without requiring a heavyweight database system to implement transactional semantics. An efficient TM architecture that supports rich software functionality is a requirement in both cases. MICRO

References

1. M. Herlihy and J.E.B. Moss, ''Transactional Memory: Architectural Support for Lock-free Data Structures,'' *Proc. 20th Int'l Symp. Computer Architecture* (ISCA 93), IEEE CS Press, 1993, pp. 289-300.
2. A. Adl-Tabatabai, C. Kozyrakis, and B. Saha, ''Unlocking Concurrency: Multi-Core Programming with Transactional Memory,'' *ACM Queue,* Dec. 2006, pp. 24-33.
3. A. McDonald et al., ''Architectural Semantics for Practical Transactional Memory,'' *Proc. 33rd Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 53-65.
4. L. Hammond et al., ''Transactional Memory Coherence and Consistency,'' *Proc. 31st Int'l Symp. Computer Architecture* (ISCA 04), IEEE CS Press, 2004, pp. 102-113.
5. B.D. Carlstrom et al., ''The Atomos Transactional Programming Language,'' *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 06), ACM Press, 2006, pp. 1-13.
6. K.E. Moore et al., ''LogTM: Log-Based Transactional Memory,'' *Proc. 12th Int'l Conf. High-Performance Computer Architecture* (HPCA 06), IEEE CS Press, 2006, pp. 254-265.
7. T. Harris et al., ''Composable Memory Transactions,'' *Proc. 10th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 05), ACM Press, 2005, pp. 48-60.
8. E. Allen et al., *The Fortress Language Specification*, Sun Microsystems, 2005.
9. T. Harris and K. Fraser, ''Language Support for Lightweight Transactions,'' *Proc. 18th Ann. ACM SIGPLAN Conf. Object-Oriented Programming* (OOPSLA 03), ACM Press, 2003, pp. 388-402.
10. P. Charles et al., ''X10: An Object-Oriented Approach to Non-Uniform Cluster Computing,'' *Proc. 20th Ann. ACM SIGPLAN Conf. Object-Oriented Programming* (OOPSLA 05), ACM Press, 2005, pp. 519-538.
11. J. Oplinger and M.S. Lam, ''Enhancing Software Reliability with Speculative Threads,'' *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 184-196.
12. L. Ceze et al., ''Bulk Disambiguation of Speculative Threads in Multiprocessors,'' *Proc. 33rd Int'l Symp. Computer Architecture* (ISCA 06), IEEE CS Press, 2006, pp. 227-238.
13. B.D. Carlstrom et al., ''Transactional Collection Classes,'' *Proc. Symp. Principles and Practice of Parallel Programming* (PPoPP 07), 2007.

Austen McDonald is a PhD candidate in computer science at Stanford University. His research interests are transactional memory and parallel architectures. McDonald has a BS in computer science from the Georgia Institute of Technology.

Brian D. Carlstrom is a PhD candidate in computer science at Stanford University. His research interests include transactional programming models as well as other aspects of modern programming language design and implementation. Carlstrom has an SB and MEng in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a member of the ACM.

JaeWoong Chung is a PhD candidate in electrical engineering at Stanford University. His research interests include transactional memory virtualization and operating system support for transactional programming. Chung has a BSEE and an MSEE from Korean Advanced Institute of Science and Technology.

Chi Cao Minh is a PhD candidate in electrical engineering at Stanford University. His research interests include computer architecture and parallel programming environments. Cao Minh received an MSEE from Stanford University and a BSEE from the University of Texas at Austin.

Hassan Chafi is a PhD candidate in electrical engineering at Stanford University. His research interests include transactional memory and parallel architectures. He has a BSEE from the University of Maryland, College Park.

Christos Kozyrakis is an assistant professor of electrical engineering and computer science at Stanford. His research focuses on architectural support for parallel programming, security, and power management. Kozyrakis has a PhD in computer science from the University of California, Berkeley.

Kunle Olukotun is an associate professor of electrical engineering and computer science at Stanford University. His research interests include computer architecture, parallel programming environments, and formal hardware verification. Olukotun has a PhD in computer science and engineering from the University of Michigan. He is a fellow of the ACM.

Direct questions or comments to Austen McDonald, Stanford University, Computer System Laboratory, 353 Serra Mall, Stanford, CA 94305-9040; austenmc@stanford.edu.

For more information on this or any other computing topic, please visit our Digital Library at http://computer.org/publications/dlib.