A significant road block to TM research is the lack of workloads designed with TM in mind. The Workshop on Transactional Memory Workloads (WTW) is being held as an effort to resolve this road block. We intend this to be a workshop in the best sense of the word, by allocating as much time to discussion as presentation, with an intended goal of charting a path toward developing a set of TM workloads that are shared within the TM community.

**Workshop Details**

- **Date:** Saturday, June 10th, 2006
- **Time:** 1:00pm - 5:30pm
- **Location:** Marriott Ottawa, Ottawa, Canada (co-located with PLDI / ACM Transact)

**Program Committee**

- **Program Chair:** Craig Zilles, UIUC
- Krste Asanovic, MIT
- Maurice Herlihy, Brown
- Jim Johnson, Microsoft
- James Larus, Microsoft
- Charles Leiserson, MIT
- Ravi Rajwar, Intel

**Corporate Sponsorship**

We would like to thank the Intel Corporation for their generous sponsorship of this workshop.
Keynote: “Taking Transactions Mainstream: Social Failure Modes and Recovery” (1:00pm - 2:00pm)
Don Box, Microsoft

Session I: 2:00pm - 3:00pm
Parallelizing SPECjbb2000 with Transactional Memory
JaeWoong Chung, Chi Cao Minh, Brian D. Carlstrom, Christos Kozyrakis, Stanford

Digital CAD Tools as Transactional Memory Workloads
Satnam Singh, Microsoft

Using Transactions in Delaunay Mesh Generation
Milind Kulkarni, L. Paul Chew and Keshav Pingali, Cornell

Break: 3:00pm - 3:30 pm

Session II: 3:30pm - 4:30pm
Concurrent Cache-Oblivious B-Trees Using Transactional Memory
Bradley Kuszmaul and Jim Sukha, MIT

The Linux Kernel: A Challenging Workload for Transactional Memory
Hany E. Ramadan, Christopher J. Rossbach and Emmett Witchel, University of Texas at Austin

Early Release: Friend or Foe?
Travis Skare and Christos Kozyrakis, Stanford

Panel Session: Toward Transactional Memory Workloads
4:30pm - 5:30pm
Keynote Address

“Taking Transactions Mainstream: Social Failure Modes and Recovery”

Don Box, Microsoft

Abstract: The last half of the 1990's saw a concerted effort to take transactions out of the database and shove them into mass-market programming environments. Both Microsoft and Sun heralded their respective offerings (MTS and EJB) as "the answer." However, by the end of the decade, it was no longer clear that the right question was being asked nor that the average developer liked the answer they were getting all that much. This talk looks at the successes and failures of this "transactions + objects" wave and tries to identify a few key areas we need to get right in the "transactions + memory" wave that is looming on the horizon.

Bio: Don Box is one of the original four developers of SOAP, a basic messaging layer for web services. Don Box is currently a programmer with a blossoming advertising business in the Pacific Northwest that also sells operating systems and applications. Don's first assignment with the firm was to work on Microsoft's implementation of SOAP, Windows Communication Foundation (a.k.a. Indigo). Don's current assignment is focused on making it more efficient for people to make their computers do what they want. Don is also a series editor with Addison Wesley where he's written four books in the area of software integration.

Panel Session

“Toward Transactional Memory Workloads”

Many of the challenges of architecting Transactional Memory systems derive from a lack of understanding how programmers will use such systems. In this respect, research in Transactional Memory systems would be greatly facilitated by the public availability of a set of workloads that were designed with transactional memory in mind. Developing a collection of workloads could be achieved through a community effort, but a number of challenges remain in realizing this goal.

Panelists:
Ali-Reza Adl-Tabatabi, Intel
Tim Harris, Microsoft
Bradley Kuszmaul, MIT
Michael Scott, Rochester
David Wood, Wisconsin
Parallelizing SPECjbb2000 with Transactional Memory

JaeWoong Chung, Chi Cao Minh, Brian D. Carstrom, Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{jwchung, caominh, bdc, kozyraki}@stanford.edu

1 Introduction

As chip-multiprocessors become ubiquitous, it is critical to provide architectural support for practical parallel programming. Transactional Memory (TM) [4] has the potential to simplify concurrency management by supporting parallel tasks (transactions) that appear to execute atomically and in isolation. By virtue of optimistic concurrency, transactional memory promises good parallel performance with easy-to-write, coarse-grain transactions. Furthermore, transactions can address other challenges of lock-based parallel code such as deadlock avoidance and robustness to failures.

In this paper, we use transactional memory to parallelize SPECjbb2000 [9], a popular benchmark for Java middleware. SPECjbb2000 combines in a single Java program many common features of 3-tier enterprise systems, hence it is significantly more complicated than the data structure microbenchmarks frequently used for TM research [3, 5, 8]. Since SPECjbb2000 models the operation of a wholesale company with multiple warehouses, the original code is already parallel, with a separate Java thread managing each warehouse. Different warehouses accesses mostly disjoint data structures, hence dependencies are rare. We focus on parallelism within a single warehouse, which is a more challenging case1. Conceptually, there are significant amounts of parallelism within a single warehouse as different customers order or pay for different objects. Nevertheless, all operations within a warehouse access the same data structures (B-trees), hence dependencies are possible and difficult to predict. Overall, single-warehouse SPECjbb2000 is an excellent candidate to explore the ease-of-use and performance potential of transactional memory with irregular code.

We study multiple ways of defining transactions within the SPECjbb2000 code and explore their performance potential using an execution-driven simulator for a CMP system with TM support. We demonstrate that even the simplest, coarse-grain transactions (one transaction for a whole operation) lead to significant speedups. Nevertheless, performance is still limited by frequent dependency violations on shared data structures. To mitigate the violation overhead, we use nested transactions, both closed and open, to achieve up to a 75% speedup increase. Closed nesting reduces the penalty of violations and open nesting reduces the frequency of violations. While nested transactions, particularly open, require some additional effort by the programmer, they are still significantly easier to use than fine-grain locks. Overall, we conclude that for SPECjbb2000, transactional memory lives up to its promise of good performance with simple parallel code.

2 SPECjbb2000 Overview

SPECjbb2000 is a Java benchmark, designed to measure how well systems execute Java server applications [9]. The benchmark emulates a 3-tier system for a wholesale company with multiple warehouses. Figure 1 shows the basic structure of SPECjbb2000. The first tier consists of client threads that perform random input selection to interact with the business

1 The performance benefits from parallelizing across warehouses are orthogonal and complementary to the benefits from parallelizing within a single warehouse.
logic in the second tier. The second tier is the most significant part of the benchmark and consists of a transaction server that accesses the data managed by the third tier. SPECjbb2000 holds all data with in-memory objects using B-trees (Java objects). Since there is no database, SPECjbb2000 does no disk I/O, and since a single program includes both the server and the clients, there is no network I/O either.

The second tier can be configured to have multiple warehouses, with each serving a number of districts. By default, SPECjbb2000 runs one thread per warehouse, which exposes significant amounts of concurrency as, for the most part, warehouses operate on disjoint data. In this paper, we parallelize the second tier of SPECjbb2000 for a single warehouse and a single district. From the end user’s point of view, this is an interesting case as a business should not have to introduce more physical warehouses to scale the performance of its 3-tier system. If the warehouse can scale to handle more customers, the same should be the case for the computing system that controls its operation. From the programmer’s point of view, parallelizing within a single warehouse is a challenging case as the code is irregular. Conceptually, there is significant concurrency within a single warehouse, as different customers operate mostly on different objects (orders, payments, etc.). Hence, one can use multiple threads to handle these operations. However, these threads will be accessing data from the same B-trees, causing dependencies between threads. These dependencies are difficult to predict as the customer operations are randomly generated. We should point out that the performance benefits from parallelizing across warehouses are orthogonal and complementary to the benefits from parallelizing within a single warehouse.

As shown in Figure 1, the second tier contains three key modules (TransactionManager, District, and Warehouse). The TransactionManager is responsible for receiving tasks from the clients and then making the appropriate changes in District and Warehouse. To parallelize within a warehouse, multiple TransactionManager threads are used. The District module contains a global counter, called newID, that is used to assign a unique number to each new order. When orders are created, District records them in the orderTable B-tree. Similarly, Warehouse uses the itemTable and stockTable B-trees to keep track of the appropriate items. Since newID and the three B-trees are shared and modified by all threads, they are possible sources of dependencies and conflicts between the concurrent TransactionManager threads.

Figure 2 shows the sequential (baseline) pseudocode for a TransactionManager thread. We have abstracted out several details including thread initialization and client tasks distribution. There are five types of client tasks to process: new_order, payment, order_status, delivery, and stock_level. The new_order and payment types represent 43% and 43% of the total number of tasks respectively. The pseudocode focuses on new_order because its processing involves more updates to shared data structures. Specifically, it involves reading and incrementing the newID, searching for and updating nodes in itemTable and stockTable, and allocating a new entry in orderTable. With multiple TransactionManager threads operating in parallel on one warehouse, conflicts through these shared data structures will
TransactionManager::go() {
    while (workToDo) {
        switch (transactionType) {
            case new_order:
                // 1. initialize a new order transaction
                newOrderTx.init();

                // 2. create a unique order ID and a new order with the ID
                orderId = district.nextOrderId(); // newID++
                order = createOrder(orderId);

                // 3. retrieve items and stocks from warehouse
                warehouse = order.getSupplyWarehouse();
                item = warehouse.retrieveItem(); // itemTable.search()
                stock = warehouse.retrieveStock(); // stockTable.search()

                // 4. calculate cost and update node in stockTable
                process(item, stock);

                // 5. record the order for delivery
                district.addOrder(order); // orderTable.insert()

                // 6. print the result of the process
                newOrderTx.display();
                break;
            case payment:
                // ...
                break;
            case order_status:
                break;
            case delivery:
                // ...
                break;
            case stock_level:
                // ...
                break;
            }
        }
    }
}

Figure 2: The pseudocode for a single-threaded (baseline) SPECjbb2000 TransactionManager.
be frequent. With lock-based coding, a programmer could simply acquire a coarse-grain lock to protect lines 9–21. The coarse-grain lock would guarantee correct execution but would also eliminate most concurrency within a warehouse. Alternatively, and after some observation, the programmer could acquire per data structure locks for the newID update (line 9), the stockTable node update (line 18), and the orderTable insertion (line 21). The per data structure locks would allow additional concurrency, up to one per data structure. Finally, a programmer could implement all objects using fine-grain locks, which would allow maximum concurrency, but at maximum coding complexity. Even if the fine-grain code for the object is available in a library, the programmer must make sure that the order in which the threads access objects will not lead to deadlocks or data races.

3 Methodology

We run transaction-based code on an execution-driven simulator for a CMP that follows the TCC architecture. Details about transactional execution with TCC are available in [7]. TCC provides transactional memory using lazy conflict detection to provide non-blocking guarantees. The CMP includes 8 cores with private L1 caches (32 KBytes, 1-cycle access) and private L2 caches (256 KBytes, 12-cycle access). The read-set and write-set of the transactions generated by SPECjbb2000 fit in the processor caches, hence there is no overhead for overflows and virtualization. The processors communicate over a 16-byte, split-transaction bus. All non-memory instructions in our simulator have CPI of one, but we model all details in the memory hierarchy for loads and stores, including inter-processor communication. The simulator implements closed- and open-nested transactions as described in [6], and the simulated hardware supports up to 4 levels of nesting (or nesting depth of 4). To run SPECjbb2000, we use the Jikes RVM, version 2.3.4 [1].

Even though we performed our study on top of a particular TM architecture, our conclusions about concurrency in SPECjbb2000 and its interaction with transactional memory techniques are largely independent from implementation details. We expect that SPECjbb2000 would perform similarly on other, well-engineered, transactional memory systems.

Figure 3 presents the performance results for the various transactional versions of SPECjbb2000. The results represent execution time with 8 TransactionManager threads on 8 processors normalized to the sequential (baseline) execution time of a single thread on one processor. Lower bars represent better performance. The bars also indicate the percentage of execution time wasted on dependency violations between concurrent transactions. The labels above each bar represent the speedup over the baseline (maximum speedup is 8). We measured execution time by running 368 client tasks, and the results focus on benchmark execution time, skipping virtual machine startup.

4 Transactional Parallelization of SPECjbb2000

We parallelized the TransactionManager for SPECjbb2000 with transactions in three different ways. The first approach uses flat, coarse-grain transactions, while the other two take advantage of the hardware support for nested transactions (open and closed).

4.1 Flat Transactions

Figure 4 shows the pseudocode for the first transactional version of TransactionManager. This transactional code is equivalent to coarse-grain lock-based code: the whole customer task (new_order) is expressed as a single, atomic transaction. Similarly, each other type of task is a single, flat transaction (payment, order_status, delivery, and stock_level). This version of the code is by far the simplest for the programmer. All she has to do is to use an atomic{} block (2 extra lines of code) to specify that the whole task should be atomic with respect to all other threads, regardless of the number, type, or order of objects accessed within the task. In other words, the programmer is only required to understand the atomicity behavior of the application at the highest possible algorithmic level. No understanding of the exact implementation of the task or the data structures is needed.

The first bar in Figure 3 shows that the coarse-grain (flat) transactional code provides a speedup of 3.09 on the 8-processor CMP. Despite using coarse-grain transactions, there is significant concurrency as threads often operate on different types of
Figure 3: The execution time of the transaction-based parallel version of SPECjbb2000 for 8 processors, normalized to the sequential time. The labels above each bar represent speedups.

Figure 4: Pseudocode of the first transactional version with flat, coarse-grain transactions.

```java
TransactionManager::go() {
    while (workToDo) {
        switch (transactionType) {
            case new_order:
                atomic { // begin transaction
                    // 1. initialize a new order transaction
                    // 2. create a unique order ID and a new order with the ID
                    // 3. retrieve items and stocks from warehouse
                    // 4. calculate cost and update warehouse
                    // 5. record the order for delivery
                    // 6. print the result of the process
                    } // commit transaction
                    break;
                ...
        }
    }
}
```

4.2 Closed Nesting

To reduce the overhead of each conflict between transactions from different threads, we used closed-nested transactions [6] as shown in Figure 5. Again, the whole customer task (new_order) is a transaction (lines 5 to 18). However, we define two nested transactions: (1) around the updates to the newID and the Warehouse B-trees (lines 8 to 12) and (2) around the updates to the orderTable B-tree (lines 14 to 17). The first nested transaction establishes a new order ID and identifies the items involved. The second nested transaction updates the order table. When a nested transaction begins, we track its tasks, on different B-trees, or different portions of the same B-tree. Compared to coarse-grain lock-based code that gets a speedup of approximately 1, the performance of the simple, coarse-grain, transactional code is very encouraging.

Nevertheless, approximately 63% of the execution time for each thread is wasted executing transactions that eventually rollback due to a dependency violation. We used the TAPE profiling tool to identify the main sources for the violations [2]. The first main source of conflicts is the update to the Warehouse B-tree (Figure 2, line 18) and the orderTable B-tree (Figure 2, line 21). The second main source was the newID counter that is read-modified-written for every new order by a customer (Figure 2, line 9). We attempted to reduce the penalty and frequency of these conflicts by using nested transactions.
TransactionManager::go() {
    while (workToDo) {
        switch (transactionType) {
            case new_order:
                atomic { // begin outermost transaction
                    // 1. initialize a new order transaction
                    ...
                }
                atomic { // begin nested transaction
                    // 2. create a unique order ID and a new order with the ID
                    // 3. retrieve items and stocks from warehouse
                    // 4. calculate cost and update warehouse
                    } // end nested transaction (merge with parent)
                }
                atomic { // begin nested transaction
                    // 5. record the order for delivery
                    // 6. print the result of the process
                    } // end nested transaction (merge with parent)
                // commit outermost transaction
                break;
                ...
        }
    }
}

Figure 5: The SPECjbb2000 parallel pseudocode with closed-nested transactions.

read-set and write-set independently. Hence, if a conflict with another thread is detected and it involves only the nested (inner) transaction, we only need to rollback to the beginning of the nested transaction and not to the beginning of the outer one. For example, if we detect a conflict on the orderTable B-tree while the thread is in line 16, we only roll back and re-execute from line 14 as opposed to line 5. This can significantly reduce the overhead of a conflict. However, when a closed-nested transaction commits, we simply merge its read-set and write-set with the parent transaction. Hence, if we detect a violation on the newID counter while the thread is in line 14, we have to roll back all the way to line 5 as we can no longer separate the first nested transaction from the outermost one.

The second bar in Figure 3 shows that closed-nested transactions eliminate approximately half of the overhead from violations and lead to an overall speedup of 5.36 on the 8-processor CMP (73% improvement over flat transactions). Closed nesting does not eliminate all overheads as the number of conflicts remains constant and some of them still roll back all the way to the beginning of the outermost transaction. In terms of coding difficulty, closed-nested transactions require additional effort from the programmer in terms of defining the nested transaction boundaries. Nevertheless, the code changes do not affect the correctness of the code. With respect to the other threads, the atomicity boundaries are defined by the outermost transaction. Closed-nested transactions are merely a performance optimization. Hence, a programmer can define their boundaries after identifying the common conflict patterns with a profiling tool like TAPE [2].

Nested transactions typically introduce a small runtime overhead as a few additional instructions must be executed on their boundaries [6]. In this case, we did not observe any slowdown (additional useful time), as the nested transactions are large enough to amortize these overheads. Actually, we observed a small reduction in useful time due to better caching behavior when transactions do not roll back all the way to the beginning of the outermost transaction.

4.3 Open Nesting

Figure 6 shows the transaction-based version of SPECjbb2000 using an open-nested transaction. Specifically, we used an open-nested transaction for the update to the newID (lines from 8 to 10) to eliminate the common conflict between all threads operating on a new order. Without the open-nested transaction, all new orders are serialized due to the read-modify-
TransactionManager::go() {
    while (workToDo) {
        switch (transactionType) {
            case new_order:
                atomic { // begin outermost transaction
                    // 1. initialize a new order transaction

                    open_atomic { // begin open-nested transaction
                        // 2. create a unique order ID and a new order with the ID
                    } // commit open-nested transaction

                    // 3. retrieve items and stocks from warehouse
                    // 4. calculate cost and update warehouse
                    // 5. record the order for delivery
                    // 6. print the result of the process
                } // commit outermost transaction
                break;
        ... break;
    }
}

Figure 6: The SPECjbb200 parallel pseudocode with open-nested transactions.

write on the shared newID variable. The open-nested transaction allows new order tasks to overlap significantly. When the open-nested transaction reaches its end (line 10), we do not just merge its write-set (the newID counter) to the parent’s write-set. We actually commit it to shared memory so that other threads can use the new value of the counter\(^2\). Hence, two threads executing new order tasks generate a conflict only if their updates to newID overlap. This case is rare as the read-modify-write operation is fast and has a low penalty (it causes a rollback to line 8, not line 5). Note that the two threads may later detect a conflict on another object (e.g., orderTable), which will cause one of the two to roll back its transaction. When that transaction re-executes, it will get a new newID. For SPECjbb2000 this is safe, as order IDs need to be unique but not necessarily sequential. In other cases, an open-nested transaction must be accompanied by compensation code to run if the parent transaction aborts after the nested transaction commits (see [6] for implementation details).

The third bar in Figure 3 shows that adding one open-nested transaction in the original code from Section 4.1 leads to speedup of 4.96q (60% improvement over flat transactions). Even though there are still other conflicts that occur, the open-nested transaction virtually eliminates one of the most frequent sources of conflicts. The disadvantage of open-nested transactions is that the programmer must be very careful when using them as they change the atomicity behavior with respect to other threads. Much better understanding of the code is necessary to decide when an open-nested transaction can be safely used and if compensation code is necessary. For the case of the newID counter in SPECjbb2000, using open nesting is relatively simple, but this case does not necessarily generalize. On the other hand, expert programmers can use open nesting to create fast libraries of commonly used objects. In this case, the library would contain an object that returns unique, but not sequential, IDs. Regular programmers can use open nesting through such libraries by understanding the service provided without being exposed to the coding complications of open nesting.

We should note that we could have also eliminated the conflict on newID by providing each thread private counters with a sufficient offset so that they do not overlap. Conceptually, this is as difficult as using open nesting for this specific case, as the programmer must recognize that private counters are sufficient and implement an offset scheme that is safe. Moreover, the safety of the offset scheme is dependent on the number of threads used and the length of a SPECjbb2000 run. Hence, in some cases, open nesting can be a viable alternative to privatization. Another interesting alternative is to use a separate flat transaction to create a new ID before entering the main transaction by moving up line 9 over the main transaction. Programmers need to be cautious when splitting a single transaction into smaller ones as they may introduce races due to

---

\(^2\)For more details on the semantics and implementation of open nesting refer to [6]
atomicity breach. In this case, the programmer must notice that there is no dependency between the new order transaction initialization code (line 6) and new ID creation (line 9).

5 Conclusions

In this paper, we used transactional memory to parallelize the SPECjbb2000 code that operates on a single warehouse. We demonstrated that flat, coarse-grain transactions, which are trivial to use, lead to significant speedups over the sequential version. The frequency and penalty of conflicts between concurrent transactions can be reduced significantly by using nested transactions. Closed-nested transactions are easy to use as they only involve performance tuning and do not affect correctness. Open-nested transactions are more difficult to use as the programmer must ensure that they do not break the atomicity requirements of the application and that compensation code is provided if needed.

There are several more versions of the SPECjbb2000 code that one can construct using different boundaries or combinations for closed and nested transactions. In particular, combining the closed and open nested transactions presented in Sections 4.2 and 4.3 respectively should provide for further performance improvements. However, we believe that the presented versions sufficiently demonstrate the ease-of-use and performance potential of transactional memory with irregular code. They also raise interesting issues with respect to nested transactions in user-level programming.

References


Digital CAD Tools as Transactional Memory Workloads

Satnam Singh
Microsoft
One Microsoft Way
Redmond WA 98052, USA
+1 425 705 8208
satnams@microsoft.com

Abstract

This paper proposes some digital CAD tools for circuit design as candidates for parallel STM workloads and suggests that others are probably suited to conventional lock based synchronization. This domain has an interesting and diverse set of challenging algorithms which can often not parallelized despite the long run-time of many digital design tools. We present our preliminary experience of adapting an existing FPGA design and layout tool to use STM.

1. INTRODUCTION

Computer aided (CAD) tools for digital design often require compute intensive calculations for tasks like modeling, synthesis, simulation, netlist generation, automatic placement and routing, delay analysis, and formal verification. This paper proposes some compute intensive digital CAD algorithms as workload candidates for experimental transactional memory systems. Such tools rarely have parallel implementations however digital designers would greatly benefit from improved performance from parallel computing hardware. We also identify some algorithms that may be better suited to lower level synchronization and coordination approaches (or for use with open transactions).

We also report our preliminary experience of adapting an existing FPGA CAD tool for circuit design and layout. The original version of this system can take several minutes to take totally floorplanned circuit descriptions and elaborate them into lower level implementation representations that can be used by vendor tools to produce the final programming bitstream. We attempted to parallelize some of the more compute intensive portions of the system using transactions where there was access to shared state information from multiple threads.

2. DIGITAL CAD TOOLS

2.1 CIRCUIT MODELING

The early stages of digital design typically involve modeling circuits at a high level of abstraction. Sometimes this is done by writing models in a conventional programming language like C or by using a special library written in C++ to support event-based simulation (SystemC). System modeling may also use conventional hardware description languages like VHDL and Verilog as well as alternative languages like Haskell (e.g. the ForSyDe system [7]).

System level modeling of digital circuits is a compute intensive operation which may involve simulation runs that takes hours or perhaps days. System level modeling is used to explore the design space for architecture evaluation and more efficient execution of system level models may result in better implementations.

Transactional memory may help in several aspects of system level modeling:

Safe concurrent access to shared resources in the model. For example, at the modeling level two or more circuits may need to share and modify some resource (e.g. a memory). A detailed implementation would at a later stage specify the exact mechanism for providing consistent access to the shared resource. However, at the early abstract modeling layer one is not concerned with exactly how the consistent access is implemented. Here one could use transactional memory to model such shared resources and this makes it easier to parallelize the modeling code that needs to work with such shared resources.

Certain kinds of exceptional behavior may be more easily specified by using transactional alternatives like orElse at the modeling level which are later refined into more specific strategies for dealing with exceptions. For example, a timeout request for some bus-based interaction could be captured with a composite transaction (A orElse B) where A is the normal behavior and B corresponds to the behavior when a timeout occurs (such an event in the model should raise in exception in A which would cause the B branch to be attempted).

Transacted activities in the model may be directly modeled by transactional memory. For example, certain kinds of bus protocol interactions and network protocol interactions may be directly modeled with an atomic block whereas the
modeling code for such behavior without using transactions as a language feature may be quite involved.

Attempting to use transactional memory for system level modeling of digital circuits seems like an appealing approach because it may allow for faster execution of models and in some circumstances the abstraction offered by transactional memory fits the requirements of the model.

2.2 Placement and Routing
Automatic placement and routing involve automatically determining the location of gates on a chip (or a location on an FPGA) and then automatically determining how to connect the gates that have been laid out with wires. These are very compute intensive operations which may benefit from parallel implementations. One straightforward way to exploit parallelism for both placement and routing is to spawn off totally separate processes for placement and routing which try different strategies and accept the first result that satisfies certain criteria (e.g. a critical path delay below some threshold for performance requirements). Typically these tasks are executed on different computers and one does not need to write any tricky concurrent code.

A less common strategy is to try and parallelize a given run of a placement or routing task. Although such an approach is more difficult to program it may potentially lead to better results because a program can be written to learn from paths which were explored but did not lead to a solution that would satisfy all the given constraints. Knowing what kinds of placements are not feasible or what kind of routes would violate timing requirements is information that could be shared amongst several parallel threads which are exploring various placements and routes. Transactional memory could provide an effective way to manage this information amongst several threads.

Atomic blocks also facilitate the parallel implementation of placement and routing. For placement, the overall problem may be broken down into several sub-problems which can be executed atomically in parallel. Either each sub-problem succeeds (i.e. finds a valid placement or route) which allows the result to be committed and reflected in the master data structure for the design or it is deemed to have failed and the calculation is aborted and the main circuit data structure is left unaffected.

The circuit routing problem may benefit from nested transactions to avoid unnecessary recalculations. Sometimes a routing problem from one point to another involves performing several sub-routes which may themselves have sub-routes. Here one could perform sub-route calculations in parallel and compose them hierarchically using nested transactions which yields significantly simpler control logic for the calculation and permits a parallel implementation.

Often placement is performed first and then routing is performed afterwards. Some systems then ping-pong between placement and routing to try and reduce critical paths or other design constraints like area. However, routing should be able to start as soon as some placement is done. Having a circuit implementation data structure with transacted access (at an appropriate granularity) would make it much easier to write programs that performed placement and routing in parallel. Earlier routing information (e.g. wire delays) may also help to inform the router sooner of when it needs to rip-up a given placement and try again.

2.3 Simulation
Circuits are simulated at several different levels of abstraction: system level, design level (VHDL or Verilog before synthesis), and implementation level (after placement and routing). The primary mode of verification for digital circuits is testing through simulation which may take hours or even days.

Most simulation runs are based on an event-based simulation model which is also the model behind several hardware description languages and libraries e.g. VHDL, Verilog and SystemC. The event-based simulation model is not particularly amenable to parallelization. However, there is some scope for executing parts of the event-based simulation model machinery in parallel. The general approach for such simulation techniques involves having a list of sensitive signals which trigger event handling code whenever an assignment occurs to such variables. The event handling code may itself trigger further assignments to sensitive signals which have to occur in the current logical time step or at a future time step. Within a logical time step one may pick the sensitive signals for processing in any order and the handler code may be run in parallel. The concurrent access to sensitive signals from the event queue and the enqueuing of more sensitized signals could occur using transacted memory. However, this is a case where there may be very high contention for the single event queue which may lead to poor performance on many transactional memory systems. Furthermore, this may be a case where an explicit lock is the right mechanism.

2.4 Verification
Automatic circuit verification by static analysis is a computationally intensive task which could benefit from parallelization. Examples of static analysis for digital circuit verification include combinational equivalence checking and model checking.
Combinational equivalence checking involves taking two circuit descriptions (graphs) of sequential circuit and then trying to identify corresponding state elements using heuristics. Once this step has been successfully completed the corresponding combinational logic fragments between state elements are checked for functional equivalence. One way of doing this is to transform these chunks of combinational logic into a canonical form e.g. by using BDDs. This approach works well when the circuits to be compared encode state information in exactly the same way (or in a very similar way). This approach can be used to verify very large circuits (millions of gates). An alternative to using BDDs is to use a SAT solver to check if two combinational circuit fragments are functionally equivalent.

It is not certain that STM is a good choice for parallelizing BDDs or SAT. We have implemented a prototype SAT solver in Haskell using STM to represent shared clause information which helps to prune the search space although right now we do not have encouraging results. However, it may be the case the BDDs and SAT packages really need to be engineered to be very efficient and the very limited way in which shared information is manipulated in such systems may make them reasonable candidates for the explicit use of locks. Furthermore, the internal state information that is shared between threads exploring the search space is not exposed to users of the SAT library so there is probably less for a compositional concurrency approach. Alternatively, perhaps open transactions can provide a reasonable compromise between the convenient programming model of STM and the efficiency of lower level synchronization and coordination mechanisms.

A more general sequential equivalence checking approach involves using model checking and this can be used when the two state encodings are completely different. Model checking involves building up an explicit (or symbolic) representation of the state space of the circuits behavior (as a Kripke structure) and then (in principle) performing a search of this structure to check to see if all the reachable nodes satisfy some property (or to see if an undesirable state is reachable as part of a safety property check). It is possible to build a bounded model checker using induction and a SAT solver [8]. A model check could provide an interesting example of a STM workload application that uses transactions at different levels simultaneously. For example, the underlying infrastructure for the model checker (a SAT solver or BDDs) could use (open) transactions; the code for constructing the model could use transactions to allow the state space to be constructed in parallel; the actual state space exploration could occur in parallel and reachability analysis could also be performed in parallel. However, not all of these operations require transactions since some of them are truly independent parallel searches with no shared state information.

3. STM in Concurrent Haskell

The following section introduces a case study that used STM in Haskell [2] to parallelize a system for FPGA circuit design. This section is largely borrowed form [4] and provides a background to the STM mechanism in Haskell. Concurrent Haskell [5] is an extension to Haskell 98, a pure, lazy, functional programming language. It provides explicitly-forked threads, and abstractions for communicating between them. These constructs naturally involve side effects and so, given the lazy evaluation strategy, it is necessary to be able to control exactly when they occur. The big breakthrough came from using a mechanism called monads [6]. Here is the key idea: a value of type IO a is an “I/O action” that, when performed may do some input/output before yielding a value of type a. For example, the functions putChar and getChar have types:

\[
\begin{align*}
\text{putChar} & : \text{Char} \rightarrow \text{IO} () \\
\text{getChar} & : \text{IO} \text{ Char}
\end{align*}
\]

That is, putChar takes a Char and delivers an I/O action that, when performed, prints the string on the standard output; while getChar is an action that, when performed, reads a character from the console and delivers it as the result of the action. A complete program must define an I/O action called main; executing the program means performing that action. For example:

\[
\begin{align*}
\text{main} & : \text{IO} () \\
\text{main} & = \text{putChar} \ 'x'
\end{align*}
\]

I/O actions can be glued together by a monadic bind combinator. This is normally used through some syntactic sugar, allowing a C-like syntax. Here, for example, is a complete program that reads a character and then prints it twice:

\[
\begin{align*}
\text{main} & = \text{do} \{ \text{c} \leftarrow \text{getChar}; \text{putChar c}; \text{putChar c} \}
\end{align*}
\]

Threads in Haskell communicate by reading and writing transactional variables, or TVars. The operations on TVars are as follows:

\[
\begin{align*}
\text{data} \ \text{TVar} \ a & \\
\text{newTVar} & : \text{a} \rightarrow \text{STM} \ (\text{TVar} \ a) \\
\text{readTVar} & : \text{TVar} \ a \rightarrow \text{STM} \ a \\
\text{writeTVar} & : \text{TVar} \ a \rightarrow \text{a} \rightarrow \text{STM} ()
\end{align*}
\]

All these operations all make use of the STM monad, which supports a carefully-designed set of transactional operations, including allocating, reading and writing transactional variables. The readTVar and writeTVar operations both return STM actions, but Haskell allows us to use the same do \{ \ldots \} syntax to compose STM actions.
as we did for I/O actions. These STM actions remain tentative during their execution: in order to expose an STM action to the rest of the system, it can be passed to a new function atomically, with type

\[ \text{atomically :: STM } a \rightarrow IO a \]

It takes a memory transaction, of type STM \( a \), and delivers an I/O action that, when performed, runs the transaction atomically with respect to all other memory transactions. For example, one might say:

\[
\text{main = do \{ ...; atomically (getR r 3); ... \}}
\]

Operationally, atomically takes the tentative updates and actually applies them to the \( TVars \) involved, thereby making these effects visible to other transactions. The atomically function and all of the STM-typed operations are built over the software transactional memory. This deals with maintaining a per-thread transaction log that records the tentative accesses made to \( TVars \). When atomically is invoked the STM checks that the logged accesses are valid – i.e. no concurrent transaction has committed conflicting updates. If the log is valid then the STM commits it atomically to the heap. Otherwise the memory transaction is re-executed with a fresh log.

Splitting the world into STM actions and I/O actions provides two valuable guarantees: (i) only STM actions and pure computation can be performed inside a memory transaction; in particular I/O actions cannot; (ii) no STM actions can be performed outside a transaction, so the programmer cannot accidentally read or write a \( TVar \) without the protection of atomically. Of course, one can always write atomically (\( \text{readTVar v} \)) to read a \( TVar \) in a trivial transaction, but the call to atomically cannot be omitted. As an example, here is a procedure that atomically increments a \( TVar \):

\[
\text{incT :: TVar Int -> IO ()}
\]

\[
\text{incT v = atomically \{ do x <- readTVar v writeTVar v (x+1) \}}
\]

The implementation guarantees that the body of a call to atomically runs atomically with respect to every other thread; for example, there is no possibility that another thread can read \( v \) between the read\( TVar \) and write\( TVar \) of incT.

An example of how a concurrent data structure from the Java JSR-166 [3] collection can be written using STM in Haskell appears in [1].

4. CASE STUDY: CIRCUIT LAYOUT IN LAVA
The Lava system is a library of CAD tools for designing and verifying circuits for implementation on Xilinx FPGAs (http://raintown.org/lava). The system is implemented as a collection of Haskell libraries and designers describe circuits using a domain specific library which provides convenient higher order combinators for describing data parallel architectures. A key feature of Lava is that it provides combinators that specify circuit layout and behavior. This section introduces the layout mechanism in Lava and then goes on to show how we attempted to parallelize the circuit implementation and layout using STM.

There are two types of layout combinators: deterministic and non-deterministic. The deterministic layout combinators always produce the same layout for the same input specification and these form the basic core layout combinators in Haskell. Non-deterministic layout combinators are more interesting from a concurrency perspective because these combinators attempt to explore a variety of layouts until one is found which satisfies a given metric (typically width or height or both).

Lava circuits are conceptually enclosed in rectangular tiles which can be laid out relative to each other. There are two kinds of tiles that Lava supports: (a) two-sided tiles which communicate only along two opposite sides and (b) four sided tiles that can communicate along all four sides of the tile.

Lava uses a Cartesian coordinate system to describe the position of circuit tiles. The origin is at the bottom left hand corner. The Lava coordinate system is designed to make it easier to produce circuit descriptions that are portable between various FPGA architectures by abstracting as much as possible information about where resources are located.

The serial composition combinator in Lava is written as \( >> \) and it takes two circuits as its arguments and returns a new composite circuit. This combinator performs two distinct functions:

It composes behaviour by taking the output of the first circuit and connecting it to the input of the second circuit (i.e. connecting two circuits in series).

It composes layout by placing the second circuit to the right of the first circuit. Every circuit starts off with its bottom left hand corner at location \( (0,0) \). By using layout combinators the relative locations of circuits can be modified.

Note that one of the novel features of Lava is that one combinator combines both layout and behaviour. Rather than relegating layout information to inflexible annotations Lava affords layout information equal status to behavioural specification.
An example of serial composition is shown below for the circuit $\text{and2} \rightarrow\rightarrow \text{inv}$. The circuit $\text{and2}$ (a two input AND gate) takes a pair of signals as its input and returns a single output bit. This composition is illustrated in Figure 1.

The $\text{and2}$ circuit is realized in a two-sided Lava tile which means that signals (either inputs or outputs) can occur on the left or right hand sides of the tile but not the top or bottom. The $\text{inv}$ circuit (an invertor) is also realized in a two-sided tile. To make a NAND gate from these components all one needs to do is to connect the output of one circuit to the input of the other circuit. This task can be performed by the serial composition combinator without having to name the intermediate signals.

![Figure 1. Serial Composition](image)

However the serial composition combinator does more than just connect wires together. Serial composition looks at the sizes of the circuits it has to compose and it lays out one circuit directly next to the other circuit with their bottoms aligned. For example consider the tiles containing the $\text{inv}$ and $\text{and2}$ gates which each have size $(1,1)$. The serial composition combinator takes the output of the $\text{and2}$ gate and connects it to the input of the $\text{inv}$ gate resulting in a circuit that behaves like a NAND-gate. It also places the $\text{inv}$ gate to the right of the $\text{and2}$ gate so it ends up in location $(1,0)$. The composite NAND-gate circuit tile has size $(2,1)$ as illustrated in Figure 2.

![Figure 2. Layout of Serial Composition](image)

When Lava generates an EDIF netlist it translates its own coordinate system into an appropriate coordinate system for the chosen target architecture. Lava can also describe netlists containing no layout information using regular function composition. Lava also include blank spacer tiles to help with pitch alignment of datapath circuits.

The serial composition combinator is useful when we wish to compose and lay out two circuits that communicate. Sometimes however we wish to lay out circuits relative to each other which do not communicate. One of the many combinators that Lava provides for composing circuits in parallel is the $\text{par2}$ combinator. This combinator takes two two-sided tiles and places the second tile above the first tile aligned along their left hand edges. An example layout for the circuit $\text{par2 inv and2}$ is shown in Figure 3.

![Figure 3. Parallel Composition](image)

Any circuit or combinator that takes two inputs can be written as an infix operator in Lava by enclosing the name of the circuit or combinator in back quotes e.g. $\text{par2 inv and2}$ is the same as $\text{inv } \text{par2 and2}$. The $\text{par2}$ combinator produces a composite circuit that takes a pair
(i.e. a two element tuple) as its input and returns a pair. The first element of the input pair is the input destined for the first (lower) circuit and the second element of the input pair is the input destined for the second (upper) circuit. The first element of the output pair is the output from the first (lower) circuit and the second element of the output pair is the output from the second (upper) circuit. The layout of the composite circuit tile of size \((1,2)\) in the Lava coordinate system is shown in Figure 4.

When Lava is used to implement circuits the flow involves elaborating the embedded domain specific descriptions into an internal representation which is further processed to calculate layout information and finally an implemented netlist is written out either in a vertex-oriented format (the EDIF netlist format) or a node-orientated format (the VHDL hardware description language). These outputs can be fed directly to Xilinx’s implementation tools to produce the final programming bitstream required to actually configure an FPGA chip.

![Figure 4. Layout of Parallel Composition](image)

The running time of the circuit implementation phase is considerable for large circuits with large amounts of nested layout information (several minutes or almost an hour for big circuits). Until now the Lava system has been written as a purely sequential program. We recently modified the system to use Haskell’s STM mechanism and its lightweight threads as well as Haskell’s parallelism features (e.g. the \texttt{par} operator).

First, we consider constructs in the domain specific language which deal with concurrency and we try and exploit this property to make the processing of such constructs also concurrent. For example, the parallel composition operator \texttt{par2} in the original system was defined as:

\[
\text{par2} :: \text{Monad} m \Rightarrow (a \rightarrow m b) \rightarrow (c \rightarrow m d) \rightarrow (a, c) \rightarrow m (b, d)
\]

\[
\text{par2} \text{ circuit1 circuit2 (input1, input2)}
\]

\[
= \text{do output1 <- circuit1 input1}
\]

\[
\text{output2 <- circuit2 input2}
\]

\[
\text{return (output1, output2)}
\]

The Haskell \texttt{par} combinator can be used to request the parallel execution of a calculation. For example \(e1 \text{ `par` } e2\) will try and calculate \(e1\) and \(e2\) in parallel and return the result of \(e2\). The parallel composition combinator can be re-written to take advantage of the fact that the processing of the two circuits in parallel can take place concurrently:

\[
\text{par2} :: \text{Monad} m \Rightarrow (a \rightarrow m b) \rightarrow (c \rightarrow m d) \rightarrow (a, c) \rightarrow m (b, d)
\]

\[
\text{par2} \text{ circuit1 circuit2 (input1, input2)}
\]

\[
= \text{do output1 <- circuit1 input1}
\]

\[
\text{output2 <- circuit2 input2}
\]

\[
\text{return (output2 `par` output1, output2)}
\]

The monad class used here is parameterized (as \(m\)) and this overloaded description can be instantiated with different specific monads to achieve results like netlist generation, circuit simulation and timing analysis.

The Haskell \texttt{par} combinator can be used to request the parallel execution of a calculation. For example \(e1 \text{ `par` } e2\) will try and calculate \(e1\) and \(e2\) in parallel and return the result of \(e2\). The parallel composition combinator can be re-written to take advantage of the fact that the processing of the two circuits in parallel can take place concurrently:

\[
\text{par2} :: \text{Monad} m \Rightarrow (a \rightarrow m b) \rightarrow (c \rightarrow m d) \rightarrow (a, c) \rightarrow m (b, d)
\]

\[
\text{par2} \text{ circuit1 circuit2 (input1, input2)}
\]

\[
= \text{do output1 <- circuit1 input1}
\]

\[
\text{output2 <- circuit2 input2}
\]

\[
\text{return (output2 `par` output1, output2)}
\]

This optimization is only safe if the code that implements the netlist generation algorithms safely access the shared data structures that represent the internal representation of the circuit. We use Haskell’s STM mechanism to code up safe concurrent access to the shared circuit representation.

In the original system a data-structure was used to represent the state of the circuit netlist which was built up as the domain specific circuit description was processed. The original type for the circuit state was:

The Lava system provides many other combinators that compose layout and behavior e.g. in rows, columns, grids etc. Users can then write quite sophisticated recursive layout descriptions using just algebraic layout information and the Lava system then automatically computes the Cartesian coordinate of each cell. A good example of such a sophisticated layout is the highly compact and high performance Batcher’s bitonic sorter which is floorplanned in Lava for Xilinx FPGAs (described at [http://raintown.org/lava/sorter.htm](http://raintown.org/lava/sorter.htm)).
type LavaState = (Int, Int, [Port], [InstanceTree], DriverList)

type Out = State LavaState

The state of the circuit was represented by a tuple (LavaState) that had fields which counted the number of nets in the circuit, the number of instances (gates, or nodes in the circuit graph), a list of the circuit’s I/O ports, a list of circuit fragments (organized as a tree) and a list of ports that are drivers (i.e. an output port). This state type is then used to define a type (Out) for a monad which encapsulates this state information. The Haskell State monad has the following interface:

```
class Monad m => MonadState s m | m -> s where
    get :: m s
    put :: s -> m ()

newtype State s a
    = State {runState :: (s -> (a, s))}
```

For example, a function which adds a new instances of a circuit gate to the global circuit netlist could be written as:

```
newNetNumber :: Out Int
newNetNumber
    = do (netCount, instCount, ports, instances, dl) <- get
        put (netCount+1, instCount, ports, instances, dl)
        return netCount
```

The get function retrieves the state that was encapsulated by the monad and the put function modifies the state.

The STM version of the Lava system modifies the definition of the state variable to use transacted values for most fields:

```
newtype LavaState
        = (TVar Int, TVar Int, [Port],
           TVar [InstanceTree],
           TVar DriverList)
```

We now have to use a monad transformer because we need to nest one monad inside another. The inner monad is the IO monad because the atomic actions that operate over the transacted variables are executed in the IO monad. The outer monad provides the state encapsulation.

A state transformer monad is introduced using the following type:

```
newtype StateT s m a
    = StateT {runState :: (s -> m (a, s))}
```

The new state type definition is then:

```
type Out = StateT LavaState IO
```

A state transformer monad is defined with the StateT type and its parameters are the state to be encapsulated in the outer monad (here a tuple containing mainly transacted variables) and then the type of the inner monad (in this case the IO monad).

To allow us to write IO actions inside state transformer monads use can use the lift function which will lift IO operations to operations in the outer monad. This now lets us write the STM version of the function that returns a fresh net number:

```
newNetNumber :: Out Int
newNetNumber
    = do (netCountTV, instCountTV, ports, instances, dl) <- get
        lift (atomically
            (do nc <- readTVar netCountTV
                writeTVar netCountTV (nc+1)
                return nc))
```

As before the get function returns the state (this time retrieved from a state transformer monad). One of the state elements is a transacted variable that contains the next available fresh net number (netCountTV). The operations that we would like to perform atomically are (i) read a value from the transacted variable and return it; and (ii) increment the transacted count. These operations occur in the STM monad and the atomically acts on these STM instructions to execute actions in the IO monad if the transaction commits. The IO monad instructions do not have the right type to be used directly in this function and must be lifted to the type of the state transformer (Out) by using the lift function.

We re-wrote the Lava system to perform several operations in parallel by explicitly creating Haskell lightweight threads using calls to forkIO. We defined a function which combined the operations for atomically executing an STM block which results in an IO action which is forked off in a new thread and then lifted to the type of the outer monad:

```
liftFork :: STM () -> Out ()
liftFork program
    = lift (do forkIO (atomically program)
                    return ())
```

We used this in several places to specify the parallel execution of code in the STM monad. For example, we redefined the function that used to add circuit nodes to the main graph using regular State monad put and get operations with the following version that uses liftFork:
addInstance :: Instance -> Out ()
addInstance newInstance
  = do (nc, ic, ports, instancesTV, dl) <- get
   liftFork
     (do instances <- readTVar instancesTV
         writeTVar instancesTV
         ((Inst newInstance):instances))

A consequence of these modifications is that the process of netlist generation is no longer deterministic. The previous version of Lava would always generate the same netlist for a given circuit including the names of internal nets and instances (gates) and the list of instances always occurred in the same order. The modified STM version does not have this property (although the generated netlists are still totally valid and pass through FPGA simulation and implementation tools).

One of the most compute intensive parts of the Lava implementation flow is the layout calculations which take the algebraic layout combinators in the users’ circuit descriptions and produce a corresponding set of Cartesian coordinates for each gate subject to placement constraints. This process involves geometric translations in the x and y coordinates which could be performed in parallel. This code for the elaboration of the “left of” combinator which places a circuit graph \( t1 \) to the left of a circuit graph \( t2 \) is shown below.

```
elaborateInstTree (dx', dy') (LeftOf t1 t2)
  = do lhs <- forkWait
       (mapM (elaborateInstTree (dx, dy)) t1)
       let (w,h) = sizeOf (concat lhs)
       rhs <- mapM
         (elaborateInstTree (dx+w, dy)) t2
       return (concat lhs ++ concat rhs)
       where
         dx = 0 "max" dx'
         dy = 0 "max" dy'
```

This function effects a translation by \((dx', dy')\) of its argument (in this case the \(\text{LeftOf}\) constructor for circuit graphs \(t1\) and \(t2\)). The body of this code has to work out the width of circuit \(t1\) and then translate circuit \(t2\) to the right so it is right next to \(t1\). The original code was slightly modified by adding a call to a function called \(forkWait\) which takes a calculation in the IO monad which returns a result and schedules it for asynchronous execution. The STM mechanism is used to provide a future-style coordination pattern:

```
forkWait :: IO a -> IO a
forkWait program
  = do it <- atomically $ newEmptyTMVar
       forkIO (do v <- program
                  atomically $ putTMVar it v)
       atomically $ takeTMVar it
```

forkWait takes an IO action (\(program\)) and creates a new transacted variable that is used inside a forked program for synchronization. The \(forkWait\) function wait until the asynchronously executed task has completed and returns the result of the asynchronous calculation. The layout elaboration function recursively calls itself to translate its constituent sub-circuits and these calls will cause further layout IO actions to be asynchronously scheduled for execution.

5. CONCLUSIONS

We propose digital CAD tools as an interesting domain of compute intensive applications that could make good STM workloads. Many of the algorithms used for circuit design and analysis tools have wider applicability e.g. SAT-solvers, BDD packages, simulated annealing, model checking, and placement and routing problems.

We demonstrated how we were able to modify an existing CAD tool for FPGA design and layout to use STM with only small modifications to the source code. This system expends a large amount of time computing the layout of circuits for FPGA chips. We were able to exploit some of the inherent parallelism in the nature of the problem to help structure the parallelism of the actual software used to compile circuit descriptions. We are currently profiling the new system on an 8-processor machine to understand the performance impact of our re-write.

Some of the modifications that we present could have been satisfactorily performed using locks. For example, when computing a deterministic layout the work of computing sub-layouts are entirely independent with the exception of node numbers which can be shared with a lock or one may devise a scheme node numbers are generated in a way that permits parallel splitting (thus avoiding the lock altogether).

Although the examples in this paper have been presented in Haskell which is also the implementation language of Lava one needs to also consider the use of transactions in mainstream languages like C/C++ which are often used to implement CAD tools for performance reasons. Attempting to use transactions in an imperative language with pointers, aliasing etc. is a considerable challenge. If languages like Java and C# were to evolve STM-based features then they may become more competitive as CAD tool
implementation languages because they may make it easier to exploit multiple cores than lower level languages like C and C++.

REFERENCES


Abstract
Mesh generation is a key step in graphics rendering and in using the finite-element method to solve partial differential equations. The goal of mesh generation is to discretize the domain of interest using polygonal elements such as triangles (in 2-D) or tetrahedra (in 3-D).

One popular mesh generation algorithm is Delaunay mesh generation, which produces meshes with certain quality guarantees that are important for problems in which the geometry of the problem changes with time. Delaunay mesh generation works by iterative refinement of a coarse initial mesh. The sequential algorithm repeatedly looks for a “bad” mesh element that does not satisfy the quality constraints, computes a neighborhood of that element called its cavity, and replaces the elements in that cavity with new elements, some of which may not satisfy the quality guarantees themselves. It can be shown that the algorithm always terminates and produces a guaranteed quality mesh, regardless of the order in which bad elements are processed.

Delaunay mesh generation can be parallelized in a natural way because elements that are far away in the mesh do not interfere with each other as they are being processed. We present experimental results showing that in practice, there is indeed a lot of parallelism that is exposed in this way. However, exploiting this parallelism in practice can be complicated. Compile-time analysis and parallelization are infeasible because of the input-dependent nature of the algorithm. One alternative is optimistic parallelization. We show how logical transactions can be identified in a natural way in this code, argue that current transactional memory implementations are inadequate for this application, and suggest an alternative conception of transactional memory that addresses these problems.

1. Introduction
Most partial differential equations cannot be solved exactly, so it is necessary to use numerical techniques such as the finite-element and finite-difference methods to solve them approximately. The finite-element method transforms the calculus problem of solving the partial differential equation into the algebraic problem of solving systems of linear equations. A key step in this process is mesh generation.

In general, mesh generation refers to the problem of discretizing a continuous domain by placing points in a plane (in the two-dimensional case) or in a space (in higher dimensions), and forming a mesh over those points. Mesh generation algorithms are also useful in graphics, where they are used to tessellate curved surfaces so they may be represented as polygons, which can be rendered more easily. In this paper, we consider the problem of mesh generation for two-dimensional domains, although most of the ideas discussed here generalize to three dimensions.

In graphics as well as in finite element analysis, mesh quality is an important consideration. The problem may require that the mesh meet certain quality guarantees, so not every tessellation of the domain is adequate. These guarantees may include bounds on the size of the largest angle in any triangle or on the size of the largest triangle.

One technique for producing guaranteed quality meshes is Delaunay mesh generation. Delaunay mesh generation replaces elements that do not satisfy the constraints with elements that do, producing new, refined meshes by inserting new points into the mesh. The basic algorithm was described by Chew [4] and extended by Ruppert [13]. They provide the appropriate refinement procedure, as well as mathematical guarantees regarding the quality of the mesh upon the termination of the procedure. Shewchuck’s Triangle program [15] is an efficient implementation of the Delaunay mesh generation algorithm.

In this paper, we argue that the correct approach to parallelizing the Delaunay mesh generation refinement is to use optimistic parallelization. We also discuss the pros
and cons of using transactional memory for implementing optimistic parallelization for this application.

The remainder of this paper is structured as follows. Section 2 describes the Delaunay mesh generation algorithm and provides pseudocode listings for the major parts of the sequential algorithm. Section 3 discusses the parallelization opportunities inherent in the algorithm and shows how optimistic parallelization is essential for this application. In Section 4, we discuss how existing transactional memory techniques can be applied to this algorithm, and discuss their shortcomings, together with possible solutions, in Section 5. We describe ongoing work in Section 6.

2. Delaunay Mesh Generation

A Delaunay mesh is a mesh over a set of points which satisfies the Delaunay property [4]. This property, also called the empty circle property, states that the circumcircle of any element (triangle) in the mesh (i.e. the circle which circumscribes the three vertices of the triangle) should not contain any other point in the mesh. An example of such a mesh is shown in Figure 1. In the absence of four co-circular points, a given set of points on a surface has only one triangulation that satisfies the Delaunay property.

In practice, the Delaunay property alone is not sufficient, and it is necessary to impose various quality constraints governing element shape and size. To meet these constraints, Delaunay mesh generation algorithms use an iterative refinement procedure that fixes elements that do not satisfy quality constraints.

This refinement procedure is best understood as a worklist algorithm. In a worklist algorithm, units of work are placed on a list. When necessary, a unit is removed from the list and processed. Any additional work units produced during this step are placed back onto the worklist. In the case of Delaunay mesh generation, the units of work are elements that do not meet the quality constraints (“bad” elements or triangles). The refinement procedure terminates when the worklist is empty.

Figure 2 shows the pseudocode for the mesh generation procedure. The key steps of this procedure are as follows.

1. Find all the bad elements in the mesh and place them into a workqueue, which is an implementation of the worklist [line 3]. Then repeat the following steps until the queue of bad triangles is empty [line 4].
2. Pick an element from the queue [line 5], such as the shaded element in Figure 3(a). The processing of other bad elements may have removed this element from the mesh. If so, there is no work to be done [line 6].
3. Find the circumcenter of the element. This is the new point that will be added to the mesh [line 7]. In Figure 3(a), this is the black point.
4. With respect to this new point, several existing elements will no longer satisfy the Delaunay property (i.e. the new point lies within their circumcircles). Determine the set of elements that are affected by the new point. The set of elements is called a cavity, and the process of finding these elements is called cavity expansion [line 8]. In Figure 3(b), the shaded grey elements represent the cavity.
5. Calculate a new set of elements which fills the cavity while incorporating the new point. This is the retriangulation step [line 9].
6. Replace the cavity with the new elements (i.e. remove the old elements from the mesh, and add in the newly calculated elements) [line 10]. See Figure 3(c).
7. Because the newly created elements are not guaranteed to meet the quality constraints, any new elements

Figure 1. This mesh obeys the Delaunay property. Note that the circumcircle for each of the triangles does not contain other points in the mesh.
that are “bad” must be added to the workqueue [line 11].

From the last step, it might appear that the algorithm could potentially not terminate. However, it is guaranteed that no elements can be created whose sides are smaller than the initial segments in the mesh. Since newly created elements are smaller than the original elements, the refinement process must terminate [4]. Note that this holds regardless of the order in which bad elements are processed.

We now describe how the key steps of the refinement procedure (expansion, retriangulation and update) work. A graphical representation of these steps for a single element is shown in Figure 3.

**Expansion:** In this step, we determine both the cavity (the elements affected by the new point) as well as the border elements of the cavity (the elements immediately surrounding the cavity). Because a cavity is a connected region of the mesh, this step can be accomplished with a simple breadth-first search, starting from the initial bad element, and adding affected elements to the cavity. The border of the cavity is defined by the elements we encounter that are not affected by the new point (which also serves as the termination criterion for that direction of search).

**Retriangulation:** At this stage, we determine the new set of elements that will replace the affected elements. The new elements should fill the same space as the original cavity, but include the new point. This is accomplished by determining the “boundary” of the cavity (i.e. the outer edges of the cavity), and creating new elements whose vertices are the new point and the vertices of a segment on the border.

**Update:** Because both the cavity and the new elements share the same boundary, and hence have the same border elements, replacing the cavity with the new elements is straightforward.

**Encroachment:** There is one special case, called encroachment. If the cavity contains a boundary segment of the overall mesh (i.e. the tightest circumcircle of the segment contains the new point), we say that the cavity encroaches upon the boundary. We first build a cavity around the boundary segment by placing a new point at its midpoint, and perform the complete refinement process. After this is done, we return to the original bad element and process it again.

3. Parallelization

Two facts are noteworthy about the mesh generation algorithm described in Section 2.

- Each cavity is a connected region of the mesh, and is small compared to the overall mesh.
- There is no specific order in which bad elements need to be processed.

3.1 Parallel Mesh Generation

The first fact leads naturally to a parallel algorithm. Because the cavities are localized, we note that two bad elements that are far enough apart on the mesh will have cavities that do not interfere with one another. Furthermore, the entire refinement process (expansion, retriangulation and graph updating) for each element is completely independent. Thus, the two elements can be processed in parallel, an approach which obviously extends to more than two elements (see Figure 4).
Figure 4. An example of processing several elements in parallel. The left mesh is the original mesh, while the right mesh represents the refinement. In the left mesh, the dark grey triangles represent the “bad” elements, while the horizontally shaded are the other elements in the cavity. In the right mesh, the black points are the newly added points and vertically shaded triangles are the newly created elements.

3.2 Compile-time Parallelization

Compile-time parallelization approaches perform dependence analysis to determine a partial order of program operations, and schedule operations for parallel execution if there are no dependences between them. We do not know of any compile-time parallelization technique that will succeed in finding parallelism in this problem. Since each iteration of the while loop of Figure 2 reads and writes the mesh data structure, any compile-time analysis technique that treats the entire mesh as a monolithic unit will assert that there are dependences from every iteration to all succeeding iterations. Since the mesh is read at the beginning of each iteration and updated at the end of that iteration, there is little useful overlap of computations between iterations. A more sophisticated, fine-grained analysis might try to use techniques like shape analysis [8, 14] to discern if the reads and writes to the mesh data structure in different iterations are disjoint. However, such an analysis requires determining whether the cavities of two bad triangles are disjoint, but this depends on the mesh, and thus is not a question that can be determined at compile time. Note also that any compile-time parallelization of the code in Figure 2 will still process bad triangles in the same order as the sequential code would, which is unnecessarily restrictive.

3.3 Optimistic Parallelization

Since static parallelization will not work, we turn instead to optimistic parallelization. At this stage, we leverage the second insight regarding the sequential algorithm: the elements can be processed in any order. Therefore, we do not need to adhere to a specific schedule of processing bad triangles, but can instead expand cavities whenever we can ensure that they can run in parallel with other concurrent expansions. To implement this sort of parallelization, we can perform dynamic checks to detect interference during cavity expansion. For example, we can lock mesh elements during cavity expansion; if some element needed for a cavity expansion is already locked by another cavity expansion, there is interference and one of the cavity expansions must be rolled back. If no interference is detected, we make the appropriate changes to the mesh. In this way, we are able to exploit the inherent parallelism in the mesh generation algorithm even without knowledge of which elements can be processed in parallel, but at the risk of doing useless work in computations that get rolled back.

3.4 Experimental Results

Optimistic parallelization is useful only if the risk of rollbacks is small. A priori, it is unclear whether or not optimistic parallelization is useful for Delaunay mesh generation. In addition, the amount of parallelism is very data-dependent and depends on the size of the mesh, the number of bad triangles, etc. The probability of conflict between two concurrent cavity expansions depends not only on these factors but on the scheduling policy for parallel activities.

Antonopoulos et al. [2] have investigated how many cavities could be expanded in parallel in a mesh of one million triangles (see Figure 5). They focused on coarse-grain parallelization on a distributed-memory computer, which required mesh partitioning and distribution. They found that across the entire problem, there were more than 256 cavities that could be expanded in parallel until almost the end of execution, and, halfway through execution, there were between 350 and 800 thousand cavities that could be expanded in parallel. These results
Figure 5. Feasibility study from [2], examining the number of concurrently expandable cavities during the execution of the Delaunay mesh generation algorithm. The x-axis represents time, while the y-axis shows the number of expandable cavities. The upper and lower borders of the shaded area represent upper and lower estimates for expandable cavities.

Figure 6. Parallel efficiency results for a lock-based implementation of parallel Delaunay mesh generation.

show that there is adequate potential for parallelism in this problem for mesh sizes of practical interest.

The study of Antonopoulos et al. provides upper and lower bounds for potential parallelism because the probability of conflicts between cavity expansions depends on the policy for enqueuing and dequeuing bad triangles on the worklist of Figure 2; some orderings will lead to more parallelism than others. Furthermore, the practically achievable parallelism is constrained by the issues involved in updating the shared structures (i.e. the worklist and the mesh) concurrently. Thus, although the Antonopoulos result shows that the problem exhibits significant parallelism theoretically, there are many factors which may affect the realistic parallelism available in the problem.

To understand these issues, we built a prototype shared-memory implementation. Coordination among concurrent activities was provided by locking. As the cavity was expanded, the triangles needed by the cavity were successively locked. If a triangle needed by a cavity was already locked by some other cavity, a conflict was recorded and cavity expansion was aborted. A two-phase locking scheme was used to avoid cascading rollbacks. We found that even this prototype implementation gives fairly good results, achieving a parallel efficiency (parallel speedup divided by number of processors) of approximately 75% on four processors as shown in Figure 6.

These results suggest that optimistic parallelization of Delaunay mesh generation should work well in practice.

4. Transactional Memory

As is well-known, the use of locks to implement parallel algorithms can be cumbersome since the programmer has to focus on proper lock placement, ensure appropriate roll-backs in case of conflicts, etc. In contrast, transactional memory [1, 5, 6, 7, 12] promises a simple solution to parallelization, relieving the programmer of many of these burdens. In this section, we show how the sequential algorithm can be transformed to run in parallel, using transactions to provide proper synchronization between concurrent activities. Pseudocode for this algorithm is shown in Figure 7.

Most studies of transactional memory are concerned with converting parallel code with locks into parallel code that uses transactions. Our research project is concerned more with identifying opportunities for optimistic parallelization in sequential code. The use of the transactional model only provides us with a synchronization mecha-
nism but does not specify how the program itself should be parallelized.

In our approach, there is a thread pool in which there are several threads, each of which draws work from the workqueue of Figure 2. Because we want to ensure that concurrent cavity expansions do not interfere, we see that each iteration of the loop in Figure 2 naturally maps to a single transaction. The interference detection provided by transactional memories can detect when two cavities overlap, and hence serve as the trigger for rolling back expansion. The buffering and rollback mechanisms inherent in transactional memory implementations allow for rollbacks without any additional programmer input.

There are a few issues that arise during this type of parallelization. First, we must perform all modifications to the workqueue (both choosing which elements to process as well as enqueuing newly created bad elements) outside the transactions. This is because modifications to shared structures made by a transaction are not visible to other transactions until that transaction commits. Thus, to avoid multiple transactions choosing the same element, we move these operations outside the transaction, and synchronize them using standard mechanisms such as monitors or locks. The second point is that all reads and writes that touch shared structures (which are any that read/create elements or update the mesh) must use transactional reads and writes. This may require rewriting the data structures in terms of transactional operations.

An important note is that, unlike many of the benchmarks that transactional memories are applied to, the transactions in this problem are long running and access relatively large amounts of memory. Table 1 provides some data about the average performance characteristics of a single transaction in this problem when executed on an Itanium 2.

We see that the transaction executes for far longer than most microbenchmarks. Also, the number of memory operations is significantly higher than the number of operations involved in the microbenchmarks of, e.g., [5]. Note also that although the number of L1 cache misses is not necessarily representative of working set size, it indicates that the actual working set is likely to be much larger than the 250 lines of L1 cache on our target system, meaning that hardware transactional memories may overflow their transactional caches.

These characteristics make simple hardware transactional memories [7] unsuitable for our purposes. However, software transactional memories, such as [5, 6], or more advanced hardware approaches [1, 12] may suffice, although their efficiency in the context of long-running transactions must still be studied.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Average value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>730K</td>
</tr>
<tr>
<td>Stores</td>
<td>60K</td>
</tr>
<tr>
<td>Loads</td>
<td>88K</td>
</tr>
<tr>
<td>L1 accesses</td>
<td>80861</td>
</tr>
<tr>
<td>L1 misses</td>
<td>7523</td>
</tr>
</tbody>
</table>

Table 1. Performance characteristics of a single transaction in Delaunay mesh generation

5. Transactional Memory Limitations

While transactions are an intuitive approach to the optimistic parallelization demands of the mesh generation algorithm, current implementations of transactional memory (both software and hardware) exhibit a few problems that may adversely affect the performance of the parallel Delaunay mesh generation algorithm. These two issues are scheduling of conflicting transactions and conservative interference detection.

5.1 Scheduling of Conflicting Transactions

Consider two concurrent cavity expansions whose cavities overlap. If these transactions attempt to commit simultaneously, it is possible for both to be rolled back [5]. This is because most realistic implementations of transactional memory do not commit the modified state of a transaction in a single step, so it is possible that both transactions attempt to update shared state in inconsistent ways. If both transactions are immediately re-executed, they are likely to conflict again. This type of livelock is a common issue with transactional memories. The traditional solution to this problem is some form of random back-off, to allow some forward progress. From an efficiency standpoint, this is not ideal; a better solution would be to abort one of the transactions and to use the now-idle resources to execute a different transaction. This solution avoids livelock while not incurring the overhead of back-off.

However, as we noted above, the isolation of concurrently executing transactions means we must select the elements to process outside of the transactions (otherwise we risk two transactions choosing to work on the same element). Unfortunately, this means that the default approach is to continue attempting to process a single element until the transaction successfully commits, precluding our use of a more efficient scheduling policy as described above.
One solution is to use a transactional memory system which provides user-specified abort handlers, such as LogTM [10]. Thus, on abort, rather than simply rolling back the transaction, one could re-enqueue the current element and dequeue a new element before restarting the transaction, thus ensuring that the next execution of the transaction will operate on a different element than the previous execution.

Open Nesting

A more general solution may be provided by the open nesting approach as discussed in [11]. Nested transactions that are performed in an “open” manner within a parent transaction are allowed to modify shared memory without waiting for the enclosing transaction to commit. Thus, it is possible for a transaction to make changes that are immediately visible to other transactions, rather than waiting until commit.

This would allow the operations on the worklist to appear within the transaction. We can execute the worklist dequeue to obtain an element to process as an open transaction. Because the open nested transaction modifies the worklist before the outer transaction commits, concurrent transactions will see the results of previous dequeues and hence choose distinct elements to work on.

Because shared memory is modified directly, it is necessary to provide “undo” actions to reverse the effects of an open nested transaction in the case of an abort. In this case, a worklist dequeue is undone with an enqueue. On abort, a transaction will re-enqueue the element it is working on, providing the desired rollback behavior as described above.

While this seems like a natural solution, there are many constraints that must be placed on open nested transactions for them to behave properly within the transactional model. Care must be taken that modifications can always be undone and that modifications will not affect the isolation and atomicity of the overall transaction. Current implementations of open nesting, such as ATOMOS [3], do not provide the necessary guarantees to ensure that open nesting is performed safely, choosing instead to relax the isolation guarantees provided by transactions. Thus, the burden falls upon the programmer to utilize open nesting correctly.

We feel that as currently realized, open nesting is overly complex; using it correctly requires the programmer to reason about concurrent effects and potential race conditions. Further investigation is warranted to design an open nesting system that provides the functionality we desire while maintaining the programmability of normal transactions.

5.2 Conservative Interference Detection

The second drawback of current transactional memories is that they are too conservative; they may detect interference and trigger unnecessary rollbacks. This is an odd charge to level against transactional memory, since one of the motivations for the transactional model is the conservative synchronization enforced by many locking schemes, an issue that transactional memories attempt to avoid.

Consider the example of insertion into a sorted linked list, as seen in Figure 8. Here we see two transactions attempting to insert new nodes into the list at different points. Ideally, these two insertions can proceed in parallel, as they will not interfere with one another. However, the first transaction, in traversing the list to reach the insertion point, has read locations that the second transaction will attempt to modify. A standard transactional memory will detect interference between these two transactions and eventually determine that the only safe course of action is serialization, despite the “high-level” independence of the two transactions.

At a high level, the problem is that there are certain invariants on the abstract data type that must be preserved for correct execution of the program. Ensuring that there are no conflicting reads and writes to the concrete data structures that implement the abstract data type is sufficient but not necessary to ensure that the high level invariants are respected. In the transactional code shown in Figure 7, two transactions interfere if and only if their cavities overlap. However, the transactional memory merely enforces that they do not perform incompatible reads and writes to memory locations of the concrete data structure, which might result in false positives which detect spurious interference.

Let us consider how these false positives could arise in the mesh generation problem. At the abstract level, we can treat the mesh as a graph. Each node of the graph represents an element in the mesh, and adjacency in the

---

**Figure 8.** Concurrent insertions into a list
graph models adjacency in the mesh (so each element has at most three neighbors). In this high level view of the structure, we see that we want to detect interference (and hence trigger a roll back) only if two transactions access the same nodes in the graph (e.g., one reads an element which is then removed from the graph by another transaction).

However, transactional memory does not detect interference at this high level. Rather, it focuses on conflicts that occur when accessing specific memory locations – a fundamentally low-level view of interference. Because transactional memory operates at a low level, the amount of interference detected is highly dependent on the concrete implementation of the mesh data structure. However, regardless of the concrete implementation, it is not possible to avoid false positives.

**Adjacency list:** The simplest implementation of a graph is as an adjacency list. The adjacency list maintains a map from each node in the graph to a list of its neighbors. If this structure is implemented using a standard library such as STL, it will be based around balanced-trees (the default STL map). Thus, any additions or removals of nodes from the graph will result in large portions of the data structures being accessed and modified (due to procedures such as tree rebalancing). Furthermore, because the edges are stored in an adjacency list, steps such as *expand* in the algorithm will require reading large portions of this data structure (to find neighbors). The upshot is that even if two transactions do not access the same elements of the mesh, they will access large parts of the mesh data structure, and are likely to cause low level interference, leading to a high number of false positives.

**Local edge information:** One technique that could be used to prevent some low level interference is moving the adjacency information from a global adjacency list down to the nodes themselves. Thus, each node maintains a list of its neighbors, but this information is not globally visible. Instead, the only globally visible portion of the graph is a “membership set” which keeps track of which nodes are in the graph. Thus, operations that require finding neighbors (such as *expand*) no longer require reading from a global structure. Although this implementation still suffers from false positives, they are less likely for two reasons. One is that two transactions that access different elements will not interfere in an adjacency list (since there isn’t one), instead only potentially interfering in the membership set. The second is that elements are only added and removed from the mesh at the end of a transaction, meaning that the potential interference expected by modifying the data structure is only “active” for a short period of time.

**Hash set:** A final optimization that may reduce the number of false positives is to implement the membership set with a hash set instead of an STL set (which uses a balanced tree). This will reduce interference as transactions will no longer access the structure extensively for operations such as tree rebalancing. However, even the use of a hash set does not eliminate the potential for false positives. Note that if two distinct elements hash to the same bucket, transactional memory will still detect interference even if the two transactions access disjoint sets of elements and therefore should not interfere. This problem is exacerbated because, in general, a transaction adds and removes several elements from the mesh (as it removes all the elements in the cavity and inserts all the newly created elements), increasing the likelihood of low level interference.

### High Level Transactional Memory

As we see, regardless of the concrete implementation of the mesh data structure, transactional memory still has the potential to detect irrelevant interference and hence trigger unnecessary roll backs.

We believe that what may be necessary is a transactional memory implementation that is aware of high level abstractions. Thus, to this implementation, it would be apparent that the insertion and removal of distinct elements from the mesh should not trigger interference, even if at the low level a typical transactional memory implementation would have detected interference. Implementing such a transactional memory may require significant work, as the transactional memory should ensure that the operations on the structure that do not trigger interference still complete correctly when executed concurrently.

### 6. Conclusion

We presented Delaunay mesh generation, an algorithm for producing guaranteed-quality meshes. We showed how this algorithm can be easily parallelized by processing multiple elements from the set concurrently. However, this approach requires that the cavities created during processing do not interfere with each other. We demonstrated how optimistic parallelism, and specifically the transactional model, provides a straightforward, easily implemented way to guarantee that the parallel execution is correct.

We also discussed shortcomings in current transactional memory implementations that may hinder parallel efficiency: inefficient handling of conflicting transactions
as well as overly conservative interference detection. We feel that addressing these issues is an important avenue of exploration in transactional memory research.

We believe there are many other algorithms that have the properties similar to those of the Delaunay mesh generation algorithm — a worklist structure with infrequent dependences between work units, which cannot be determined statically. One example, also from the same general domain, is advancing front mesh generation [9]. In this scheme, the mesh is “grown” inwards from the boundaries by selecting segments on the boundary and adding a new point on the interior to create a new element. Effectively, this requires placing all the segments of the boundary into a queue, and as segments are processed, updating the queue with the new boundary segments. Advancing front mesh generation can be parallelized in much the same way as Delaunay mesh generation, and the use of a similar transactional approach holds promise.

References


 Concurrent Cache-Oblivious B-Trees Using Transactional Memory

Bradley C. Kuszmaul  Jim Sukha

ABSTRACT
Cache-oblivious B-trees for data sets stored in external memory represent an application that can benefit from the use of transactional memory (TM), yet pose several challenges for existing TM implementations. Using TM, a programmer can modify a serial, in-memory cache-oblivious B-tree (CO B-tree) to support concurrent operations in a straightforward manner, by performing queries and updates as individual transactions. In this paper, we describe three obstacles that must be overcome, however, before one can implement an efficient external-memory concurrent CO B-tree.

First, CO B-trees must perform input/output (I/O) inside a transaction if the underlying data set is too large to fit in main memory. Many TM implementations, however, prohibit such transaction I/O. Second, a CO B-tree that operates on persistent data requires a TM system that supports durable transactions if the programmer wishes to be able to restore the data to a consistent state after a program crash. Finally, CO B-trees operations generate megalithic transactions, i.e., transactions that modify the entire data structure, because performance guarantees on CO B-trees are only amortized bounds. In most TM implementations, these transactions create a serial bottleneck because they conflict with all other concurrent transactions operating on the CO B-tree.

Of these three issues, we argue that a solution for the first two issues of transaction I/O and durability is to use a TM system that supports transactions on memory-mapped data. We demonstrate the feasibility of this approach by using LibXac, a library that supports memory-mapped transactions, to convert an existing serial implementation of a CO B-tree into a concurrent version with only a few hours of work. We believe this approach can be generalized, that memory-mapped transactions can be used for other applications that concurrently access data stored in external memory.

1. INTRODUCTION
Whereas most hardware and software transactional memory systems (e.g., [1, 2, 14–17, 19, 20, 23, 28]) implement atomicity, consistency, and isolation, but not durability (the so-called “ACID” properties [13]), we have developed a software transactional system that can provide full ACID properties for memory-mapped disk-resident data. This paper reports our experience using a transactional memory interface, with full ACID properties, to implement a cache-oblivious B-Tree.

Today, traditional B-trees [5, 10] are the dominant data structure for disk-resident data because they perform well in practice. In theory, traditional B-trees perform well in a performance model called the Disk-Access Machine (DAM) Model [4], an idealized two-level memory model in which all block transfers have unit cost, the block size is \(B\), and the main-memory size is \(M\). The choice of \(B\) defines the single granularity of optimization in the DAM model. For example, an optimized B-tree with fixed-sized keys has a branching factor of \(\Theta(B)\), and thus requires \(O(\log_B N)\) memory transfers for queries, which is optimal within the DAM model. The widespread use of B-trees suggests that the DAM model is used implicitly as a simplifying approximation for writing disk-intensive code.

It is difficult to choose the right value for \(B\), however. The block size could be set to correspond to the CPU’s cache line size (perhaps 64 bytes), to the disk’s advertised block size (perhaps 4096 bytes), or possibly some larger value, such as the average track size on disk (on the order of 1/4 megabyte for today’s disks). Ideally, a B-tree would simultaneously minimize the number of cache lines, the number of disk blocks, and the number of tracks accessed during a query.

One way to avoid this block-size tuning problem is to employ data structures that work well no matter what the block size is. A cache-oblivious data structure is a data structure in which the parameters of the memory hierarchy (such as the cache-line size, the cache size, the disk block size, or the main memory size) are not coded explicitly, nor are they discovered by an active tuning process. In contrast, a cache aware data structure knows about the memory hierarchy.

Theoretical developments on cache-oblivious data structures and algorithms have shown in principal how to achieve nearly optimal locality of reference simultaneously at every granularity. In the cache-oblivious model [12, 26], an alternative to the DAM model, one can prove results about unknown memory hierarchies and exploit data locality at every scale. The main idea of the cache-oblivious model is that if it can be proved that some algorithm performs a nearly optimal number of memory transfers in a two-level model with unknown parameters, then the algorithm also performs a nearly optimal number of memory transfers on any unknown, multilevel memory hierarchy.

Our study focuses on the cache-oblivious B-tree (CO B-tree) data structure described in [6], and how to use a transactional memory interface to support concurrent queries and updates on the tree. Transactional memory is well-suited for programming a concurrent CO B-tree, since, arguably, a serial CO B-tree is already more complex to implement than a traditional serial B-tree. The CO B-tree is representative of the kind of data structure that we can implement with transactional memory: a data structure that may be more complicated, but asymptotically more efficient than the traditional alternative.

Furthermore, the cache-oblivious nature of the CO B-tree makes it difficult to parallelize the search tree operations using traditional methods for mutual exclusion. In a normal B-tree, the block size \(B\) presents a natural granularity for locking. For a CO B-tree that has no tunable parameters to set, however, the locking granularity would also need to be specified at an abstract level. Transactional memory interacts synergistically with cache-oblivious data structures because transactions allow the programmer to specify parallelism in an implementation-independent way.

A natural approach to programming a concurrent CO B-tree is to convert every query or update operation of a serial CO B-tree into its own transaction. We encountered three obstacles to making this
strategy work.

The first obstacle for the simple concurrent CO B-tree is transaction I/O. When the entire data set no longer fits into main memory, a query or update transaction may need to perform I/O to retrieve data from disk. If the programmer is working in a system with two levels of storage, then the programmer must make explicit I/O calls inside the transaction, typically through a buffer management subsystem, to bring a new page into memory and kick an existing page out. Buffer pool management adds another layer of complication to an already complex concurrent CO B-tree implementation. Furthermore, calls to the buffer management system should not be included as part of the transaction, since we can not easily undo I/O operations. Many proposed TM systems have not specified a programming interface or semantics for I/O operations that occur inside a transaction.

The second obstacle is transaction durability. For a CO B-tree that stores persistent data, the user would like the guarantee that the stored data will not be corrupted if the program accessing disk crashes. Database systems usually support durable transactions by updating a log on disk after every transaction commit. The log contains enough information to restore persistent data to a consistent state. Since TM systems already track the changes made by a transaction, support for durable transactions would be a natural extension.

The final obstacle with using TM on a CO B-tree is that the CO B-tree sometimes generates what we call "megalithic transactions." A megalithic transaction is one that modifies a huge amount of state, effectively serializing performance. For the CO B-tree, there are some updates that must rebuild the entire data structure, producing a megalithic transaction. A megalithic transaction represents an extreme case because it runs for a long time and conflicts with all other transactions.

To address the first two obstacles of transaction I/O and transaction durability, we use LibXac, a page-based software transactional memory system that we developed, to implement the CO B-tree. LibXac supports transactions on memory-mapped data, allowing any application to concurrently access data from the same file on disk without explicit locking or I/O operations. We believe LibXac’s interface is useful for generic concurrent external-memory data structures, since the issues of transaction I/O and durability are not specific to the CO B-tree.

More generally, external-memory data structures are a good match for a software transactional memory system (STM), because any runtime overheads of the STM can be amortized against the cost of disk I/O. Even if a memory access using an STM system costs an order of magnitude more than a normal memory access this overhead is small compared to the cost of a disk access for moderate-size transactions.

This paper describes the issues we encountered when using a transactional-memory interface to implement a concurrent cache-oblivious B-tree. Section 2 describes LibXac, our page-based software transactional memory implementation. Section 3 describes our experience implementing the CO B-tree, and explains how we used LibXac to address the first issues of transaction I/O and durability. Section 4 describes the CO B-tree structure in greater detail, explains how update operations can generate megalithic transactions, and discusses possible solutions for this problem. Finally, Section 5 concludes with a description of related work and directions for future work.

2. THE LibXac TM SYSTEM

We developed LibXac, a prototype page-based software transactional memory implementation that addresses the two problems of transaction I/O and durability. LibXac provides support for durable memory-mapped transactions, allowing programmers to write code that operates on persistent data as though it were stored in normal memory. In this section, we present LibXac’s programming interface and an overview of its implementation.

Programming Interface

In many operating systems, different processes can share memory by using the system call mmap to memory-map the same file in shared mode. Programmers must still use locks or other synchronization primitives to eliminate data races, however, since this mechanism does not provide any concurrency control. Using LibXac, programmers transactionally memory-map a file using xMmap, and prevent data races by specifying transactions on the mapped region.

Figure 1 illustrates LibXac’s basic features with two programs that access the same data concurrently. Both programs modify the first 4-byte integer in file data.db. The program on the left increments the integer, and the program on the right decrements it. When both programs run concurrently, the net effect is to leave the integer unchanged. Without some sort of concurrency control however, a race condition could cause the data to be corrupted.

Line 1 initializes LibXac, specifying the directory where LibXac will store its log files. LibXac will store enough information to guarantee that the data in the file can be restored to a consistent state even if the program crashes during execution. Line 9 shuts down LibXac.

Line 2 opens a shared memory segment by using xMmap() to memory-map a particular file. The xMmap() function takes a filename and number of bytes to map as arguments, and returns a pointer to the beginning of the shared region. LibXac allows concurrency at a page-level granularity, and requires that specified length be a multiple of the page size.

Lines 3-8 contain the actual transaction. Line 5 is the actual body of the transaction.

Transactions are delimited by xbegin() and xend() function calls. The xend() function returns a status code that specifies whether the transaction was committed or aborted. If the transaction commits, then the while loop stops executing. Otherwise the code invokes the backoff() at Line 7, and then the while loop tries to run the transaction again. The application programmer can provide whatever implementation of the backoff() function they wish (for example, it might employ randomized exponential backoff [22]).

When using LibXac, the control flow for a program always proceeds through from xbegin to xend, even if the transaction is aborted. It is the programmer’s responsibility to ensure that xbegin() and xend() function calls are properly paired, so that control flow does not jump out of the transaction without first executing xend(). LibXac also provides an xValidate function that the programmer can call in the middle of a transaction to check whether a transaction will need to abort because of a transaction conflict. The programmer can then insert code to stop executing a transaction that will not commit.

Nested transactions are automatically subsumed into the outermost transaction. This mechanism allows programmers to call functions inside a transaction that may themselves contain a transaction. An xend() call nested inside another transaction always succeeds, since the subtransac- tion successfully “commits” with respect to the outer transaction.

LibXac can also be configured to support transactions that are not durable.
Memory Model

LibXac’s memory model provides the standard transactional correctness condition, that transactions are *serializable*. In fact, LibXac makes a stronger guarantee, that even transactions that end up aborting always see a consistent view of memory. Because LibXac employs a variation of a multiversion concurrency control algorithm [8], it can guarantee that an aborted transaction always sees a consistent view of the shared memory segment during execution, even if it conflicted with other transactions. Said differently, the only distinction between committed and aborted transactions is that a committed transaction atomically makes permanent changes that are visible to other transactions, whereas an aborted transaction atomically makes temporary changes that are never seen by other transactions.

When a transaction is aborted, only changes to the shared segment roll back however. Changes to local variables or other memory remain, allowing programmers to retain information between different attempts to execute a transaction. See [30] for more details LibXac’s memory model.

Implementation

LibXac is implemented on Linux, without any modifications to the kernel or special operating system support. See [30] for a more thorough description of the implementation.

At a high-level LibXac executes transaction as follows.

1. When a transaction begins, the protection is set to “no access allowed” on the entire memory-mapped region.

2. The transaction’s first attempt at reading (or writing) a page causes a segmentation fault on that page. The SEGFAULT handler installed by LibXac maps the appropriate version of the page into the transaction’s address space as read-only data. LibXac relies on the ability to change the memory mapping of a particular virtual page, which Linux supports, but some operating systems may not.

3. When the transaction attempts to write to a page for the first time, the system must handle a second segmentation fault (because in Linux there is no easy way to distinguish between reads and writes.) LibXac generates a new version of the page and maps that version into the address space with read-write permissions.

4. When the *xend()* function is invoked, the runtime determines whether the transaction can commit. If so, then a log is generated of all the pages that were modified. That log includes a copy of both the old and new version of each page, as well as a commit record. The memory map is then reset to no-access.

5. Each transaction that modifies a page creates a new version of the page. Eventually, after the runtime determines that no transaction will ever need the version that is actually stored in the original file, it copies the youngest committed version of the page it can back to the original file. Thus, if the transaction processing becomes quiescent, the original file will contain the final version of the database.

6. After a committed version of a page is copied back into the original file, it eventually gets written to disk, either implicitly by the operating system’s page eviction policy, or explicitly by a checkpoint operation.

During a transaction commit, our implementation performs the synchronous disk write required for logging, but we have not yet implemented checkpointing or a recovery program.

3. TRANSACTION I/O AND DURABILITY

In this section, we explain how transactional memory-mapping solves the issues of transaction I/O and durability, and describe our experience using LibXac to implement a CO B-tree.

A typical approach to managing disk-resident data is to program in a two-level store model with explicit disk I/O, managing a cache explicitly. Programming a two-level store is laborious, however, because the programmer must constantly translate from disk addresses to memory addresses. Moreover, programming cache-oblivious file structures using a two-level store is problematic, because the nature of cache-obliviousness precludes defining a particular block size for reading and writing.

We began with a serial implementation of a CO B-tree [18] that employs memory mapping, thereby handling the I/O issue automatically in the serial case. The Unix *mmap* function call provides the illusion of a single-level store to the programmer. The *mmap* function maps the disk-resident data into the virtual address space of the program, providing us with a large array of bytes into which we embed our data structure. Thus, the CO B-tree code can rely on the underlying operating system to handle I/O instead of calling explicit I/O operations.

The fact that LibXac’s interface is based on memory-mapping also solves the problem of transaction I/O for a concurrent CO B-tree. The only additional complication concurrency introduces is the possibility of transactions with I/O aborting. Since LibXac only tracks operations on the mapped shared memory segment, however, the I/O operations are automatically excluded from the transaction. Thus, the application programmer does not need to worry about a transaction being aborted in the middle of an I/O operation.

LibXac also satisfies the requirement for transaction durability because it logs the contents of pages that a transactions modifies and synchronously writes the commit record to disk when a transaction commits. Our prototype system does not have recovery implemented, but it saves enough information to the log to allow a recovery program to restore the *xmap*’ed file to a consistent state.

Because of the simplicity of LibXac’s programming interface, starting with LibXac and a serial memory-mapped CO B-tree, we were able to easily create a CO B-tree that supports concurrent queries and updates with only a few hours of work. To demonstrate that our approach is practical, we ran a simple experiment performing 100,000 inserts into various implementations of a concurrent B-tree. Each insertion is performed as a durable transaction,

```c
0 /* Increment the 1st integer in data.db */
1 xInit("/logs");
2 memptr = (int*)xMmap("data.db", 4096);
3 while (1) {
4     xbegin();
5     memptr[0] --;
6     if (xend() == COMMITTED) break;
7     backoff();
8 }
9 xShutdown();
10 /* Decrement the 1st integer in data.db */
11 xInit("/logs");
12 memptr = (int*)xMmap("data.db", 4096);
13 while (1) {
14     xbegin();
15     memptr[0] --;
16     if (xend() == COMMITTED) break;
17     backoff();
18 }
19 xShutdown();
20
Figure 1: Two programs that access shared data concurrently, using LibXac transactions.
```
with a randomly chosen 8-byte integer as the key. In a run with \( P \) processes, each process performed either \( \lfloor \frac{100000}{P} \rfloor \) or \( \lceil \frac{100000}{P} \rceil + 1 \) insertions.

We ran this test in three different environments: in a normal B\(^+\)-tree implemented using LibXac, a CO B-tree using LibXac, and on a Berkeley DB [29] B-tree. The block size for both B-trees is 4096 bytes, and the keys of all B-trees are actually padded to 512 bytes.

![Figure 2: 100,000 Inserts on concurrent B-trees performed by multiple processes.](image)

Figure 2 presents preliminary performance results. Note that the poor performance of the two B-trees relative to the CO B-tree is likely due to the fact that \( B \) was not properly tuned. In practice, the effective value of \( B \) should be much larger than the default 4K page size specified by Linux.

Since each transaction is dominated by the cost of the synchronous disk write during commit. Even though our experiment is run on a machine with only 4 processors and a single disk, all three B-trees are able to achieve speedup using multiple processes because they all implement group commit [11].

We do not interpret these results as evidence that one system (LibXac or Berkeley DB) or data structure (CO B-tree vs. normal B-tree) necessarily outperforms the other. Our claim is only that a system that supports memory-mapped transactions and which is competitive with the traditional alternatives can be feasible to implement in practice.

4. MEGALITHIC TRANSACTIONS

Of the three problems we described for a CO B-tree implemented using transactional memory, the problem of megalithic transactions is the most troublesome. In this section, we describe the CO B-tree data structure of [6] in more detail, explain how update operations give rise to megalithic transactions, and briefly comment on ways to address this issue.

Rather than explaining the entire CO B-tree data structure, we focus on a piece of the data structure, the packed memory array (PMA), that illustrates the problems we faced. A PMA is an array of size \( O(N) \), which dynamically maintains \( N \) values in sorted order. The values are kept approximately evenly spaced, with small gaps to allow insertions without having to move too many elements on average. Occasionally a large amount of data needs to be moved, but if the gaps are managed properly [6], the average cost of insertions remains small. The CO B-tree stores its values in a PMA, and uses a static cache-oblivious search tree as an index into the PMA. Thus, an insertion into the CO B-tree involves an insertion into the PMA and an update of the index.

We only sketch the algorithm for insertion into the PMA (see [6] for more details). To insert an element into a PMA, if there is a gap between the inserted element’s neighbors, then we insert the element into a gap position. Otherwise, we look for a neighborhood around the insertion point that has low density, that is, look for a subarray that is not storing too many data elements. Given a sufficiently sparse neighborhood, we rebalance the neighborhood, i.e., space out the the elements evenly. In order to get rebalances to run quickly on average, one must apply stricter density rules for larger neighborhoods. The idea is that because rebalancing a large neighborhood is expensive, after the rebalance we need a greater return, i.e., a sparser final neighborhood. Once the neighborhood corresponding to the entire PMA is above the maximum density threshold, we double the size of the array and rebuild the entire PMA, and thus the entire CO B-tree.

For the CO B-tree, updates that must rebalance the entire PMA, or at least a large fraction of it, produce what we call a megalithic transaction. A megalithic transaction is one that modifies a huge amount of state, effectively serializing performance. A megalithic transaction represents the extreme case because it is a large transaction that conflicts with all other transactions. Thus, some contention management strategy [16] is needed to avoid livelock when transactions conflict.

An operation that modifies all of memory does not necessarily cause performance problems in the serial case, but it does cause problems in the parallel case. For the serial case, the CO B-tree we used has good amortized performance. Although the average cost of an update is small, some updates are expensive. In a serial data structure, the cost of the expensive operation can be amortized against previous operations. In the parallel implementation, the expensive updates cause performance problems because they increase the critical path of the program, reducing the average parallelism.

If updates are infrequent compared to searches, their performance impact can be mitigated by using a multiversion concurrency control [8], in which read-only transactions never conflict with other transactions. Our LibXac prototype provides multiversion concurrency control and allows the user to specify special read-only transactions that will always succeed. But although we have not confirmed this fact experimentally, this particular CO B-tree data structure appears to have limitations in the parallelism of its update operations.

The best solution for a megalithic transaction is to eliminate it altogether, through algorithmic cleverness. Ideally, deamortizing the operations on a CO B-tree would make the footprint of every update transaction small, thereby eliminating any megalithic transactions. Another approach may be to find a way to split up the large rebalancing operations of the CO B-tree’s PMA into multiple transactions. The worst-case time for a single update may still be large in this case, but that update would at least not block all other concurrent transactions from committing.

Since improving or deamortizing a data structure often make the structure more complicated, transactional memory can help us by simplifying the implementation. In general, programming with a transactional memory interface instead of with explicit I/O makes it plausible that a programmer could implement an even more sophisticated data structure such as a cache-oblivious lookahead array [25] (which provides dramatic performance improvements for workloads with many more insertions than searches) or a cache-
oblivious string B-tree [7] (which can handle huge keys efficiently).

Another approach to working around megalithic transactions is to use some sort of loophole in the strict transactional semantics, such as the release mechanism [16] or so-called open-transactions [24].

5. RELATED AND FUTURE WORK

Using LibXac, we are able to overcome the obstacles of transactional I/O and durability for a concurrent CO B-tree implemented using transactional memory.

Our experimental results suggest that LibXac provides acceptable performance for durable transactions. We would pay a high penalty in performance, however, if we were to use LibXac for the sort of non-durable transactions that many transactional-memory systems provide. The LibXac runtime must handle a SEGFAULT and make an mmap system call every time a transaction touches a new page. This method for access detection introduces a per-page overhead that on some systems can be 10 µs or more. With operating system support to speed up SEGFAULT handlers, or introduction of a system call that reports which pages a transaction has read or written, one might be able to use a system such as LibXac for efficient non-durable transactions.

Without operating system or compiler support, implementing durable transactions on memory-mapped data has been viewed as an open research problem [3, 9]. The problem is that the operating system may write memory-mapped data back to disk at any time. The correctness of a transaction logging scheme usually requires that the data be written back to disk only after the log has been written, but with memory mapping, the operating system may write the data back too soon. A relatively slow implementation of portable transactions for single-level stores that incurs a synchronous disk write after every page accessed by a transaction is described by [21]. Recoverable virtual memory [27] supports durable transactions, but the interface requires programmers to explicitly identify the shared memory being accessed by a transaction. LibXac’s approach of remapping pages as they are modified is, to our knowledge, the first portable and efficient solution to durable transactions on memory mapped data.

6. REFERENCES


The Linux Kernel: A Challenging Workload for Transactional Memory

Hany E. Ramadan  Christopher J. Rossbach  Emmett Witchel
University of Texas at Austin
{ramadan,rossbach,witchel}@cs.utexas.edu

1. Introduction

The Linux operating system kernel [4] is a large, mature, freely available, and well-tuned concurrent program. As such it is an ideal workload for a transactional memory hardware design.

Operating systems need transactional memory for performance scalability, to help maintainability, and to provide services related to transactions to user programs. Most general purpose computing platforms run operating systems, and OS services must be scalable or applications will see the OS as a scalability bottleneck. The OS should not interfere with applications making use of the increased number of processing contexts available on modern CPUs. There has been enormous effort over the past decade to make the OS scalable, and the result has been increased code complexity that is starting to threaten continued innovation. For instance, mm/filesmap.c has 50 lines of comments detailing lock ordering constraints. Finally, if the OS is to provide transaction-related services (such as supporting user-level transactions across a context switch), it could probably do so most naturally if the OS itself were implemented with transactions.

This paper raises issues about how an OS can take advantage of a transactional memory hardware system. While there have been trace-based studies of OSes on transactional hardware [1], and designs for virtualizable transactions [22], these have ignored many interesting issues to make running an OS on transactional hardware truly practical.

The contributions of this paper include the following observations.

• The most natural way to handle interrupts requires that a single thread of control can have multiple concurrently active transactions. Existing models do not accommodate this approach; we propose a new model called transaction stacking to enable this functionality. (Section 2)

• Conflict management, the mechanism that determines which transaction must restart when two transactions conflict, is essential for performance. The OS has several locking mechanisms that provide different priorities for readers vs. writers, and these hints should be communicated to the hardware. (Section 3)

• While transactions promise to simplify the programming model for concurrent programs, there is considerable complexity in the Linux kernel related to concurrency that might or might not be helped by transactions. Examples include: per-CPU data structures, disabling interrupts, and blocking operations. (Section 4)

• We present preliminary results of implementing a hardware transactional memory model in a machine simulator, and booting a transactionalized version of the Linux kernel on the simulator. (Section 6)

2. Interrupts

Interrupts generally refer to asynchronous events, such as the countdown timer expiring, or the disk device signaling the completion of a data transfer, while exceptions refer to synchronous events like system calls and invalid opcodes. Interrupts start the OS executing from
a hardware-defined location in privileged code. Interrupts can occur during the execution of the OS itself, or during execution of user code. While an interrupt is being handled, another one may be raised, even by the same device. In order to ensure forward progress interrupt handlers mask interrupts that are of equal or lower priority to the interrupt being handled.

Interrupts occur much more often than context switches—timer interrupts can fire every 1 millisecond, whereas a typical time slice for a Linux process is 100 milliseconds. Moreover, with processor speeds growing more slowly, I/O devices are poised to narrow their performance gap (e.g., through multi-gigabit network interfaces), maintaining the pressure for frequent interrupts.

User-mode programs also experience asynchronous control flow, primarily via signal handlers, which share similar issues as interrupts. However, interrupts are far more frequent in the OS than signals are at user level.

### 2.1 Interrupts and transactions

We believe that the best way to integrate interrupt handling with transactional memory is to allow a single thread of control to have multiple active, but independent, transactions at once. We call this stacking, which is distinct from nesting because the transactions are independent. We discuss stacking in the next section.

This section considers possible OS strategies for integrating interrupt handling with transactions and demonstrates that support for stacking is necessary. Consider the arrival of an interrupt while the kernel is executing. The same thread (the OS on processor N) executes the interrupt handler in the same address space. What should the system do? Possibilities include:

- Make a rule against two active transactions in interrupt handlers. If interrupt handlers cannot actually use transactions, it is possible to simply execute the handler code. If the handler performs a memory operation that conflicts with the interrupted transaction, the hardware would abort the paused transaction after the handler returns. However, denying transactions to interrupt handlers denies an important tool for synchronization to the part of the OS that needs it the most.
- Abort the first transaction when the second one starts. This would allow the interrupting event-handler to use memory transactions. The aborted transaction must be re-executed once the event handler finishes. However, this approach aborts all interrupted transactions, whether or not there is a conflict with the interrupt handler’s transaction. This approach aborts many more transactions than necessary.
- Nest the transactions [19]. The problem with nesting the transactions is that there is typically no meaningful relationship between the interrupted transaction and the transactions which the interrupt handler creates. Flattening or closed nesting is not an option. If the outer transaction fails, flattening would fail the inner transaction and hence cancel the effect of receiving the interrupt. Open nesting, which would allow a parent abort to perform compensatory actions, would mean that every interrupt handler would need code to undo its effects. Were such code possible, it would be more complicated than the locking that transactions are intended to replace.
- Treat the interrupt as a context switch. Recent proposals for transactional hardware [1, 22] have included the ability for a thread’s transaction to survive across a context switch. These systems maintain overflow state on a per-process basis, enabling a transaction to be in a “swapped out” state. Virtualizable transactions [22] associate a transaction data structure with each address space. This allows a thread transitioning from user to kernel code to flush its user-level transaction state to memory while it executes kernel-level transactions. The memory flush might hurt performance, but a thread can have two active transactions, one in each address space. If an interrupt arrives while the kernel is executing, then not even virtualizable transactions can help because the thread and address space are the same for both active transactions. Adding multiple context identifiers to the kernel address space to enable interrupts to be treated as context switches does not seem worth the hardware investment.

### 2.2 Stacked transactions

We suggest a mechanism for allowing interrupt handlers to use transactional memory called stacked transactions. Stacked transactions allows multiple independent concurrent transactions in a single thread. Note that the concepts of nesting and stacking are orthogonal; one can have a stacked, nested transaction. “Stacked” is borrowed from the terminology that interrupt handlers are “stacked” on top of each other.

The mechanisms developed for allowing transactions to survive context switches can be used to implement the “stacked” transaction model, however these mechanisms were not developed with stacking in mind.
so it is likely that more efficient designs are possible. Because interrupt handlers usually execute with interrupts disabled, they tend to be short. It should be possible to virtualize stacked transactions with mechanisms that are less expensive than those required for virtualization of transactions across context switches.

### 2.3 Issues for stacked transactions

The ability of a single thread of control to own multiple outstanding transactions has the potential to affect various aspects of transactional memory systems. We investigated two issues in our implementation effort: conflict management, and stack memory. Conflict management policies, the mechanisms that determine which transaction “wins” if two conflict, need to be sensitive to whether conflicting transactions are stacked. Assume the OS is executing transaction A and receives an interrupt and begins executing transaction B. If A and B conflict, the system must abort A, otherwise the system will livelock.

A second issue arises under the following circumstances: while transaction A is active, it makes a function call that returns, but some stack\(^1\) memory values modified by the call conflict with those modified by the interrupt handler\(^2\). Reuse of the stack memory creates an artificial conflict between otherwise independent transactions. This is illustrated in Figure 1. If the caller started the transaction and then called the callee\(^3\), the callee’s stack frame becomes part of the callers transaction state. This can cause a spurious conflict with an interrupt handler for an interrupt that arrives after the callee has returned. To avoid this problem software could drop the memory stack locations from a transaction’s set when the function returns, or the hardware might exclude these ranges from transactional sets in the first place.

A correctness issue arises if transaction A starts in a function that returns before the interrupt handler runs, as shown in Figure 1. The non-transactional writes of the interrupt handler change the state of the stack locations used by transaction A. When the handler returns, transaction A aborts, restoring its program counter and stack pointer to the values they had at the start of the transaction. Unfortunately the stack frame that was active when the transaction started has been overwritten by the interrupt handler. This problem is tricky to solve. Perhaps the top of the stack becomes part of the state checkpointed by the hardware, and is restored on a transaction retry.

### 3. Conflict management hints are essential

Transactional hardware will need to accept programmer hints for conflict resolution. For instance, an argument to `xbegin`, the instruction that begins a transaction, might specify whether to favor readers or writers. OS performance might require several shades of favoritism, as reader/writer spin locks naturally favor readers, but read-copy-update (RCU)\(^2\) data structures favor readers even more heavily. Other forms of favoritism, for instance a low priority transaction that defers to most other transactions, should be investigated. Kernel developers have encoded rich information about how synchronization conflicts should be resolved, and transactional synchronization would disregard that information at peril of performance.

Consider seqlocks and RCU data structures: seqlocks are designed to favor writers, while RCU data structures favor readers. Seqlocks are similar to reader/writer spinlocks, but they give higher priority to the writer. Writers may always proceed (though only one writer is allowed at a time), while readers may have

---

\(^{1}\)Note that this refers to a thread’s memory stack, not to stacked transactions.

\(^{2}\)This is an issue regardless of whether the interrupt handlers uses a transaction or not.

\(^{3}\)In the figure the transaction starts in the callee.
to retry their operations. RCU data structures prioritize readers by avoiding reader locks for a restricted class of data structures (dynamically allocated data structures that are accessed by pointers). Writers must copy the object they wish to modify, and then atomically replace the old object with the new. Writer code can have locks and might require data structure redesign. Readers cannot sleep or be preempted.

The need for sophisticated hardware contention management is pressing in the OS because real contention can be the common case for some workloads. For instance, in our experiments we were able to induce many real data conflicts in Linux’s directory entry cache (dcache) code by doing simultaneous reads and updates within a directory. Transactions are optimistic and therefore are most effective when real contention is rare. It is likely that conflict management for transactions in the OS will require some adaptation so the system does not become unresponsive when the real conflict rate spikes. During such activity conservative locking is the most effective strategy.

4. Will transactional memory simplify programming?

A major benefit of transactional memory is that it simplifies reasoning about concurrent programs. The problem with this argument for Linux is that there is considerable complexity in the kernel to deal with synchronization and coordination that is not easily expressed with transactional semantics. This section discusses synchronization primitives within Linux that cannot or perhaps should not be replaced by transactional memory: per-CPU data structures and blocking primitives (semaphores, completions, and mutexes).

4.1 Per-CPU data structures

Separating out state that does not need to be shared across processors is good design practice. Modern operating systems formalize this with the notion of per-CPU variables and data structures. Per-CPU data structures do not need to be protected against access by any other processor. Is eliminating cross-processor synchronization a worthwhile complication to the programming model? Should per-CPU data structures be turned into transactions? Doing so would keep the programming model uniform, but might harm performance. Can the transactional models leverage the fact that a variable is guaranteed not to be accessed from another processor?

Per-CPU variables form a building block for complicated code. For example, the Linux kernel slab memory allocator [3] uses per-CPU variables to implement a shared heap. The initial version of the slab memory allocator (slab.c) in the Linux 2.2 kernel was roughly 2,005 lines of code (2.2.26). It increased to 3,070 lines of code for the 2.6 kernel (2.6.11)4. Linux maintainers note that “many of the changes in the slab allocator for 2.6 are . . . related to the reduction of lock contention.” [24].

The OS disables interrupts to protect per-CPU data structures from concurrent access by threads on the same processor. Transactions can provide isolation between threads on the same CPU in simple cases, but more research is needed to determine whether transactions can eliminate the need for most of this type of interrupt disabling.

4.2 Blocking operations

There is ongoing research on integration of blocking operations with a transactional model [23], for instance the transactional extensions to Concurrent Haskell [8] have introduced modular blocking primitives that monitor a transaction’s working set. The Linux kernel supports three different blocking synchronization primitives (semaphores, completions and mutexes), all optimized for different environmental assumptions.

Semaphores are objects that allow a certain number of waiters (usually one) into a critical section. Waiters are descheduled and placed on a queue, where they are awakened by a thread releasing the semaphore. For instance, processes queue themselves waiting for console access if they cannot get immediate access. Completions are a type of semaphore that avoid a race condition on a dynamically allocated semaphore. Mutexes [18] are a smaller, faster, binary-only semaphore with more restricted use than semaphores (they were introduced in Linux 2.6.16).

Blocking primitives raise the following research questions.

• If the latency of a blocking operation is dominated by the wait, is it necessary to optimize the operation? Maybe blocking operations are fine the way they are implemented because threads spend much more

4In the most recent version of the kernel (2.6.16.1), the code size has increased to 3,863 lines, primarily to support NUMA.
time waiting for a resource to become available than they do queuing themselves for the resource.

• If transactional primitives can reduce the instruction count to grab or release a blocking object, how much does that help performance and scalability? Maybe transactions play a useful role in the implementation of blocking primitives.

• Blocking primitives can be used in complicated ways. The semaphore that protects the memory mapping data structures is tested during the frequently executed page fault handling path. Different processing happens during a page fault if the semaphore is held or not. Would a reimplementation of the semaphore need to support this kind of operation?

5. I/O in transactions

Transactions must be restartable, so most proposals disallow I/O during a transaction. Our experiments revealed that Linux often performs I/O with spin locks held, thwarting an easy conversion of spin locks to use transactions. About one-third of Linux’s spin locks had I/O performed at some point while they were locked. Some locks are held for long periods of time during significant I/O (e.g. the real-time clock lock during boot).

We did observe that many I/O operations performed with spin locks held can be correctly re-executed with possibly small performance consequences. For instance, inter-processor interrupts (IPIs) are used to do system-wide TLB invalidations. Invalidating TLBs multiple times does not affect correctness, so it would be possible to include TLB shootdowns within a transaction, even though the transaction performs I/O. The performance consequences need to be investigated.

6. Preliminary Results

We have early results from implementing a generic hardware transactional memory model in the Simics [15] machine simulation framework (version 3.0.10). Our model implements stacked transactions, as described in Section 2.2, so that interrupt handlers are able to use transactions. We replaced the majority of spin locks in the Linux kernel, version 2.6.16.1\(^5\), with transactions. Here we discuss preliminary results using system boot as the workload.

\(^5\)The “.1” release came less than a week after the 2.6.16 release, and fixes a dead-lock introduced in the kernel scheduler, among other things.

Lock acquisition is translated to a begin transaction instruction, and the lock release is translated to an end transaction instruction. We could not replace every spin lock with a transaction. The most common problem was if a spin lock is ever held while I/O is performed. In that case, we conservatively do not convert it to use transactions. Of the 1,437 calls to spin_lock in Linux, about two-thirds are for locks that are never held during I/O in the workloads we executed.

We ran experiments with 2, 4, 6, and 8 simulated processors. Our simple performance model assumes 1 instruction per cycle, and infinitely fast devices. The memory hierarchy has a two-level cache per processor, with split instruction and data caches at the L1 level and a unified L2. The L1 caches are each 16Kb, 4-way associative, with 64-byte cache lines, assuming a 1-cycle cache hit and a 16-cycle cache miss penalty. The L2 caches are 4Mb, 8-way associative, with 64-byte cache lines and a 200 cycle miss penalty to main memory. The L2’s communicate using a MESI snooping protocol, and the main memory is a single shared 256 MB memory.

Figure 2 compares normalized boot times for the kernel using traditional spinlocks, and transactions. The transactionalized kernel shows a modest performance gain of about 2%. These results, while preliminary, are at least encouraging.

7. Related work

Lamport was among the first to propose that concurrent reading and writing of data need not require locks [14]. Notions of optimistic concurrency control
first appeared in the database domain [13], but did not gain wide acceptance in the database community [17].

Herlihy introduced the concepts of lock-free, wait-free and obstruction-free synchronization[9, 10], while transactional memory as a programming concept has its roots in [12, 11].

Among more recent research on hardware transactional memory (HTM) is Speculative Lock Elision [20, 21], which implements atomicity with the cache and speculatively identifies locks, and Transactional Coherence and Consistency [7, 6], wherein all computation is transactionalized. Unbounded Transactional Memory [1] and Virtual Transactional Memory [22] have addressed issues of virtualization and providing the programmer with freedom from platform-specificity and resource limitations.

Operating systems that make heavy use of non-blocking primitives include Synthesis [16] and the Cache Kernel [5].

References

Early Release: Friend or Foe?

Travis Skare and Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{travissk, kozyraki}@stanford.edu

1 Introduction

Transactional Memory (TM) [9] has the potential to simplify concurrency management by supporting parallel tasks (transactions) that appear to execute atomically and in isolation. There is already a significant body of work on programming language constructs for transactional memory [7, 6, 5, 3, 1, 2]. Nevertheless, there still exists little consensus on several constructs, particularly those motivated by performance optimizations.

In this paper, we study a set of data structure algorithms to evaluate the ease-of-use and performance benefits of the early release (ER) construct [8, 5]. Early release allows a transaction to remove a data address from its transactional read-set long before it commits. Once an address has been released, other transactions can write to this address without generating a conflict with the releasing transaction. The programmer or a compiler must guarantee that early release of an address is safe: removing the address from the read-set should not violate the overall application atomicity and consistency. The tradeoff with early release is obvious: on one hand, removing addresses from the read-set reduces the probability of conflicts that incur expensive long stalls or rollbacks. On the other hand, there is an additional burden to guarantee that early release is safe for a particular address, regardless of any other code that may be executing in parallel. Therefore, if a programmer manually applies early release, she must be extremely careful about when, where, and with which address early release is used.

Previous studies have used a single data structure algorithm (linked list) to conclude that various forms of early release can be a particularly useful programming construct for performance optimization of transactional programs [8, 4]. In this paper, we study five data structure algorithms and rewrite their transactional memory versions to use early release. Our observations are as follows: from the point of view of performance, early release provides a significant benefit only for highly sequential data structures such as a linked list or array-based heap. For concurrent data structures such as hash tables and trees (AVL trees and B-trees), there is no significant performance advantage from early release, even for write-intensive workloads with significant rollback penalties. From the ease-of-use point of view, we demonstrate with code examples that the complexity of manually using early release is often similar to that of using fine-grain locks. Therefore, unlike previous work, our results suggest that early release is not a particularly useful construct for user-level programming with transactional memory systems. Of course, early release may still be useful to an optimizing compiler that is able to automatically identify when it is safe and profitable to use in transactional code.

2 Methodology

We ran transaction-based code on an execution-driven simulator for a CMP that follows the TCC architecture [11]. TCC provides transactional memory using lazy conflict detection to provide non-blocking guarantees. Read-set and write-set tracking is at word granularity. The CMP includes up to 32 cores with private L1 caches (32 KBytes, 1-cycle access) and private L2 caches (256 KBytes, 12-cycle access). The read-set and write-set of the transactions in the studied algorithms fit in the processor caches, hence there is no overhead for overflows and virtualization. The processors communicate over a 16-byte, split-transaction bus. All non-memory instructions in our simulator have a CPI of one, but we model all details in the memory hierarchy for loads and stores, including inter-processor communication.

We added an early release instruction to the base TCC model. The instruction takes a word address and removes it from the transaction read-set. If the word is in the L1 cache, the early release instruction executes in one clock cycle, otherwise it
takes 12 cycles. TCC tracks read-sets at word granularity which helps with the ER implementation. However, early release is difficult to implement consistently in other hardware TM systems that use cache line granularity. In such systems, since an early release instruction provides a word address, it is not safe to release the entire cache line. Alternatively, we can define an early release instruction that releases an address range. Since the range may not always be aligned to cache line boundaries, TM systems that track state at cache line granularity would still experience difficulties.

We used a number of data structure algorithms for our evaluation (linked list, array-based heap, hash table, AVL tree, and B-tree of degree 5). Such data structures are interesting because they include significant parallelism, they are mostly pointer-based, they are regular enough for a programmer to master, and they can be used to create workloads with varying conflict frequencies or transaction sizes. While transactional versions for most of these data structures have been presented before, we contribute new versions with early release. For each data structure, we constructed the following benchmark code.

First, we prepopulate the data structure with 6,000 elements. For all structures but the heap, the key for the elements is a randomly-generated 8-character string. For the heap, the keys are randomly generated integers. Then, we perform a series of data structure accesses. To create workloads with frequent conflicts that favor early release, we use the following mix of accesses: 35% insertions of new elements, 35% deletions of existing elements (after searching for them), and 30% reads of elements (after searching for them). A transaction includes one access followed by 400 cycles of work on the retrieved data. The amount of work per transaction makes for a high rollback penalty, further benefiting early release.

We coded our benchmarks in C using a simple inlined API to define transaction boundaries and early release of addresses. We also coded the same benchmarks using both coarse-grain (CG) and fine-grain (FG) locks for comparison purposes. The coarse-grain case uses a single lock during the data structure access, but releases the lock during the additional work. The fine-grained case uses per-node locks to expose more concurrency in data structure accesses. The lock-based code ran on a version of the simulator that has the same resources as the TCC version but uses regular cache coherence and multiprocessor synchronization. For both transactional and non-transactional code, we ran simulations using 1 to 32 processors. For brevity, we present the results for 8, 16, and 32 processors. Larger processor counts generate the highest conflict frequency, but including some smaller counts provides insights into scaling. Results are presented in Figure 1 as execution time, normalized to sequential execution (represented by 100%). Lower bars are better. We break down execution time to useful (regular instructions and cache stalls), violation (time wasted on transactions that rollback), and overhead due to early release (additional instructions). We found that the overhead of early release instructions is insignificant for all benchmarks (less than 1%), hence the corresponding bar is practically invisible in all cases.

Even though our results are measured on a particular hardware TM system, we believe that the conclusions can be generalized. While other hardware or software systems may have higher rollback overheads compared to TCC, we have artificially increased the cost of rollbacks by introducing a significant amount of work per data structure access. In a software TM system, one could call early release once per object instead of once per word, hence reducing the overhead due to additional instructions [8]. However, we found that no workload exhibits significant overhead due to the early release instructions. The coding complexity arguments are equally applicable for all TM systems regardless of the exact API used. On the other hand, if early release is applied automatically by a compiler, there is no coding complexity visible to the programmer.

3 Linked List

The linked list data-structure is not optimal for parallel accesses, as it arranges elements in a single list and search takes $O(N)$ steps to access. The coarse-grain lock (CG) code for linked list accesses (not shown) simply adds a set of Lock and Unlock statements around the sequential code. Similarly, the original TM code (not shown) simply adds begin_xaction and end_xaction around the sequential access code and the additional work per access. The TM and CG code is easy to write given the sequential code.

Figures 2 and 3 show the code for linked list insertion with fine-grain lock (FG) and TM with early release respectively (TM+ER). The code for deletions is similar. The FG code holds a lock on the current node as it scans the list to ensure correct insertion. Locks are acquired and released in a careful hand-over-hand process to make sure that there is no point at which no lock is held (current and next locks overlap). The TM+ER code is similar, but uses Release instead of Unlock and there is no need for Lock statements (addresses are inserted in the read-set and write-set automatically on loads and stores). Overall, the complexity of developing FG and TM+ER code is similar. Misplacing or misusing a Release statement is

---

1 We also performed experiments with larger pre-populations (e.g., 60,000 elements). The results show similar trends, so we omit them for brevity.

2 The begin_xaction and end_xaction statements are outside of the insert function. They also enclose the code for the extra work per element.
as easy as misplacing or misusing an Unlock. Unlike with locks, forgetting an early release statement has no correctness implications. However, having an extra one, using the wrong address, or using it too early can lead to incorrect code.

As pointed out by an anonymous reviewer, “the early release code does not maintain transactional consistency if a search is performed on a linked list and the key is not found, since this leaves the entire list absent from the read set. Non-serializability can result if there are concurrent insertions. For example, thread A checks list 1 for X, and inserts Y in list 2 if not found; thread B checks list 2 for Y, and inserts X in list 1 if not found. Transactional semantics say that either X or Y should be inserted. However, early release allows both X and Y to end up in the lists. Both the early release and fine-grained lock version are thus not composable.” This observation provides further evidence on the dangers of using early release in a manual manner. It also indicates that any automated use of early release by a compiler cannot be applied as a local optimization. The global atomicity behavior of the application must be taken into account. Hence, early release may be difficult to use within library code.

Figure 1 shows the execution time of the four versions of the linked list code. The CG code allows overlapping of the additional per-access work, but no concurrency on the list. The FG code leads to 2.9x better performance in the 16-processor case. Since threads operate on different parts of the long list, they only occasionally block each other when one thread
inserts as the other tries to scan through. Nevertheless, FG is limited by the instruction overhead and cache misses on lock acquires and releases. The TM code suffers from frequent violations. Transactions that scan towards the end of the list are likely to be rolled back by insertions or deletions towards the front as all the list pointers end up in their read-set. Even so, TM is 2.5x faster than CG code in the 16-processor case, which has similar complexity, and is only marginally slower than FG code. Early release allows transactions to have the minimal possible read-set (two elements) and leads to the best performance. TM+ER eliminates violation overhead and offers a 1.5x speedup over simple TM. Looking at how the two TM versions scale, we begin to see higher violation overhead in the 32-processor case for TM. Hence, we surmise that early release would become increasingly important as we scale TM to larger processor counts for data-structures like linked lists. Note, however, that this is a best case scenario for TM+ER. If the workload is less write-intensive, conflicts are less frequent and less expensive, and the advantage of TM+ER over ER is quickly reduced.

4 Array-Based Heap

Due to space reasons, we do not present any of the code for the array-based heap. The heap exhibits the properties of the linked list taken to an extreme. Again, elements are arranged in a single sequence, but now we also have frequent element swapping throughout the heap on updates. Due to these bubble-up operations, violations are very frequent for the transactional versions with such a write-intensive workload. Just like the linked list case, the FG and TM+ER code for the heap attempts to hold a lock for two elements or include in the read-set two elements. The two algorithms have similar structure and almost identical complexity.

Figure 1 shows the execution time of the four versions of the heap code. Again, for a highly sequential structure with a write-intensive workload, TM+ER leads to significant performance benefits over TM (2.8x with 16 processors). Again, a workload with less insertions or deletions exhibits smaller differences. Nevertheless, FG is actually 50% faster than TM+ER. Early release eliminates many, but not all, violations for heap updates, which are very write-intensive. Notice that for both the linked list and the heap, the maximum speedup we achieve with the 16-processor CMP is less than 7. The two data-structures are not well-suited for parallel accesses, particularly as we scale the number of processors in the system. Excluding the FG code, no other version scales well with more processors.

5 AVL Tree

AVL trees are balanced binary trees that search in $O(\log N)$ time. Multiple threads can operate in parallel on different branches. Significant interference is only observed on rotations for rebalancing. Even in that case, we typically have to rotate only a sub-tree, so most threads are not affected. The CG and TM codes are again simple updates from the sequential code. The TM has a small read-set in most cases, as it only touches a single branch.

Figures 4 and 5 show the code for AVL tree insertion with fine-grain lock (FG) and TM with early release (TM+ER). The code for deletions is similar. The FG lock holds a lock to the immediate parent of the point of insertion. As in the case of the linked list, locks are acquired and released in an overlapped, hand-over-hand manner. If a rotation is necessary, a coarser-grain lock is acquired for the whole sub-tree. The TM+ER releases all nodes but the current parent as it scans down the branch using again an overlapped release process. Again, the FG and TM+ER version have very similar complexity.

Figure 1 shows the execution time of the four versions of the AVL tree code. Again, CG can only overlap additional work; all operations to the tree are serialized. The other three versions of the code, TM, FG, and TM+ER, perform and scale similarly. Despite the overhead of lock acquisition, FG has a performance advantage over the TM versions. Even though early release does eliminate some rollbacks, it does not provide a significant advantage over TM (less than 15% improvement is observed). Most rollbacks are due to subtree rotations that are difficult to avoid in both TM and TM+ER.

6 B-Tree and Hash Table

Due to space limitations, we do not present any of the code for the B-tree or the hash table. Much like the AVL tree, the B-tree allows for fast searches and concurrency across branches with minimum interference. The B-tree code is similar to the AVL tree in all cases, but properly adjusted to the degree and nature of the tree. The CG and TM code are trivial while the FG and TM+ER code requires careful locking or releasing as we scan through the tree. Figure 1 shows that the B-tree performance results are similar to the AVL results. In this case, rollbacks due to rotations are much less frequent, as the balancing requirement is relaxed and the tree nodes hold multiple pieces of data. Due to the small number of violations, we do not notice a visible performance advantage for early release. Interestingly, the TM code is even faster than the FG code.
that suffers from lock acquisition overheads, while the TM code does not experience significant overheads due to violations.

The hash table is the most concurrent data structure of all in the group we studied. Searches are very fast and interference is unlikely. As there are 256 bins, two threads/transactions rarely work on the same bin. Again, the CG and TM code is trivial. In this case, however, the FG code is also simple as we only lock a bin at a time, as opposed to the individual elements within the bin (as in the linked list case). Figure 1 shows that TM, TM+ER, and FG perform nearly identically. Early release does not help, as we rarely have two transactions working on the same list.

7 Conclusions

The results in Figure 1 show that even for the very write-intensive workloads we studied, early release provides insignificant performance improvement (or no improvement at all) over simple, coarse-grain transactions for the data structures that scale well in parallel systems (trees and hash tables). On the other hand, for linear data-structures like linked-lists and array-based heaps, early release can significantly reduce the overhead due to violations. We have also shown through specific code examples that the complexity of early release can be similar to that of fine-grain locks. While missing early release statements do not affect correctness, a misplaced or additional early release is as bad as a misplaced or missing lock/unlock statement and may lead to atomicity breaches. Hence, instead of applying early release on linear data structures like linked-lists and heaps, the programmer is probably better off switching to a more concurrent data structure and using simple transactions.

Overall, our analysis suggests that early release is not a particularly useful construct for user-level programming with transactional memory. While our conclusions apply only to the workloads we studied, they suggest that the added programming complexity is not worth the limited performance boost from early release. Moreover, if it is difficult to use early release and extract performance benefits with regular data structure code, it is unlikely that it will be sufficiently useful with more complicated code. Coarse-grain transactions, potentially with nesting support [10], are sufficient to achieve good performance with the simple code that programmers expect from transactional memory.

On the other hand, there are certain cases where early release can provide significant performance advantages, as shown by the linked-list and heap workloads. Hence, early release may still be a useful to an optimizing compiler that is able to automatically identify when it is safe and profitable to use in transactional code.

References

int List_Insert_FineGrain(LinkedList *list, string searchKey, int data)
{
    ListNode *insert = CreateNode(search, data);
    ListNode *prev = list->head, *cur=prev->next;

    Lock(list->head->lock);
    while(cur!=NULL){
        Lock(cur->lock);
        if(searchKey<=cur->key){
            insert->next=cur;
            prev->next=insert;
            Unlock(prev->lock);
            Unlock(cur->lock);
            return 1;
        }
        Unlock(prev->lock);
        prev=cur;
        cur=cur->next;
    }
    insert->next=NULL;
    prev->next=insert;
    Unlock(prev->lock);
    return 1;
}

Figure 2: Fine-grain (FG) locking code for linked list insert.

int List_Insert_TCC_EarlyRelease(LinkedList *list, string searchKey, int data){
    ListNode *insert = CreateNode(search, data);
    ListNode *prev=list->head, *cur=prev->next;

    while(cur!=NULL){
        if(searchKey<=cur->key){
            insert->next=cur;
            prev->next=insert;
            Release(&prev->next);
            Release(&insert->next);
            return 1;
        }
        Release(&prev->next);
        prev=cur;
        cur=cur->next;
    }
    insert->next=NULL;
    prev->next=insert;
    Release(&prev->next);
    Release(&insert->next);
    return 1;
}

Figure 3: TM with early (TM+ER) for linked list insert.
AVLTree_Insert_FG(AvlTree *tree, string key, int data){
    Lock(tree->root);
    AVLTree_Insert_FG_Helper(tree->root, key, data);
}

AVLTree_Insert_FG_Helper(Node *current, string key, int data){
    if(key==current->key){
        Unlock(current->Lock);
        return;
    } else if(key<current->key){
        if(current->left==NULL){
            current->left=CreateNode(key,data);
            Unlock(current->lock);
            return;
        } else{
            Node *left=current->left;
            Lock(left->lock);
            Unlock(current->lock);
            AVLTree_Insert_FG_Helper(left,key,data);
            Balance(left);
        }
    } else {
        if(current->right==NULL){
            current->right=CreateNode(key,data);
            Unlock(current->lock);
            return;
        } else{
            Node *right=current->right;
            Lock(right->lock);
            Unlock(current->lock);
            AVLTree_Insert_FG_Helper(right,key,data);
            Balance(right);
        }
    }
}

Figure 4: Fine-grain (FG) locking code for AVL tree insert.
// Can avoid a wrapper function by calling the inner function directly.
AVLTree_Insert_EarlyRelease(AvlTree *tree, string key, int data){
    AVLTree_Insert_ER_Helper(tree->root, key, data);
}

AVLTree_Insert_ER_Helper(Node *current, string key, int data){
    if(key==current->key){
        return;
    } else if(key<current->key){
        if(current->left==NULL){
            cur->left=CreateNode(key,data);
            Release(current->left);
            return;
        } else{
            Node *left=current->left;
            Release(current);
            AVLTree_Insert_FG_Helper(left,key,data);
            Balance(left);
        }
    } else {
        if(cur->right==NULL){
            cur->right=CreateNode(key,data);
            Release(current->lock);
            return;
        } else{
            Node *right=current->right;
            Release(current);
            AVLTree_Insert_FG_Helper(right,key,data);
            Balance(right);
        }
    }
}

Figure 5: TM with early release (TM+ER) for AVL tree insert.