Testing Implementations of Transactional Memory

Chaiyasit Manovit[†]
chaiyasit.manovit@sun.com

Sudheendra Hangal[‡]
hangal@magiclampsoftware.com

Hassan Chafi[⋆]
hchafi@stanford.edu

Austen McDonald[⋆]
austenmc@stanford.edu

Christos Kozyrakis[⋆]
kozyraki@stanford.edu

Kunle Olukotun[⋆]
kunle@stanford.edu

[†]Sun Microsystems
Sunnyvale, CA, USA

[‡]Magic Lamp Software
Bangalore, India

[⋆]Stanford University
Stanford, CA, USA

ABSTRACT

Transactional memory is an attractive design concept for scalable multiprocessors because it offers efficient lock-free synchronization and greatly simplifies parallel software. Given the subtle issues involved with concurrency and atomicity, however, it is important that transactional memory systems be carefully designed and aggressively tested to ensure their correctness. In this paper, we propose an axiomatic framework to model the formal specification of a realistic transactional memory system which may contain a mix of transactional and non-transactional operations. Using this framework and extensions to analysis algorithms originally developed for checking traditional memory consistency, we show that the widely practiced pseudo-random testing methodology can be effectively applied to transactional memory systems. Our testing methodology was successful in finding previously unknown bugs in the implementation of *TCC*, a transactional memory system. We study two flavors of the underlying analysis algorithm, one incomplete and the other complete, and show that the complete algorithm while being theoretically intractable is very efficient in practice.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Shared memory*; B.6.3 [**Logic Design**]: Design Aids—*Verification*; C.0 [**General**]: Systems specification methodology

General Terms

Algorithms, Verification

Keywords

Transactional memory, Testing, Verification, Specification

PACT'06, September 16–20, 2006, Seattle, Washington, USA.

1. INTRODUCTION

The shared memory programming model is very popular for parallel architectures primarily because it is relatively easy to use compared to the message passing model for example. Proper synchronization between processes, however, must be employed to ensure correct behavior. Such synchronization is normally used to provide mutual exclusion between different execution streams via acquisition and release of locks. Unfortunately, lock-based synchronization has a number of programmability disadvantages as well as performance problems in scaling to large systems, even in the absence of contention.

To solve these problems, Herlihy and Moss proposed an implementation called transactional memory which can be used to provide atomicity in the context of a multiprocessor [8]. In *transactional memory* (*TM*) systems, programmers can define a customized block of code called a transaction whose operations appear either as if they execute atomically or never execute. Although direct hardware support for TM has not yet become widespread in practice, it has nevertheless been an area of active research in computer architecture. Several other forms of TM have also been proposed [2, 6, 14, 15, 17, 18], spanning a large design space [15, 19]. In these systems, some combination of hardware and software is responsible for providing transaction atomicity guarantees to the programmer. Typically, basic hardware support is provided, and if available hardware resources are exhausted, the system falls back to an alternative software implementation. Historically, database designers have studied transactions in great detail and have been concerned about the issues of Atomicity, Consistency, Isolation and Durability (ACID); these issues, except perhaps durability, are concerns in TM systems as well. In particular, the term "atomicity" in TM context usually refers to both atomicity and isolation.

It is easy to see that in return for a simpler programming model, transactional memory imposes a greater burden on the system designer. Commercial shared memory multiprocessors today are already very complex machines involving, for example, hardware multi-threading, several levels of caches and multiple coherence protocols. TM implementations may require several other complexities like transaction caches, speculative writes, atomic reads and writes to hardware state, commit broadcasts, etc. Given the subtleties

involved with preserving ordering and atomicity guarantees to the programmer, while still allowing a high degree of parallelism for good performance, it is clear that aggressive verification is imperative to ensure that such a system works reliably. The pseudo-random testing methodology used extensively by commercial microprocessor and system design teams [3,10,11,13,21] cannot be extended easily to tests with unordered transactions or instructions which access shared memory locations. Such tests can produce multiple outcomes which are legal under the system specification, and it is not obvious how legal and illegal results can be distinguished from each other. Past work with conventional multiprocessors (without TM) has shown that using short test programs with aggressive data races can help expose subtle atomicity and ordering bugs during the pre-silicon or post-silicon verification phases of commercial systems [7].

Our main contribution in this paper are as follows:

1. We develop a framework of formal axioms for describing legal operation of a TM system. The system may include traditional memory consistency semantics for non-transactional operations.

2. We use this framework to implement a pseudo-random testing methodology for TM systems and efficient analysis algorithms based on this framework. Our approach extends earlier work on testing conventional multiprocessors to testing a system which allows a mix of transactions and ordinary instructions. We apply this methodology to a well-known TM system, namely *TCC* (*Transactional memory Coherence and Consistency* [5,6]).

3. We report on bugs found using our methodology in the TCC system design and on trade-offs with respect to completion and runtime of our analysis algorithms.

In Section 4, we show that the general problem of determining if the results of a TM program are legal is NP-Complete. We describe two algorithms for solving the analysis problem: the first is a polynomial-time algorithm which is sound, but incomplete, i.e., it may miss some errors, but will never report false errors. We then consider a backtracking-based algorithm which is both sound and complete. This algorithm has exponential time complexity in theory, but due to our optimizations, works in very reasonable time in practice. These algorithms can also be applied almost directly to the corresponding serializability problems in the context of database transactions.

In the rest of this paper, Section 2 introduces the framework of formal axioms for transactional memory, including memory consistency models for non-transactional operations. Section 3 provides an overview of our methodology to test an implementation of transactional memory. Section 4 analyzes the complexity of checking transactional memory and some flavors of the problem, and then describes our analysis algorithms, along with examples of the kind of errors it would catch. Section 5 provides results of using our methodology on the TCC system. Section 6 describes related work and Section 7 concludes.

2. FORMAL SPECIFICATION OF TM

We allow a transactional memory program to have both transactional and non-transactional memory operations; further, non-transactional operations are governed by traditional memory consistency rules, except that they may not intervene between operations within a transaction in the global order. This models a realistic multiprocessor system since it is likely that a system with support for transactions will still need to support existing non-transactional code for that instruction set architecture, as long as the memory locations accessed by transactional and non-transactional instructions are non-intersecting. Of course, transactional memory systems which require that all instructions be part of a transaction are a special case of our formulation. We also make the assumption that reordering between transactions on the same processor as well as reordering of instructions within a transaction are not allowed. Only committed transactions are important for the purposes of checking architectural results, since aborted transactions are assumed to have no programmer-visible effect on memory. We do not require that nested transactions are flattened. In some systems, inner transactions may abort without aborting the outer ones. However, the fact that the outermost transaction successfully commits should still indicate that all instructions including the successful inner transactions execute atomically. Therefore, a nested transaction still appears to a programmer as a single transaction.

We base our formal specification of transactional memory systems upon the style of specifying memory consistency models [20]. Though we use the Total Store Order (TSO) memory model for illustration in this paper, other models like SC (Sequential Consistency) and PSO (Partial Store Order) can be incorporated using a similar framework.

The notation used is as follows. The superscript and the subscript may be omitted when they are irrelevant or there is no ambiguity.

L_a^i	a load from location a by processor i.
S_a^i	a store to location a by processor i.
$Val[L_a^i]$	the value read by L_a^i.
$Val[S_a^i]$	the value written by S_a^i.
Op_a^i	either a load or a store.
M	a memory barrier.
;	operator for per-processor program order.
$\leq$	operator for global memory order.
[ ]	a transaction boundary (no nesting).

An order is a relation that is reflexive, anti-symmetric and transitive. Two kinds of orders are used in the definition of these axioms: ";" denotes a per-processor program order, and "$\leq$" denotes a global memory order in which operations *appear* to be performed by memory. $[Op_1; Op_2]$ denotes that Op_1 and Op_2 are inside the same transaction, but due to the transitivity of ";", they are not necessarily consecutive operations in program order, that is, there may or may not be Op_3 such that $[Op_1; Op_3; Op_2]$. Similarly, $[Op_1]$ does not imply that Op_1 is the first, the last, or the only operation in the transaction. Moreover, without explicitly specifying Op_1 inside a transaction boundary, Op_1 may refer to either a non-transactional or transactional operation.

Loads are represented in the global memory order by the time at which their return value is bound (i.e., cannot be changed) while stores are represented by the time at which the store is globally visible to all processors in the system. The following are the axioms for a TM system employing the TSO memory model for non-transactional operations:

TransOpOp: Program order within a transaction implies

memory order.
$[Op_1; Op_2] \Rightarrow Op_1 \leq Op_2$

TransMembar: Memory barriers are implicit around each transaction.
$Op_1; [Op_2] \Rightarrow Op_1 \leq Op_2$
$[Op_1]; Op_2 \Rightarrow Op_1 \leq Op_2$

TransAtomicity: No other memory operations can intervene between two consecutive operations in a transaction.
$[Op_1; Op_2] \wedge \neg[Op_1; Op; Op_2] \Rightarrow (Op \leq Op_1) \vee (Op_2 \leq Op)$

Order: Memory order is a total order over all operations.
$\forall Op_a^i, Op_b^j : (Op_a^i \leq Op_b^j) \vee (Op_b^j \leq Op_a^i)$

Termination: All stores eventually terminate.
$S_a^i \wedge (L_a^j;)^\infty \Rightarrow \exists L_a^j \in (L_a^j;)^\infty$ such that $S_a^i \leq L_a^j$

Membar: A memory barrier ensures that an operation preceding it in program order is memory ordered before an operation succeeding it in program order.
$Op_a^i; M; Op_b^i \Rightarrow Op_a^i \leq M \leq Op_b^i$

LoadOp: The program order from a load to any operation is maintained in the memory order.
$L_a^i; Op_b^i \Rightarrow L_a^i \leq Op_b^i$

StoreStore: The program order among stores is maintained in the memory order.
$S_a^i; S_b^i \Rightarrow S_a^i \leq S_b^i$

Informally, the LoadOp and StoreStore axioms together imply that the only kind of reordering allowed between operations on the same processor is for loads to overtake stores, i.e., a load which succeeds a store in program order may precede it in global memory order.

Value: The value returned by a load is the value written to it by the last store in the memory order, amongst the set of stores preceding it in memory order or program order.
$Val[L_a^i] = Val[Max_{\leq}(\{S_a^k | S_a^k \leq L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})]$

This version of the Value axiom permits optimizations that are allowed by the TSO memory model (a load can, through *bypassing*, see the result of a preceding store issued and buffered by the same processor before that store has completed in global order); however, this axiom is also correct for a system using only transactions or for a system with sequentially consistent semantics for non-transactional operations.

Traditional memory consistency model usually provides a swap operation which reads the original content of a memory location and "immediately" writes a new value to it. Here we denote a swap operation with $[L_a^i; S_a^i]$, borrowing the notation of a transaction boundary.

Atomicity: No operations can intervene between the load and store components of a swap.
$[L_a^i; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall Op_b^j : Op_b^j \leq L_a^i \vee S_a^i \leq Op_b^j)$

Viewing a swap as a two-instruction transaction, however, the Atomicity axiom is subsumed by the TransAtomicity axiom.

All of the above axioms together specify the behavior of a realistic TM system using TSO semantics for non-transactional operations. The TransOpOp, TransMembar, TransAtomicity, Order, Termination and Value axioms completely specify a transactions-only system (without explicit memory barriers and inbuilt atomic swap operations) like TCC [6], while the Order, Termination, Membar, LoadOp, StoreStore, Value, and Atomicity axioms specify a traditional multiprocessor system based on the TSO memory model.

It is important to note that these axioms describe the behavior of a TM system that *appears* to programmers, but *do not* describe or suggest how it should be implemented. This concept is similar to the fact that a uniprocessor can be described simply as being sequential regardless of how much instruction level parallelism it actually exploits. As an example in our context, consider the following two weaker versions of the Atomicity axiom.

Atomicity-A: No *stores* can intervene between the load and store parts of a swap.
$[L_a^i; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall S_b^j : S_b^j \leq L_a^i \vee S_a^i \leq S_b^j)$

Atomicity-B: No operations from *any processor to the same location* and no operations from *the same processor to any location* can intervene between the load and store parts of a swap.
$[L_a^i; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge$
$(\forall Op_a^j : Op_a^j \leq L_a^i \vee S_a^i \leq Op_a^j) \wedge$
$(\forall Op_b^i : Op_b^i \leq L_a^i \vee S_a^i \leq Op_b^i)$

It can be shown that a system implementing either of these two weaker axioms is still perceived by programmers as equivalent to that implementing the original version listed above [9] (the proof is outside the scope of this paper). Another example is that accesses to different memory locations within a transaction may be reordered by an implementation. As long as the dependence order and transaction semantics are correctly maintained, programmers can choose to believe that such reordering did not occur. Generally speaking, for verification purposes, it is more convenient and simple to consider the strictest version of the axioms. For deciding on implementation optimizations, system designers may find the most relaxed form of the axioms more useful.

Note, on the other hand, that these axioms are not a precise specification of an implementation and it is possible that the behavior allowed by an implementation is a subset of that allowed by the axioms.

3. TESTING TRANSACTIONAL MEMORY

Our methodology to test an implementation of transactional memory is based on a straightforward extension of prior work for checking the memory consistency model of a multiprocessor [7].

Step 1: We generate a pseudo-random multiprocessor test program with both transactional and non-transactional operations which access a relatively small number of shared memory addresses. Transactional and non-transactional operations are controlled to access non-intersecting set of addresses if so required by the TM system. The test case is instrumented to observe the architectural results of running the test, such as the value read by each non-transactional load instruction or each load instruction in a committed transaction. On a real system or in a hardware emulation environment, these results can be buffered in processor registers and only flushed to memory when the register buffer gets full, in order to minimize test perturbation. In some simulated systems, the simulation environment has a means to obtain these architectural results without any instrumentation overhead. To minimize overhead, the value

written by every generated store instruction is statically determinable and does not have to be explicitly stored as part of the results. Various properties of the generated program such as instruction mix, statistical distribution of transaction length, number of shared memory addresses, sequences of instruction patterns, etc. can be controlled by the user. The test generator needs to be aware of the specific types of instructions of the TM system, e.g., the mechanism to begin, commit, or abort a transaction, but is otherwise fairly portable, especially when it is targeted to generate programs in a high-level language such as C. For a transaction which aborts (either due to an explicit abort instruction in the test, or due to contention with another transaction), the test case will retry the transaction. We expect the TM system to already provide some guarantee of forward progress; therefore, a test which fails to complete before a timeout is considered an error. The test can includes all operations (including non-transactional operations) supported by the instruction set. For example, for a typical instruction set architecture, it would include different-sized loads and stores, compare and swap, prefetches, flushes, conditional branches, non-faulting loads, inter-processor interrupts, non-cacheable operations, etc.

In order to allow us to map each read value observed in the program back to the store which created it, we ensure that each store value used in the program is unique. This is an important requirement for the analysis algorithm, as explained in the next section.

Step 2: The test program from step 1 is assembled or compiled and run on a test environment like an actual multiprocessor system or a simulation model at the architectural, RTL (Register Transfer Level) or gate-level.

Step 3: The architectural results of the test program are fed into an analysis algorithm. The analysis algorithm is oblivious to the specifics of the TM system, as long as it has a description of the dynamic order of all operations (including transaction boundaries) that were committed and the values read/written by all loads and stores. No other visibility into the test execution is assumed, nor any specifics about how the TM system is implemented, for example, in hardware, software or a combination of the two. However, additional ordering information can be used if it is available. At the end of analysis, a pass or fail is signaled. Since it is possible that different runs of the same test program may obtain different results in the presence of external perturbation, the analysis result refers to the correctness of only that particular run of the test program.

To prepare for analysis, the dynamic sequence of program instructions on each processor is converted to a sequence of nodes in a graph. Transactions which aborted do not appear in this graph since they should have no programmer-visible effect. Nodes representing instructions which do not have programmer visible effect on memory such as prefetches and flushes are converted to no-ops. Compare and swap instructions are resolved into either a swap or an ordinary load. Nodes representing instructions which cover multiple shared words of interest are expanded, so that all loads, stores and swaps in the analysis graph are of a uniform size.

Finally, edges are added in this graph to represent constraints on the memory order $\leq$ according to the analysis algorithm described in the next section. Note that $\leq$ reflects the perceived order rather than the order in terms of actual time.

4. RESULTS ANALYSIS

Analysis algorithms try to answer the following question: Given a multi-threaded program with transactions and non-transactional operations, a program order ";" for each thread, and the results (load and store values) for a run of this program, does there exist a valid memory order "$\leq$" under which all axioms of the underlying memory system are satisfied? Specifically, we consider the version of the problem where the *read-mapping* is well-defined, i.e., the value read by every load can be mapped to the store which created that value. Before describing the analysis algorithm, we discuss the theoretical complexity of the analysis problem.

4.1 Analysis Complexity

We note here that the definitions of the Sequential Consistency (SC) memory model and the TSO memory model are almost identical except that program order ";" directly implies global memory order "$\leq$" in SC while $S; L$ does not imply $S \leq L$ in TSO.

Checking that an execution of a test program complies to the specification of the Sequential Consistency (SC) memory model is known to be an NP-Complete problem for an unlimited number of processors. This is termed the *VSC* (*Verifying Sequential Consistency*) problem by Gibbons and Korach [4]. The problem remains NP-Complete even if the mapping function between each load and the store which wrote the value it read is known. This is called the *VSC-read* problem. Similarly the *VTSO* and *VTSO-read* problem are also NP-Complete [7].

Following similar terminology, we call our problem the *VTM-read* problem (*Verifying Transactional Memory* with read-mapping), with TSO semantics for non-transactional operations. The fact that every store writes a unique value in our problem makes it obvious which store created the value read by each load. Therefore, the read-mapping information is readily available.

THEOREM 1. *VTM-read is NP-Complete, assuming an unlimited number of processors.*

PROOF (OUTLINE).

1. VTM-read is in NP. Given a schedule of memory operations, it can be verified against each VTM axiom in polynomial time.

2. The VSC-read problem is reducible to the VTM-read problem because every instance of the VSC-read problem can be mapped to an instance of the VTM-read problem by embedding each instruction in a separate transaction. The TransMembar axiom ensures that instructions ordered in ";" remain ordered in "$\leq$".

□

Gibbons and Korach further define the *VSC-conflict* problem as VSC-read when write-order per location is known. They show that VSC-conflict can be solved in polynomial time [4]. Similarly for the corresponding *VTM-conflict* problem, there is a simple polynomial time algorithm (the proof is outside the scope of this paper). One way to generate test programs that reduce results analysis to the VTM-conflict problem is to ensure that each store instruction is embedded in a transaction and is preceded by a load instruction to the same address in the same transaction (i.e., there are no

"blind stores".) This effectively allows us to derive a per-location conflict order and the resulting VTM-conflict problem can be solved in polynomial time. However, the restrictions imposed upon tests which can be generated are significant and therefore this version of the problem is marginally useful for purposes of verification.

4.2 Analysis Algorithms

Our analysis algorithms try to infer as many orders as possible between memory operations that must hold to satisfy program order, and to justify the observed behavior.

A directed graph is used as the data structure for the analysis. Nodes in this graph represent operations and edges represent ordering relations in the global memory order $\leq$. Since $\leq$ is transitive, any path in the graph implies the existence of the $\leq$ relation between the source and destination of the path. A violation of any axioms in Section 2 (excluding the Termination axiom) will cause a conflict in the ordering of two or more operations and manifest as a cycle in the graph.

A *global source* node at the root of the graph acts like a set of stores writing initial values to all shared addresses. It is ordered before all other nodes in the graph. *TransAtomicity Enforcement* is a key aspect of our analysis algorithm with respect to transaction atomicity: incoming edges incident to any node in a transaction must point to its first node; outgoing edges from any node in a transaction must similarly leave from its last node. This guarantees that the TransAtomicity axiom is satisfied by the relations embodied in the graph at all times.

The analysis algorithm starts by mapping every load value to the store which wrote that value. This mapping is well-defined because every store in our tests writes a unique value. A load reading a value never written to that address causes an obvious failure at the outset. After this step, the algorithm adds any edges implied by knowledge of global memory ordering obtained through additional observability available in the system, if any. For example in a hybrid hardware-software TM system, software may be able to record some global memory ordering information. Next, the analysis algorithm adds edges by applying the following rules.

Baseline Algorithm

Static Edges: In the first step, program order edges are added to the graph according to the following 6 rules, which depend only on the test program and are independent of run results. The first three rules are related to transactions. The next three capture TSO ordering requirements for non-transactional operations.

T1: $[Op_1; Op_2] \Rightarrow Op_1 \leq Op_2$ (TransOpOp axiom)
T2: $Op_1; [Op_2] \Rightarrow Op_1 \leq Op_2$ (TransMembar axiom)
T3: $[Op_2]; Op_3 \Rightarrow Op_2 \leq Op_3$ (TransMembar axiom)
R1: $L; Op \Rightarrow L \leq Op$ (LoadOp axiom)
R2: $S; S' \Rightarrow S \leq S'$ (StoreStore axiom)
R3: $S; M; L \Rightarrow S \leq L$ (Membar axiom)

For the remaining rules, let S, S', and L be accesses to the same address and $Val[L] = Val[S]$.

Observed Edges: For all loads, the edges specified by the following two rules are added based on the load results. These edges can be added once load values are known.

R4: $\neg S; L \Rightarrow S \leq L$ (Value axiom)
This follows because S must be in one of the two store sets in the Value axiom for L.

R5: $S'; L \Rightarrow S' \leq S$ (Value axiom)
This must be true because if both $S \leq S'$ and $S'; L$ are true, $Val[L]$ cannot equal $Val[S]$ by the Value axiom.

Inferred edges: In the last step, we add more edges based on two rules which follow from the Value axiom:

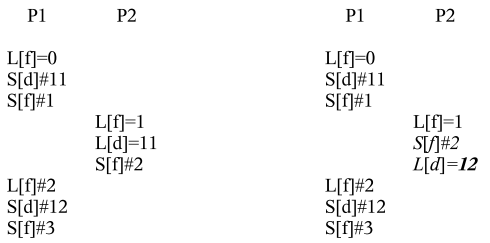
R6: $S' \leq L \Rightarrow S' \leq S$ (Value axiom)
Assuming otherwise, $S \leq S'$ (and given $S' \leq L$) will lead to a contradiction and $Val[L]$ cannot equal $Val[S]$.

R7: $S \leq S' \Rightarrow L \leq S'$ (Value axiom)
Assuming otherwise, $S' \leq L$ (and given $S \leq S'$) will lead to a contradiction and $Val[L]$ cannot equal $Val[S]$.

For rule R6, the set of all possible S' such that $S' \leq L$ can be found by traversing the graph backward from L to find its predecessors known at that time. Similarly for rule R7, traversing the graph forward from S will reach all S' such that $S \leq S'$. Note that the set of nodes traversed due to rules R6 and R7 depend on the edges already existing in the graph. If a new edge is added to the graph, applying these rules again may create further inferred edges. Therefore, we iterate over the application of rules R6 and R7 to the graph, till a fixed point is reached and no further edges are added in a complete iteration. The graph is then checked for cycles. If a cycle exists, it implies that the relations derived do not constitute a valid order because the anti-symmetry property of $\leq$ would be violated.

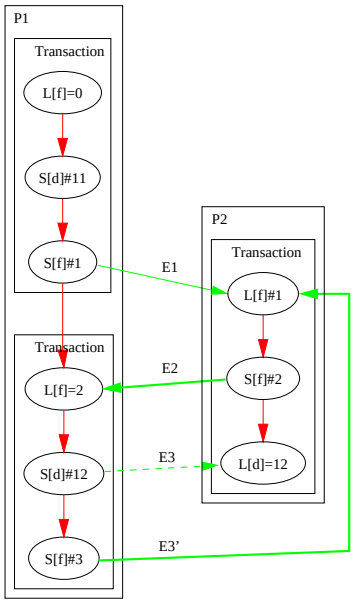
Figure 1 is an example of producer-consumer synchronization with a single producer and a single consumer. This synchronization can be achieved without locks: the producer checks the flag, produces data, and set the flag; the consumer checks the flag, consumes the data and reset the flag. However, this lock-free mechanism relies on the premise that accesses to data and flag shall not be reordered, either by hardware or software (e.g., due to a programmer mistake). With transactional memory, the ordering constraint in software can be overlooked by embedding the critical sections in transactions. This makes programming TM systems less error-prone, and hence, attractive. The notation for this and the rest of examples is as follows: S[a]#1 refers to a store which writes value 1 to location a, while L[b]=2 refers to a load from location b which returns value 2. Figure 1(a) shows the code where data (location d) and flag (location f) are accessed in the correct order. An example of possible outcomes is annotated with the code sequence. In Figure 1(b), the consumer accesses the data and flag in the opposite order. Under the TSO model, this code may produce undesirable results, yet valid, such as that exhibited in the annotation. Embedding this same code in transactions, however, precludes such undesirable results. Figure 1(c) shows why the result shown in Figure 1(b) is not valid under the TM model: edges E1, E2, and E3 are derived via rule R4, E3' is created by TransAtomicity enforcement on the dashed edge E3, and the cycle is formed by E2 and E3' (shown in bold).

Time Complexity: The above algorithm runs in polynomial time. Let the number of memory operations in the graph be n. We can consider each transaction a single node because it has a single set of incoming and outgoing edges. But overall, the number of nodes remains $O(n)$. Then the number of iterations is bounded by the number of all possible edges, $O(n^2)$ since each iteration adds at least one edge. The time complexity of each iteration is at most $O(n^3)$ since there are $O(n)$ Store-Load pairs, and we need to spend at most $O(n^2)$ time to traverse each edge in the whole graph



(a) Correct code for TSO

(b) Flawed consumer P2 – undesirable result but TSO compliant



(c) Flawed consumer P2 – undesirable result not TM compliant

Figure 1: Producer-consumer example (using increasing flag values to make store values unique).

for each pair.

Optimizations: While we have described all the inference rules used by the algorithm at a high level above, in practice, we perform several optimizations using vector clocks similar to Manovit and Hangal [12] to apply the rules efficiently and prune graph traversal when it is known that no new constraints can be inferred. We omit discussion about these optimizations since they are described elsewhere and orthogonal to the discussions in this paper.

Incompleteness: Although this polynomial-time algorithm reports no false alarms because all rules are sound, it may miss real bugs due to the fact that, in the absence of cycles in the graph, it creates a global order relation which is consistent with all the axioms *except* the Order axiom. As a result, some operations may be left unordered potentially hiding some unresolvable ordering conflicts which should have been flagged as a violation of the Order axiom. This incomplete algorithm therefore runs the risk of letting erroneous results go undetected.

Complete Algorithm

To address this incompleteness, we post-process the final graph attained by the baseline algorithm in order to discover a valid *total operation order* (TOO) which satisfies all axioms. We perform topological sort and arbitrarily assign order to operations that are left unordered by the baseline algorithm. Every time we make such an arbitrary ordering choice, we repeat inference of further constraints due to rules R6 and R7 until a new fixed point is reached. It is possible that the topological sort may get stuck due to an incorrect choice made earlier. When this happens, we backtrack to the last choice point and make a different choice; our data structures are carefully maintained such that we can undo the effect of the choice as well as further constraints which were inferred based upon that choice. It has been shown in prior work with traditional memory consistency models that performing constraint inference while searching for the global memory order is necessary for achieving good performance; without this step, the amount of backtracking required is enormous and complete analysis is impractical. Note that TransAtomicity Enforcement always applies during this post processing phase and the algorithms virtually views a whole transaction as a single node. A transaction can be selected for retirement in global memory order only if all operations within it are ready for retirement; similarly, when undoing the effect of an arbitrarily picked transaction, we undo the effect of all operations in that transaction.

5. RESULTS

In this section, we report our experiences of employing the presented methodology on a transactional memory system, *TCC* [6]. Section 5.1 briefly introduces TCC and lists specific details of our experiments. Section 5.2 illustrates samples of bugs found by our methodology. Section 5.3 reports characteristics of our analysis algorithms when applied to TCC.

5.1 Experiments on TCC

TCC is a well-known research prototype of a transactional memory system [6]. In TCC, all operations are always contained within transactions. TCC is currently available in the form of a detailed software architectural simulator, which is a C++ application designed to be linked directly with the simulated program. Its software libraries provide API interfaces to handle transactions for programs developed in C/C++. We have an ongoing effort in applying our methodology to verify the correctness of TCC implementations. Since TCC programs consist only of transactions, we never utilize the mix of transactional and non-transactional operations that our analysis is capable of handling in these experiments.

For TCC, our test generator generates a C program based on various generator controls like instruction distributions and transaction size. The C program contains memory operations in multiple threads to shared addresses, including transaction boundaries and instrumentation to observe the result of every load instruction in the program. This program is compiled with a C compiler and linked with the TCC libraries and the TCC simulator. The resulting binary is executed and it generates a results file that can be directly used by the analysis phase.

For our experiments, we had access to 2 models of TCC which we will refer to as TCC-A and TCC-B. TCC-A was the first TCC model and was fairly mature and stable. TCC-B extends TCC-A in several ways making it a more scalable and aggressive design. At the time of our experiments, TCC-

B was running many programs correctly, but was still in a relatively early phase of development.

We performed 2 kinds of experiments with TCC: the first one was aimed at finding bugs in TCC designs, and the second one characterized various features of our analysis algorithms when applied to TCC.

Bug hunting: In these experiments, we varied a number of parameters of our test generation such as the number of processors, sizes of transactions, etc. We also varied the sizes and configuration of the simulated memory hierarchy. The goal was to create extremely stressful test-cases for the system and exercise corner cases in the design. Finally, the result of each test program was analyzed using the backtracking algorithm described in the previous section. We ran tests in this mode on both TCC-A and TCC-B models.

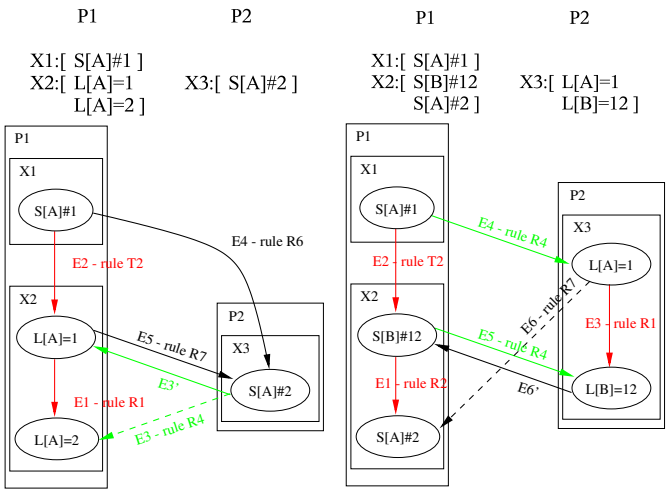
Characterization of analysis algorithms: In this set of experiments we used only the TCC-A model, fixed the sizes and configuration of the memory hierarchy to a reasonable setting, and varied only the number of operations in each test program (n), the number of processors (p), the number of shared memory location (a), and the size of each transaction in terms of the number of memory operations (s).

5.2 Bugs found in TCC

Despite the fact that we started our verification effort on the TCC-A model after it was stable and was already running test programs and benchmarks, our methodology helped uncover a previously unknown corner case bug in the software libraries. The effect of this bug was that 2 loads to the same memory location inside the same transaction returned different values in some cases, as demonstrated in Figure 2(a). This discrepancy of the load values means that the transaction was not atomic because it allowed another store to complete in between and modify the location. Detecting this as a cycle in the analysis graph requires the following steps:

- Applying rule R1 and T2 creates edges E1 and E2 which maintain the program order on P1.
- Applying rule R4 gives edge E3 that orders S[A]#2 before L[A]=2, which turns into E3' due to the *Trans-Atomicity enforcement*.
- Given S[A]#1 $\leq$ L[A]=2, edge E4 - S[A]#1 $\leq$ S[A]#2 is inferred by rule R6.
- Finally, given S[A]#1 $\leq$ S[A]#2, L[A]=1 must be ordered before S[A]#2 according to rule R7. This adds edge E5 which forms a cycle with E3'.

The root cause of this bug was that the compiler inappropriately optimized out the first load in the second transaction and, instead, directly took the value from the register that still carried the written value of the previous store, which belonged to the first transaction. This was because it could statically determine that both operations access the same address and it was not aware that the memory content could be altered. Had there been no store to that address performed by a committed transaction on another processor, it would be correct to perform the optimization. This bug was fixed by having TCC libraries correctly inform the compiler that memory content may be altered immediately after a transaction has committed.



(a) Bug in TCC-A (b) Bug in TCC-B

Figure 2: Examples of TCC bugs found by TSOtool. X1:[ ] denotes a transaction, X1.

This example illustrates that since our methodology performs end-to-end checking from the point of view of programmer visible results, it can uncover not only problems in hardware design, but also problems in software components which participate in providing transaction guarantees.

For the TCC-B model, which was newly developed and less mature, our methodology detected illegal program results (cycles in the graph) in about 9% of test-cases run. One common manifestation was similar to that previously described where 2 loads to the same address disagree on the value except that the first load may or may not see the value carried over from some previous transaction (this is a different problem from the one uncovered in TCC-A). Another bug manifestation, depicted in Figure 2(b), involved 2 memory locations and at least 2 threads. Rule R4 gives S[B]#12 $\leq$ L[B]=12, which in turn order X2 $\leq$ X3 due to TransAtomicity Enforcement. Applying rule R7 to S[A]#1 and S[A]#2, we have L[A]=1 $\leq$ S[A]#2, and hence, X3 $\leq$ X2, which conflicts with the order previously inferred. This and other failure signatures are currently being investigated by the developers of the TCC-B model.

Note that without the presence of transaction semantics, all bug manifestations shown in this section would not be considered as memory inconsistencies.

5.3 Characteristics of analysis algorithms and test programs

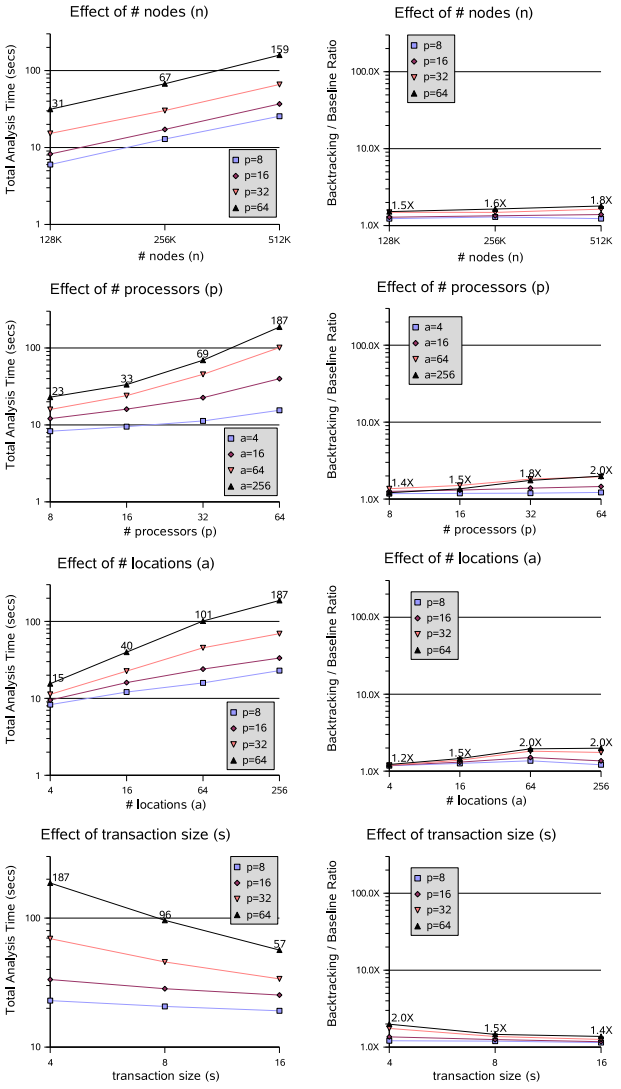
In this section, we discuss the characteristics of our analysis algorithms when applied to the TCC-A model. We studied the analysis performance while varying the number of nodes (n), the number of processors (p), the number of addresses (a), and the size of transactions in terms of the number of memory operations (s). Figure 3 summarize the effect of these variables. Figure 3(a) plots the absolute analysis time of the backtracking algorithm, which is within 200 seconds for all our test cases with p=8-64, n=128K-512K, a=4-256, and s=4-16. The performance difference between the baseline and the backtracking algorithm, shown as the slowdown ratio, is plotted in Figure 3(b). In all cases, it

takes at most twice the amount of time spent in the baseline algorithm to achieve a complete analysis using the backtracking algorithm. These graphs are plotted using the same log scale on the Y-axis to illustrate that the slowdown due to the addition of backtracking is not a major contribution to the increasing analysis time when we vary different parameters, i.e., the slopes of the graphs in Figure 3(b) are significantly lower than those in Figure 3(a).

The total analysis time grows with all the parameters except for the transaction size where the analysis time actually shrinks, with 2 probable reasons. First, the effective number of nodes is smaller with transactions as each transaction is effectively a single node. Second, the effective number of addresses is also smaller because a single transaction may access several addresses. (For simplicity, it is not too inaccurate to estimate that the effective number of addresses is a / $min(a, s)$.) This also helps explain why analyzing TCC results takes less time than analyzing TSO results for the same problem sizes in general.

We also studied at a high level how aggressively the TCC design was exercised while executing all the generated tests. For example, we measured the number of violations (which cause a transaction to be rolled back and retried) per transaction committed. In our experiments, this number was in the broad range of 0.33 to 33.84 (although the lower end is not a very small number, indicating that our tests may be biased too much toward sharing; increasing the number of shared addresses could help broaden the range). We also use the valid TOO obtained by the backtracking algorithm to approximate the degree of execution concurrency between test threads by measuring the extent of interleaving of operations from different threads relative to the total size of the program. Based on such a metric, we see a large range of achieved concurrency, almost covering both extremes (i.e., maximum concurrency and minimal concurrency). We often use such data as feedback in further tuning test parameters with the overall goal of increasing test coverage.

Although the complete algorithm has so far not found a problem missed by the incomplete one in our experiments, it increases our confidence in the results. And if it ever finds a problem, it would be of an extremely subtle nature. Therefore, we consider the complete algorithm important to run.



(a) *Deriv+Back*. (b) Slowdown ratio.

Figure 3: Analysis time of *Deriv+Back* and its slowdown ratio. The graphs for n, p, and a are plotted with s=4 and averaging out the parameter that is not shown in each respective graph. The graph for s are plotted with a=64 and averaging out n.

6. RELATED WORK

To our knowledge, this is the first paper to outline a practical methodology to test implementations of transactional memory. Gibbons and Korach established theoretical bounds on the complexity of verifying sequential consistency under various conditions [4]. Similar problems arise in the context of database transactions, where a schedule is given and the decision problem is whether it is view or conflict equivalent to some serial schedule. The fact that the VSC-read and VTM-read problems are NP-Complete and the VSC-conflict and VTM-conflict problems are in P is analogous to the fact that the View Serializability problem is NP-Complete and the Conflict Serializability problem is in P [4, 16].

Xu et al. propose a technique that essentially captures the programmer's intent and infers critical sections, i.e., transaction boundaries, in multithreaded programs from their dynamic execution paths, and use the inferred information to detect serializability violation during the execution [22]. Also, Adve and Hill study data race detection [1] which is a similar problem to detecting violations of transactions serializability. However, both these works assume that the order of synchronization events or conflicting operations is known or observed, while our work does not need to know transaction ordering. Furthermore, the consistency of the values read in critical sections is typically not checked in data race detection.

Conventional approaches for verifying atomicity and ordering are based either on test-cases limited to specific idioms whose results can be reasoned about in advance, or on extraction of global ordering information using extra observability in the system. Transactional memory renders these approaches even less effective. Extracting global order at the

hardware level tends to be complex, especially in large systems which use aggressive optimizations to maintain parallelism while preserving the illusion of transaction atomicity. Global order extraction is usually tied intimately to design details and is not easily portable across different processors and systems. Since it usually depends on intimate knowledge of internals of the system, it may miss fundamental architectural errors designed into the system. It also cannot be used on systems where such observability is not available.

7. CONCLUSIONS AND FUTURE WORK

While transactional memory simplifies parallel programming, it poses significant challenges to system designers. Since transactional memory systems invariably involve complex designs, they need aggressive and creative verification methodologies. Errors in TM system designs can result in extremely subtle and hard-to-detect bugs and will undermine the reliability of a system. We have shown that a formal axiomatic framework can capture the behavior of a system supporting both transactional and non-transactional operations, and can be used as the basis for an important pseudo-random testing methodology for transactional memory systems. Our approach has been validated on the TCC research prototype.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for helpful comments and suggestions for improving the paper. Discussions with Mark Moir, Paul Loewenstein, and Robert Cypher related to transactional memory were also enlightening.

9. REFERENCES

[1] S. V. Adve, M. D. Hill, B. P. Miller, and R. H. B. Netzer. Detecting data races on weak memory systems. In *ISCA'91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 234–243. May 1991.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, et al. Unbounded transactional memory. In *HPCA'05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[3] B. Bentley and R. Gray. Validating the Intel Pentium 4 processor. *Intel Technology Journal*, (Q1):8, Feb. 2001.

[4] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[5] L. Hammond, B. D. Carlstrom, V. Wong, et al. Programming with transactional coherence and consistency (TCC). In *ASPLOS'04: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[6] L. Hammond, V. Wong, M. Chen, et al. Transactional memory coherence and consistency. In *ISCA'04: Proceedings of the 31st Annual International Symposium on Computer Architecture*.

[7] S. Hangal, D. Vahia, C. Manovit, et al. TSOtool: A program for verifying memory systems using the memory consistency model. In *ISCA'04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 114, 2004.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[9] R. Hosabettu. SPARC V9 atomic transaction. Sun Microsystems, Feb. 2003.

[10] J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, et al. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor system. *IBM Journal of Research and Development*, 46(1):53–76, 2002.

[11] S. T. Mangelsdorf, R. P. Gratias, R. M. Blumberg, and R. Bhatia. Functional verification of the HP PA 8000 processor. *Hewlett-Packard Journal*, Aug. 1997.

[12] C. Manovit and S. Hangal. Efficient algorithms for verifying memory consistency. In *SPAA'05: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 245–252, 2005.

[13] S. Mehta, S. Ahmed, S. Al-Ashari, et al. Verification of the UltraSPARC microprocessor. In *COMPCON'95: Proceedings of the 40th IEEE Computer Society International Conference*, page 452, 1995.

[14] M. Moir. Hybrid transactional memory, Jul 2005. Unpublished manuscript.

[15] K. E. Moore, J. Bobba, M. J. Moravan, et al. LogTM: Log-based transactional memory. In *HPCA'06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

[16] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[17] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA'05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*.

[18] N. Shavit and D. Touitou. Software transactional memory. In *PODC'95: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[19] A. Shriraman, V. Marathe, S. Dwarkadas, et al. Hardware acceleration of software transactional memory. Technical Report UR CSD;TR 887, Dept. of Computer Science, University of Rochester, Dec. 2005.

[20] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Center, Dec. 1991.

[21] S. A. Taylor, M. Quinn, D. Brown, et al. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor – the DEC Alpha 21264 microprocessor. In *DAC'98: Proceedings of the 35th Design Automation Conference*, pages 638–643, 1998.

[22] M. Xu, R. Bodík, and M. D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.