Completely Verifying Memory Consistency of Test Program Executions

Chaiyasit Manovit
Sun Microsystems / Stanford University, CA, USA
chaiyasit.manovit@sun.com

Sudheendra Hangal
Magic Lamp Software, Bangalore, India[†]
hangal@magiclampsoftware.com

Abstract

An important means of validating the design of commercial-grade shared memory multiprocessors is to run a large number of pseudo-random test programs on them. However, when intentional data races are placed in a test program, there may be many correct results according to the memory consistency model supported by the system. For popular memory models like SC and TSO, the problem of verifying correctness of an execution is known to be NP-complete. As a result, analysis techniques implemented in the past have been incomplete: violations of the memory model are flagged if provable, otherwise the result is inconclusive and it is assumed optimistically that the machine's results are correct.

In this paper, we describe for the first time a practical, new algorithm which can solve this problem with certainty, thus ensuring that incorrect behavior of a large, complex multiprocessor cannot escape. We present results of our analysis algorithm on test programs run on a newly designed multiprocessor system built by Sun Microsystems. We show that our algorithm performs very well, typically analyzing a program with 512K memory operations distributed across 60 processors within a few minutes. Our algorithm runs in less than 2.6 times the time taken by an incomplete baseline algorithm which may miss errors. Our approach greatly increases the confidence in the correctness of the results generated by the multiprocessor, and allows us to potentially uncover more bugs in the design than was previously possible.

1. Introduction

This paper deals with the problem of verifying whether the outcome generated by a shared memory multiprocessor on executing a test program with data races is correct. Correctness of the result is defined by the memory consistency model [1]. Examples of common memory consistency models (also referred to as memory models in the rest of this paper) are Sequential Consistency (SC), Total Store Order (TSO), Relaxed Memory Order (RMO) [16] and Release Consistency (RC) [7]. While the focus of our discussion in this paper is the TSO memory model, our general methodology can be extended relatively easy to other memory models as well.

Our work is motivated by the challenges we have encountered in verifying large, complex multiprocessor systems. While pseudo-random testing is the backbone of microprocessor design validation efforts, it is difficult to employ this methodology effectively for multiprocessors, since a simple "golden reference model" cannot be used to check the result of a test program unless it is free of data races [13]. This is a severe limitation because tests with aggressive data races tend to expose multiprocessor bugs much faster. This is proved by the fact that verification teams often run such tests "blind", even if they are unable to check correctness of program results. They hope to expose bugs by hitting obvious problems such as a machine hang, system panic, or an unexpected program crash. However, such testing cannot hope to expose subtle multiprocessor issues such as illegal instruction ordering or atomicity violations. Even if an analysis methodology does exist to check for such violations, its efficiency is an important consideration, since test throughput is an important factor in pseudo-random testing. The more test cycles run, the higher the confidence in the design, and it is not uncommon for verification teams to be required to run several trillions of pseudo-randomly generated test instructions correctly before shipping a new processor or system [2].

Our overall multiprocessor validation flow is like this:

1. Random generator creates programs with data races
2. Each test program is run on a multiprocessor
3. Analysis algorithm verifies if program results are legal

In step 1, a multiprocessor test program is generated, which may also include instructions to store programmer visible results of its load instructions for further analysis, depending on the available observability (if the system under test is simulated, for example, we may be able to directly observe the results of load instructions). In step 2, the program is run on a test platform and the results are collected. This paper assumes the dynamic program description and its results are available, and focuses on efficient algorithms to be used in step 3. Any testing-based approach has the limitation that it is only as good as the tests employed. (Of course, we continuously tune our test-program generator to better expose corner-cases in the design). However, it has the advantage that it can be employed on a real system and not just on an abstract model. Checking end-to-end correctness on real systems is very important given the disparate elements involved –

[†] This work was carried out while the author was affiliated with Sun Microsystems India Private Limited.

e.g. multi-core, multi-threaded processors, various caches in the memory hierarchy, the coherency protocol, the system interconnect, software emulation routines in the kernel, etc. – and the complexities and corner cases in their interaction. In addition, our test methodology has the advantage that it runs on commercial multiprocessor hardware, running stock operating systems, and can even be run as a user-mode process at a customer site, since it requires no additional probes into the system, such as logic analyzers and oscilloscopes. The analysis algorithms rely only on programmer-visible results of the program and do not inherently need additional information from the system; of course, any additional ordering information is used if it is available.

Not requiring extra observability is a key attribute affecting overall test throughput, since maintaining observability often involves slowing down the system and reducing test throughput. Even in pre-silicon validation environments, hardware accelerators can simulate designs orders of magnitude faster if observability is sacrificed.

Unlike previous work [8][11], our goal in this paper is to develop a sound *and* complete algorithm (i.e. no false errors reported and no consistency errors left undetected) which is practically applicable. Instead of assuming a machine innocent unless proved guilty, we aim to determine exactly whether or not the system obeys the memory model guarantees available to the programmer, for a given test execution result.

Our paper makes the two following important contributions:

1. We describe a set of efficient algorithms for verifying TSO compliance of a test program execution. We also introduce an efficient way to perform backtracking in order to make the analysis complete.
2. We implement these algorithms and report results of applying them on large multiprocessor server systems currently under test. We show that a complete analysis incorporating edges inference with backtracking, while being theoretically exponential in the number of processors, actually runs in very reasonable time and requires minimal backtracking. Our algorithm also suggests a potential technique to solve the view serializability problem in databases, which is reducible to the VSC-read problem (see definition in Section 2.)

Paper Outline: Section 2 discusses related work and existing approaches to verifying compliance of a test execution with the memory consistency model. Section 3 presents a formal specification for TSO, and the equivalence of two variants which are necessary for correctness of the analysis algorithms. Section 4 presents a baseline algorithm, and then describes three increasingly precise extensions. Section 5 describes our results on running these analysis algorithms on large multiprocessor configurations of up to 60 processors. Section 6 concludes the paper.

2. Related work

The problem of verifying whether a multiprocessor test program execution complies with a memory consistency model was first studied by Gibbons and Korach for the SC memory model [6]. They called the problem VSC (Verifying Sequential Consistency) and proved that the basic VSC problem is NP-complete with respect to the number of operations in the program, as are several variations of the problem, when the number of processors is unbounded. In particular, the VSC-read problem, which assumes the presence of a mapping function of every read to the operation which wrote the value it read, is also NP-complete. The VSC and VSC-read problems were originally proven NP-complete by a reduction from the 3SAT and the database view serializability problem respectively [14]. The VSC-conflict problem, which is the VSC-read problem augmented with the total write order per-location, however, is in P (and similarly, so is the conflict serializability problem in databases). Similar results have been shown for the corresponding problems for the TSO memory model; VTSO and VTSO-read are NP-complete, while the VTSO-conflict problem can be solved in linear time [8][11]. The Verifying Memory Coherence (VMC) problem, which is like VSC, but involves only one memory location, is also NP-complete; however, VMC-read is in P [4]. The most interesting variants of the problem for our purpose are the VTSO and VTSO-read (or VSC and VSC-read), since pseudo-randomly generated tests can easily be mapped to these problems and the analysis is performed on architectural results visible to the program. The VSC-conflict problem, though easier to solve, is not very useful on real systems, since write ordering per location is not easily observable in general. Gibbons and Korach also propose an algorithm for VSC-read based on searching a frontier graph which has a worst case running time of $O(n^p)$ for n operations and a fixed number of processors p [5]; however, this algorithm is impractical for realistic values of p.

Our previous work describes an incomplete algorithm which makes a best effort to determine if there is a valid ordering of operations which can justify the results of the test program [8]. We also developed a simple heuristic to determine if an order satisfying all the axioms of the memory model exist [11]. However, in the many cases (up to 80% of 16 processor program runs) that the heuristic failed to determine this, we had to optimistically assume an order exists though it had not been found. This runs the risk of letting illegal results go undetected. In contrast, our new algorithm finds out exactly whether an order satisfying all the axioms of the memory model exists, and if it does, it finds the valid order as well.

Cain and Lipasti have proposed a distributed algorithm to verify correctness of program execution with respect to SC [3]; however, their techniques employs online vector clocks for each processor and at each shared memory location and assumes additional hardware logic is available for keeping these clocks updated. In our technique, in contrast, vector clocks are offline, imposing no overhead on the test program or hardware implementation. Plakal et al statically verify that a directory-based protocol implements Sequential Consistency [15], while Meixner and Sorin use their proofs to propose addition of verification hardware to the processor, cache and memory controller which can dynamically verify Sequential Consistency [12].

Microprocessor design verification teams often use the additional observability present in simulation to reason about ordering and correctness [9]. However, these techniques are heavily microarchitecture dependent, and are not usable when additional observability is absent. Taylor et al use a set of informal rules to reason about ordering of events in test execution [17]; however the completeness or efficiency of their algorithm is not described.

Finally, the generic notion of embedding memory ordering relations in a graph (and performing cycle detection on the graph to flag inconsistencies) has been used often and is originally attributed to Landin et al [10].

3. TSO specification

The axioms of the TSO memory model have been formally described by Sindhu et al [16]. We briefly discuss the notation and the axioms below. The notation used is as follows:

L_a^i	a Load to location *a* by processor *i*
S_a^i	a Store to location *a* by processor *i*
$[L_a^i ; S_a^i]$	a Swap 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
;	a per processor program order
$\leq$	the global memory order

An order is defined as a relation that is reflexive, anti-symmetric and transitive. The per processor program order is denoted by the character ; and the global memory order is denoted by the character $\leq$. The following are the TSO axioms per Sindhu et al, augmented with an additional axiom for Memory barriers.

Order: There is a total order over all stores.

$$\forall S_a^i, S_b^j : (S_a^i \leq S_b^j) \vee (S_b^j \leq S_a^i)$$

Atomicity: No stores can intervene between the load and store components of an atomic swap.

$$[L_a^i ; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall S_b^j : S_b^j \leq L_a^i \vee S_a^i \leq S_b^j)$$

Termination: If one processor does a store and another processor repeatedly does loads to the same location, there will eventually be a load that succeeds *S* in $\leq$.

$$S_a^i \wedge (L_a^j ;) \infty \Rightarrow \exists L_a^j \in (L_a^j ;) \infty \; such \; that \; S_a^i \leq L_a^j$$

LoadOp and **StoreStore**: The only reordering allowed between operations on the same processor is that loads can overtake preceding stores.

$$L_a^i ; Op_b^i \Rightarrow L_a^i \leq Op_b^i$$

$$S_a^i ; S_b^i \Rightarrow S_a^i \leq S_b^i$$

Value: The value returned by a load is the value written to it by the last store in global order, amongst the set of stores preceding it in either global order or program order.

$$Val[L_a^i] = Val[\underset{\leq}{Max}[\{S_a^k | S_a^k \leq L_a^i\} \cup \{S_a^i | S_a^i ; L_a^i\}]]$$

Membar: Membars order operations on the issuing processor.

$$Op_1 ; M ; Op_2 \Rightarrow Op_1 \leq Op_2$$

Note that the definition of the value axiom permits implementations with store buffers to locally bypass data from a store to a load, before the store is globally visible. For the SC memory model, the only difference from TSO is that this is disallowed; all relations in program order must also appear in global order.

The TSO memory model as defined in SPARC V9 [18] is slightly different from the above axioms in 2 points:

1. Memory order is total on all memory operations. (The Order axiom above only defines it to be total on all stores.)
2. Atomic swaps do not allow any other memory operation to intervene the load and the store components at all. (The Atomicity axiom above only prevents intervening stores.)

Formally:

Order: There is a total order over all operations.

$$\forall Op_a^i, Op_b^j : (Op_a^i \leq Op_b^j) \vee (Op_b^j \leq Op_a^i)$$

Atomicity: No operations can intervene between the load and store components of an atomic swap.

$$[L_a^i ; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \wedge (\forall Op_b^j : Op_b^j \leq L_a^i \vee S_a^i \leq Op_b^j)$$

It can be shown, however, that these 2 seemingly different definitions of the TSO model are essentially equivalent for verification purposes, i.e. any execution trace which satisfies the axioms of either system also satisfies the other. For ease of understanding and designing analysis algorithms, we will use the stricter versions of the Order and Atomicity axioms**.**

4. Algorithms for verifying TSO

Our main focus in this section is algorithms for the VTSO-read problem. We impose the constraint on test programs that each store in a test program writes a different value. This allows us to map each load to the store which created the value it read, and thus gives us the read-mapping function.

The following features are common to all algorithms described in this section. A program and its execution result are represented by a directed graph, whose nodes represent dynamic operations (loads or stores) in the program. 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. We ignore reflexivity of $\leq$ by not explicitly adding an edge from each node to itself. A legal outcome should not cause cycles in the graph, since this would violate the anti-symmetry property of $\leq$.

A synthetic node at the root of the graph acts like a set of stores writing initial values to all memory locations. A set of atomic operations is modeled in the graph by forcing incoming edges incident to any node in the set to point to its first node; similarly, outgoing edges from any node in the set are redirected to leave from its last node. This automatically ensures that the (stronger version of the) Atomicity axiom holds for all relations embedded in the graph at all times. A read-mapping function $w(L)$ maps each load to the store which wrote that value. A failure is directly signaled if there exists a load reading a value never written to that memory location. An inverse of the read-mapping is also computed and cached in each store node; it represents the set of all loads that read the value written by that store.

4.1. Baseline algorithm for VTSO-read

The baseline algorithm (reproduced from our previous work [11]) adds edges to the graph using following rules:

Static Edges: Program order edges are added to the graph according to the following 3 rules. These edges are independent of execution results:

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)

Note that we can redirect our verification from TSO to the SC memory model just by changing the above rules to ensure that program order between two operations also implies global order between them. The rest of the rules for all the algorithms in this section can remain exactly the same.

For the remaining rules, let S, S', and L be accesses to the same location; where $S = w(L)$ and $S' \neq S$.

Observed Edges: For all loads, the edges specified by the following 2 rules are added based on the load results.

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, L cannot read the value written by S according to the Value axiom. We only need to consider the latest store S' preceding L, because prior stores from the same thread are ordered before S'.

Inferred Edges: The last 2 rules 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 because L cannot read the value written by S as it would have already been overwritten by S'.

R7: $S \leq S' \Rightarrow L \leq S'$ (Value axiom)

Assuming otherwise, $S' \leq L$ (because there is a total order on all operations, according to the stronger version of the Order axiom), it would be illegal for L to read the value written by S as it would have already been overwritten by S'.

Note that we use the order $\leq$, which is still being derived, to determine the condition to infer more edges in rules R6 and R7. To solve this circular dependency, we iterate over these two rules until no further edges can be added to the graph and a fixed point is reached.

Intuitively, this algorithm tries to efficiently infer as much information about ordering as possible. The rules in this algorithm are selected such that they can be efficiently implemented; they are not necessarily complete. Nevertheless, if we also have available the total write order per location for the test case, the VTSO-read problem is transformed into an instance of the VTSO-conflict problem, for which this algorithm is complete [11].

A key performance enhancement for the above algorithm is the employment of vector clocks. The use of vector clocks is popular in the area of distributed computing, where they are used on each processing element to track the perceived time at other processing elements. In a similar way, we associate offline vector clocks with each operation in the program to reason about what operations on other processors must succeed this operation. These vector clocks are present only during analysis; they do not involve any maintenance by the hardware or the test program as it is running.

Using vector clocks, we avoid being exhaustive in applying rules R6 and R7 to every pair $S' \leq L$ and $S \leq S'$ in the program in each iteration. It is sufficient to start at each store node and search only for its earliest successors, either loads or stores, that access the same location but with different values. For the SC model, we only need to find the earliest such successor per thread as program order in SC implies global order. For the TSO model, a load can overtake preceding stores on the same

processor, and therefore, we split the instruction stream of one TSO processor into two *virtual SC* processors; one contains only loads and the other contains the rest (stores, atomics, and membars). In the TSO model *L;S* also implies $L \leq S$, and *S;M;L* implies $S \leq M \leq L$, and we shall represent these ordering requirements with an edge between such operations which are now separated in the two *virtual SC* processors. We attach to each node a data structure based on reverse time vector clocks to track its earliest successors in other *virtual SC* processors. This data structure helps limit the number of edges per node to the number of *virtual SC* processors. Figure 1 outlines the algorithm for rules R6 and R7.

Time Complexity: Although the total number of edges in the graph at a given point in time is bounded by $O(pn)$ where p is the number of processors and n is the number of nodes, the number of iterations in the worst case is actually bounded by the total number of possible edges, which is $O(n^2)$. This is because an edge inferred in one iteration may be rendered redundant and removed due to a stronger edge inferred in a later iteration. In each iteration, there are $O(n)$ stores whose vector clocks will be traced with $O(p)$ time complexity each. This totals to $O(pn^3)$.

Input: A per virtual SC processor instruction sequence consisting of loads, stores, and membars. A swap is considered to be both a load and a store. A function *w*, which maps a load to the store which created its value:

Data Structure: An offline Reverse Time Vector Clock at each node *x*, *x.rtvc[i]* points to the first node in virtual SC processor *i* such that $x \leq x.rtvc[i]$. Initial *rtvc[]* for all nodes are precomputed with backward topological sort.

Apply **rules R1-R5**

[rule R6 and R7] - done in iterations

```
do
 for each store S whose rtvc[] has been changed
  for each virtual SC processor i
   x := S.rtvc[i]
   if x is a load (virtual SC processor i contains only loads) then
    L := first load that accesses same location as S, x;L, and w(L)≠S
    [rule R6]
    add edge S→w(L) if not already S≤w(L)
    update S.rtvc[](and propagate to its predecessors recursively)
   else (virtual SC processor i contains stores, atomics and membars)
    S' := first store/atomic that accesses same location as S and x;S'
    [rule R7]
    for all loads L such that w(L)=S
     add edge L→S' if not already L≤S'
     update L.rtvc[](and propagate to its predecessors recursively)
    end for
   end if
  end for
 end for
until no more edges can be added
```

Figure 1. High level description of the iteration over R6 & R7 with Vector Clocks

4.2. Completely verifying TSO

We now turn our attention to algorithms which can completely verify TSO. The baseline algorithm presented in the previous section is incomplete because even when the graph is acyclic, it does not explicitly ensure that the Order axiom is satisfied. Figure 2a illustrates a case where an existing relation is not inferred by the algorithm; the edges in the graph are depicted at the point when the fixed point has been reached (edges from a store to corresponding loads reading its value are omitted to not overcrowd the graph). The notation here is: S[A]#1 refers to a store which writes value 1 to location A, while L[B]=11 refers to a load to address B which reads value 11; P*n* denotes operations on processor *n*. Notice that S[A]#1 and S[A]#2 are left unordered by the baseline analysis. However, we can reason that S[A]#1 ≤ S[A]#2 must be true. If not, S[A]#2 ≤ S[A]#1 by the Order axiom; but with this order and the fact that only one of the two values, either 11 or 12, can survive in location B after S[A]#2, both loads from location B must read the same value. This example illustrates a missing relation, but not yet a missed TSO violation; simply adding a similar, mirrored set of nodes to a different location C

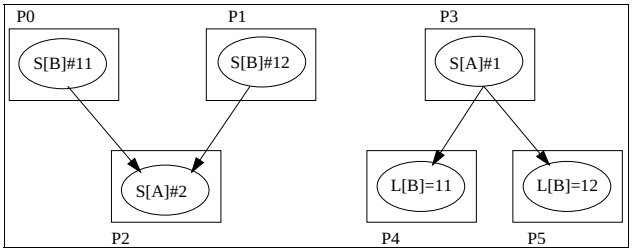

Figure 2a.

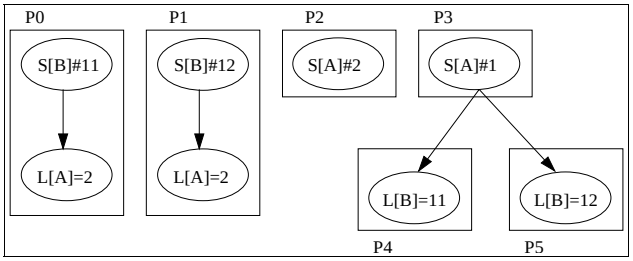

Figure 2b. (Membars are omitted)

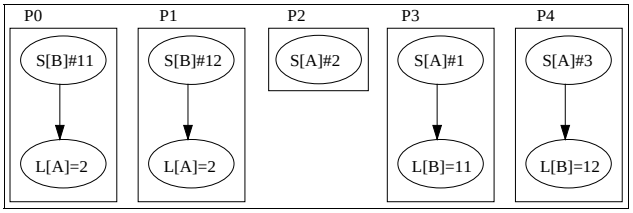

Figure 2c. (Membars are omitted)

Figure 2. Examples of incompleteness

(two stores to C ordered before S[A]#1, and two loads to C ordered after S[A]#2) creates an instance of a real TSO violation. In this case, the two stores S[A]#1 and S[A]#2 cannot be ordered, but such a violation would be missed by the incomplete algorithm in the previous section.

One may attempt to design a rule to infer the missing edge in this example. Consider the following hypothetical rule:

R8: *CommonPred*(*L*, *L'*) ≤ *CommonSucc*(*S*, *S'*)

L and *L'* are loads to the same location reading different values written by *S* and *S'* respectively. *CommonPred*(*L*, *L'*) is the latest node that precedes both *L* and *L'* in the current snapshot of the global order being derived, while *CommonSucc*(*S*, *S'*) is conversely defined. While this rule will catch the missing edge in the example shown in Figure 2a, it still misses the edge in a slightly modified scenario shown in Figure 2b because there is no common successor between S[B]#11 and S[B]#12. Note that Membars between the store and the load in the same processor are omitted from the picture (or readers may assume the SC model). S[A]#1 ≤ S[A]#2 is a missing edge because, assumed the opposite order, both L[A]=2 nodes will be ordered before S[A]#1 by rule R7, making S[A]#1 the common successor of S[B]#11 and S[B]#12 and, hence, only one value, either 11 or 12, can survive in location B.

Figure 2c illustrates that missing edges are not the only form of incompleteness. One can reason that S[A]#2 cannot be ordered before both S[A]#1 *and* S[A]#3 because that would lead to the same contradiction seen earlier with Figure 2b (when we incorrectly order S[A]#2 ≤ S[A]#1). However, such a constraint cannot be captured in our graph representation where we only draw an edge to order 2 operations when such an order is certain. Despite knowing that S[A]#1 ≤ S[A]#2 *or* S[A]#3 ≤ S[A]#2 (or both) in this example, we can draw neither edge because their presence is not certain when considered individually. To create a TSO violation that would be missed by the baseline algorithm, we can add a similar, mirrored set of nodes such that none of the stores to location A can be ordered first.

To completely verify TSO compliance, we will attempt to determine if there exists a *Total Operation Order* (TOO), which completely orders all operations (loads and stores) in the program, that also satisfies the rest of the TSO axioms. Recall that this TOO corresponds to the stronger version of the Order axiom (which is equivalent to the requirement that only stores be ordered).

A simplistic approach to determining if a valid TOO exists would be to perform a topological sort on the analysis graph after the completion of the baseline algorithm, and check if all the axioms still hold (the same baseline algorithm can be conveniently used to determine the validity of a TOO, as earlier pointed out in Section 4.1). The topological sort effectively creates an arbitrary "tie-break" decision between operations left unordered by the baseline algorithm. We have found that most often, this sort does not yield a valid order. This is because when we arbitrarily assign an order between a pair of previously unordered operations during topological sort, it often has ordering implications on other unordered operations; this creates conflicts and usually ends up violating the Value axiom. Since a straightforward algorithm based on topological sort does not work, we discuss three techniques in the following sections towards improving the chances of finding a valid TOO. In all cases, we assume the baseline algorithm has inferred all its edges and terminated without cycles in the graph.

4.2.1. Heuristic for topological sort (*Heu*). In our previous work, we ensure that each time a store node is picked by the topological sort, rule R7 is immediately applied to it [11]. An alternative, but equivalent, implementation of this heuristic is to track the *active* store (the store that was most recently picked by the topological sort) for each memory location and allow the topological sort to further pick only a load that reads the value written by the *active* store or by the store preceding the load in program order. When all loads that read the value written by an *active* store have been picked, the store becomes *inactive* and new store can be picked and made *active*. (For SC, this heuristic is similar to the conditions used to determine the validity of frontiers in the $O(n^p)$ backtracking algorithm by Gibbons and Korach [5]. However, an important difference is that their algorithm does not have any notion of initially inferring edges as in our baseline algorithm, and as a result will visit many more invalid paths in the frontier graph).

Time Complexity: A typical topological sort has O(*n*+*e*) complexity where *n* is the number of nodes and *e* is the number of edges, which is O(*pn*) in this case (because each node only maintains a vector clock). In addition, this heuristic spends O(*p*) time to evaluate the selection for the next node. This extra effort is O(*pn*) and, nevertheless, the total complexity is still O(*pn*). Note that this time complexity is for a case when the algorithm succeeds in finding a valid TOO. The heuristic may terminate much sooner when a TOO cannot be found.

Although this heuristic is intuitive and fast, we find that it is inadequate; it helps find a valid TOO only when there is relatively low sharing, i.e. *p/a* is small (where *p* is the number of processors and *a* is the number of memory locations) [11]. Section 5 provides more results.

4.2.2. Deriving edges during topological sort (*Deriv*). We can extend the heuristic technique in the previous section thus: Each time a store node is picked by the topological sort, rules R6 and R7 are reapplied iteratively to the whole graph until a new fixed point is reached. Careful implementations can minimize the computation by applying the rules only to the affected nodes. (In our implementations, such optimizations are also applied to the baseline algorithm during iteration.)

Time Complexity: Although this heuristic has to go through as many fixed points as the total number of stores which is O(n), the total number of iterations required to apply rules R6 and R7 throughout these O(n) fixed points is still bounded by the total number of possible edges, O(n^2). Therefore, the worst-case time complexity remains O(pn^3). Again, this time complexity is for the case when the algorithm succeeds in finding a valid TOO. It may terminate much sooner when this heuristic fails.

Despite the additional effort spent in deriving more edges, this algorithm's effectiveness in finding a valid TOO is still limited with intense sharing. Nevertheless, in practice, it provides significant improvement in TOO completion rate over the previous heuristic.

4.2.3. Backtracking (*Heu+Back, Deriv+Back*). Since the above heuristics are only best-effort and had unsatisfactory rates of completion (in which case the analysis is inconclusive and optimistically assumed passing), we decided to implement backtracking on top of both the heuristics described above. When the topological sort gets stuck (no instruction can be picked without violating any TSO axioms), instead of giving up, we backtrack to the last arbitrary tie-break decision made and choose a different operation to order first. Given that a valid Total Store Order will also result in a valid TOO (as pointed out in Section 2 regarding the equivalence of the two different versions of the Order axiom), we can unwind the order directly to the most recent store.

For the heuristic *Heu* in Section 4.2.1, adding backtracking is relatively easy. Adding the feature to the *Deriv* algorithm in Section 4.2.2 is less straightforward because it modifies the graph by deriving additional edges based on ordering decisions made by the topological sort. We maintain our data structures such that we can checkpoint and undo these updates when we need to backtrack and cancel the decision. Edges that are derived after a store is picked by the topological sort will be associated with the store. When we backtrack and undo the picking of a store, we remove all the derived edges associated with it and recompute vector clocks for all the affected nodes.

Time Complexity: By using a similar argument to that which Gibbons and Korach use to explain the bounds on their backtracking algorithm based on searching the frontier graph [5], the worst-case complexity of our backtracking algorithms is also O(n^p/p^p). At each step during backtracking, the additional cost of finding a new fixed point is O(pn^3). This results in O($n^p/p^p \times pn^3$) in total. However, in practice, the number and depth of backtracks is small, resulting in small penalty in terms of time over the *Deriv* heuristic in the previous section. In return for this increase in analysis time, we achieve a 100% completion rate when a valid ordering exists which justifies the results of the program.

5. Results

In this section, we present results of our extensions to the baseline analysis algorithm on test results generated from a new multiprocessor system which is actively under test at Sun Microsystems. Our results show that *Deriv+Back* performs very well; it completely analyzes programs with 512K memory operations distributed across 60 processors and finds a valid TOO for each program within 5 minutes. On average, the analysis time is less than 2.6 times that of the incomplete baseline algorithm which may miss errors. Therefore we find that *Deriv+Back* greatly increases our confidence in the correctness of the results generated by the multiprocessor, and allows us to potentially uncover more bugs in the design than was previously possible.

On the other hand, *Heu+Back* (which does not iteratively derive additional edges during the topological sort) does not perform well at all; on all tests except the ones with a small number of processors, it did not finish in a reasonable amount of time. Therefore, we ignore it from further consideration. Also recall that all of the algorithms *Heu*, *Deriv*, and *Deriv+Back* are applied on top of the baseline algorithm, that is after it has reached a fixed point. Applying them directly, without first running the baseline algorithm, we found they were much less effective: the effectiveness of *Heu* and *Deriv* in finding a valid TOO reduced dramatically and the time spent in *Deriv+Back* exploded as the number of backtracks increased substantially. While we studied these variations for completeness, we do not consider them interesting and therefore omit their detailed results from this section.

System under test: We performed the following experiments on an actual multiprocessor system designed and built by Sun Microsystems. The system we ran the test programs on has 60 processor cores. Test threads are bound to different processor cores, and run mostly concurrently since the system is quiet except for background operating system activity. We ran pseudo-random multi-threaded programs with the following instruction mix: 33.3% loads, 33.3% stores, 30% atomic swaps, 1.7% membars, and 1.7% others. We varied the number of threads/processors (p) and the number of memory locations (a) used by the programs, as well as the size of the programs (denoted as n, the total number of memory operations across all processors). The execution results of these programs were saved and later analyzed on a different system based on a previous generation 1.2 GHz Sun's UltraSPARC-III+ processor. For each tuple (n,p,a), 16 different pseudo-random programs were generated, executed, and analyzed. Unless noted otherwise, the presented results are the average over these 16 runs for each tuple. Analysis time is the major factor determining test throughput since the test threads are pre-generated and pre-compiled, and only thread selection is done at runtime. Running the test itself takes on the order of a few milliseconds on a real system.

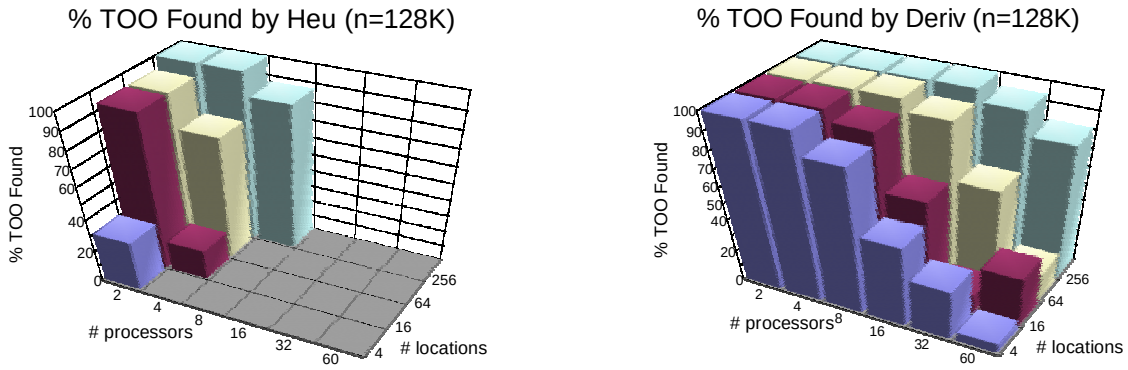

Figure 3. Effectiveness of *Heu* and *Deriv* in finding valid TOO's

While we have also applied our verification methodology to the same system in a pre-silicon software simulation environment, analysis time was not a major concern in that case. Nevertheless having a complete algorithm is useful. In simulation, though we can sometimes deal with the much simpler VTSO-conflict problem - which is in P - if total write order for each location can be observed, in reality, such ordering is often not readily available in the simulation test bench, as a single point of ordering may not exist in complex systems. Besides, software simulators usually scale up to only a few processors and cannot handle large whole-system simulations.

Figure 3 shows the effectiveness of *Heu* and *Deriv* in finding a valid TOO for n=128K. (For larger number of operations, n, their effectiveness decreases as expected.) *Deriv* provides significant improvement over *Heu* but it is still incomplete when data sharing is intense. With backtracking, *Deriv+Back* always finds a valid TOO in our experiments. A key finding is that when backtracking is necessary, the number of backtracks is at most 75, which is small for the large problem sizes used in our experiments, and the algorithm never backtracks more than 1 level each time. This means that the additional overhead due to backtracking is minimal, compared to just running *Deriv*. We also note that the analysis time overhead incurred by *Heu* is virtually constant and minimal, about 10%, while the overhead incurred by *Deriv+Back* is significant and grows with all of p, n, and a. Analyzing the largest test programs in our experiment, with n=512K, p=60, and a=256, takes, on the average, 118% more time than the baseline algorithm for cases that require backtracking (while *Deriv* would take 108% more time for cases not requiring backtracking, just a slightly smaller overhead). With a lower processor count (16 and below), the analysis time overhead is usually less than 80% over the baseline algorithm.

We deem the extra overhead in terms of analysis time worth the extra assurance that the program results are indeed correct, especially for large processor configurations where the errors may be subtle and test methodologies are limited.

Figure 4 shows the effect of n, p, and a on the analysis time. The absolute analysis time of the baseline algorithm and *Deriv+Back* are plotted in Figure 4a and 4c respectively. Figure 4b shows the ratio of the analysis time of *Deriv+Back* over the baseline. Since the graphs are plotted using log scale over the same range on Y-axis, we can view Figure 4c as being the superposition of Figure 4a and 4b. As can be seen, the slope in Figure 4b is less than that in Figure 4a, which means the increasing analysis time seen in Figure 4c are dominated by the increasing analysis time in Figure 4a. This interpretation suggests that our backtracking technique can scale (as long as the baseline algorithm scales).

We also repeated the same experiments using 2 other instruction distributions in the pseudo-random test generator: one biased toward load instructions, with 50% loads and 16% stores, and the other biased toward store instruction, with 50% stores and 16% loads (percentages of other instructions were kept the same). On the average, as the percentage of stores increases, we find that the analysis takes more time. Both the absolute analysis time of the baseline and the slowdown ratio of *Deriv+Back* are affected, as shown in Table 1.

We conjecture that a higher store density requires longer analysis time for *Deriv+Back* because there are potentially more values that are not observed at all, and hence, the baseline algorithm can infer fewer relations which would be helpful for *Deriv+Back* during backtracking. With no loads at all, on the other hand, the analysis would run very quickly because any ordering would be acceptable under TSO axioms. Therefore, we

Table 1. Baseline analysis time and slowdown ratio of *Deriv+Back* for n=256K, averaged over p and a.

	LD-biased	LD-ST equal	ST-biased
Baseline (secs)	14.9	16.5	17.5
Slowdown ratio	1.45	1.73	2.05

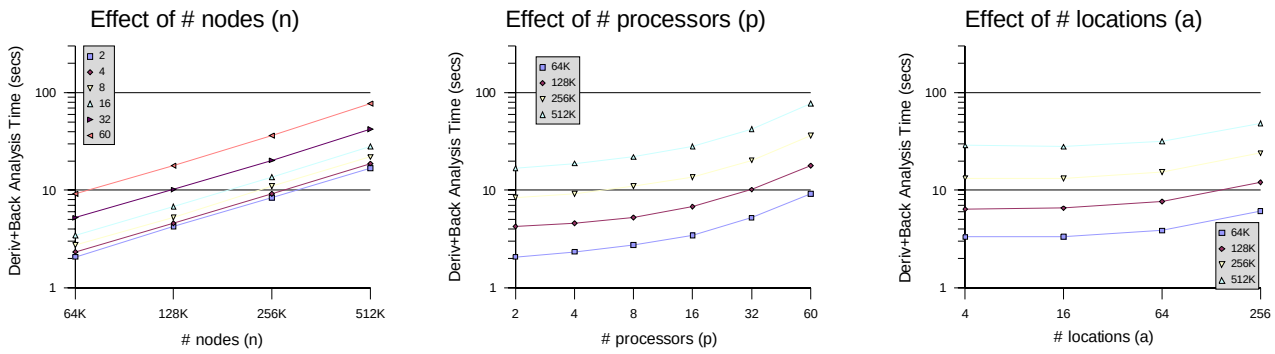

Figure 4a. Analysis time of *Baseline*

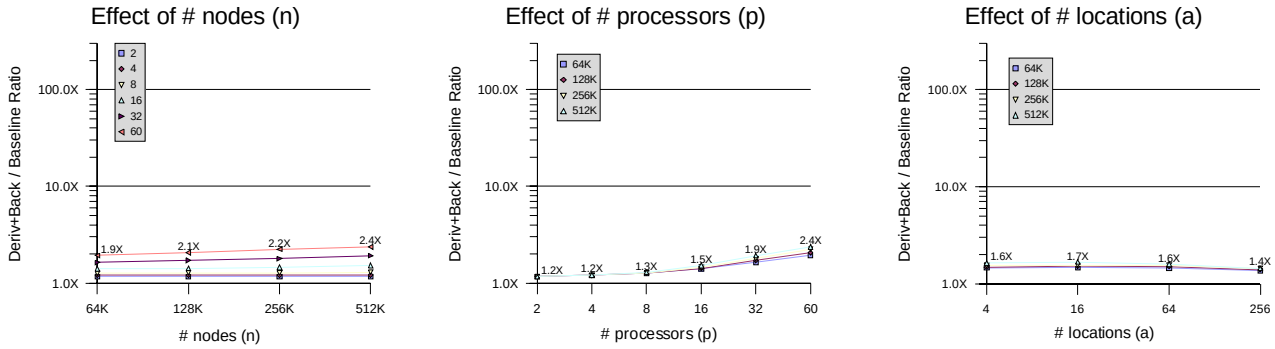

Figure 4b. Ratio of analysis time of *Deriv+Back* over *Baseline*

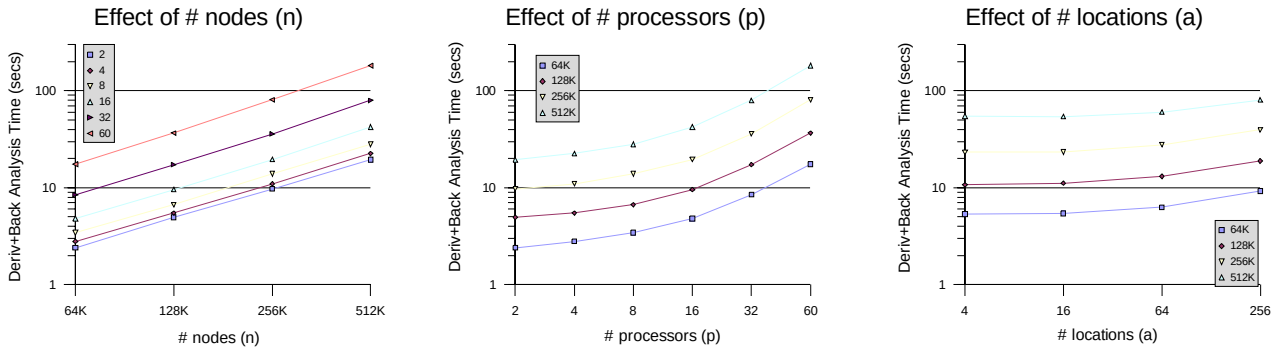

Figure 4c. Analysis time of *Deriv+Back*

Figure 4. Analysis time of *Deriv+Back vs. n* (averaged over *a*), *p* (averaged over *a*), and *a* (averaged over *p*)

expect a tipping point, as we bias the test more towards stores, where the runtime starts to decrease.

Although *Deriv+Back* has not discovered any bugs so far in the real system that are missed by the baseline analysis, we tested it with TSO violations based on the examples in Figure 2, and it successfully found the missed cycles, as expected. Being a backtracking algorithm, however, it cannot avoid the exponential analysis time complexity for such cases. We expect to explore other heuristics in order to find a smaller portion of an execution trace that contains the TSO violation.

6. Conclusions and future work

We have described a set of algorithms which can be used to verify whether a test program execution complies with the axioms of the memory consistency model. Our algorithms encompass a range of accuracy and runtime. Faster algorithms may miss errors in return for higher throughput; slower algorithms based on backtracking will not miss errors, but have an additional runtime overhead of 20-160%.

Overall, our findings indicate that backtracking is essential for a good completion rate when no violation exists; however the actual number of backtracks needed even in a large program is relatively small, and the backtracking depth is shallow. Therefore, it is well worth the trade-off to implement backtracking, since it implies only a small overhead compared to algorithms whose completion rate is much lower.

Although we present our algorithms and results based on the TSO and SC memory models, the framework that we have developed is applicable to other memory models including Relaxed Memory Order (RMO) and Transactional Memory.

7. Acknowledgments

We thank the anonymous reviewers for their useful comments, Robert Cypher for many helpful discussions regarding the VTSO problems, and Shrenik Mehta and Durgam Vahia for managerial support.

8. References

[1] S.V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", *Digital Western Research Laboratory Technical Report*, 1995.

[2] B. Bentley and R. Gray, "Validating The Intel Pentium-4 Processor", *Intel Technology Journal*, 1st Quarter 2001.

[3] H.W. Cain and M.H. Lipasti, "Verifying Sequential Consistency Using Vector Clocks", *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[4] J.E. Cantin, M.H. Lipasti, and J.E. Smith, "The complexity of Verifying Memory Coherence", *Proceedings of the 15th annual ACM symposium on Parallel Algorithms and Architectures*, pp. 254-255, ACM, 2003.

[5] P.B. Gibbons and E. Korach, "On Testing Cache-Coherent Shared Memories", *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1994.

[6] P.B. Gibbons and E. Korach, "Testing Shared Memories", *Siam Journal on Computing*, pp. 1208-1244, August 1997.

[7] K. Gharachorloo, D. Lenoski, J. Laudon et al, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors", *Proceedings of the 17th International Symposium on Computer Architecture (ISCA)*, 1990.

[8] S. Hangal, D. Vahia, C. Manovit, J. Lu, and S. Narayanan, "TSOtool: A Program to Verify Multiprocessor Memory Systems Using the Memory Consistency Model", *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2004.

[9] J. Ludden, W. Roesner, G.M. Heiling et al, "Functional Verification of the POWER4 Microprocessor and POWER4 Multiprocessor Systems", *IBM Journal of Research and Development*, Vol. 46, No. 1, 2002.

[10] A. Landin, E. Hagersten, and S. Haridi, "Race-free Interconnection Networks and Multiprocessor Consistency", *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, 1991.

[11] C. Manovit and S. Hangal, "Efficient Algorithms for Verifying Memory Consistency", *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[12] A. Meixner and D.J. Sorin, "Dynamic Verification of Sequential Consistency", *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.

[13] D. Marr, S. Thakkar, and R. Zucker, "Multiprocessor Validation of the Pentium Pro Microprocessor", *Proceedings of COMPCON*, 1996.

[14] C. Papadimitriou, "The Serializability of Concurrent Database Updates", *Journal of the ACM*, 26 (1979), pp. 631-653.

[15] M. Plakal, D.J. Sorin, A.E. Condon, and M. Hill, "Lamport Clocks: Verifying a Directory Cache-Coherence Protocol", *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998.

[16] P.S. Sindhu, J.M. Frailong, and M. Cekleov, "Formal Specification of Memory Models", *Xerox PARC Technical Report*, December 1991.

[17] S. Taylor, C. Ramey, C. Barner, and D. Asher, "A Simulation-Based Method for the Verification of Shared Memory in Multiprocessor Systems", *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2001.

[18] D.L. Weaver, T. Germond, Editors, *The SPARC Architecture Version 9*, Prentice Hall, 1994.