Efficient Algorithms for Verifying Memory Consistency

Chaiyasit Manovit
Sun Microsystems
430 N. Mary Ave
Sunnyvale, CA 94085 USA
Chaiyasit.Manovit@Sun.COM

Sudheendra Hangal
Sun Microsystems India Private Limited
Divyashree Chambers, Shantinagar
Bangalore, 560 025 India
Sudheendra.Hangal@Sun.COM

ABSTRACT
One approach in verifying the correctness of a multiprocessor system is to show that its execution results comply with the memory consistency model it is meant to implement. It has been shown in prior work, however, that accurately verifying such compliance even of a single execution result is an NP-complete problem, for an unlimited number of processors. In this paper, we present a suite of post-mortem algorithms that perform the compliance check in an efficient, although not exhaustive, manner. Our algorithms employ the concept of vector clocks together with a heuristic made from a variation of the problem in P class. An implementation of these algorithms has been successful in efficiently detecting several bugs during the course of validating the design of commercial microprocessors and systems.

Although our algorithms are presented with the Total Store Order (TSO) memory model, the ideas can also be applied to other models ranging from Sequential Consistency (SC) to a more relaxed one such as Relaxed Memory Order (RMO).

Categories and Subject Descriptors
B.3.2 [**Memory Structures**]: Design Styles – *shared memory*

General Terms
Algorithms, Theory, Verification

Keywords
Memory consistency models, Multiprocessor verification, Sequential Consistency, Total Store Order, Vector Clocks

1. INTRODUCTION
The memory consistency model is one of the major attributes of shared memory multiprocessor systems. It establishes a contract between the hardware and software regarding the behavior of accesses to shared memory. Its implication is not only limited to functionality; it also impacts performance and programmability of the system as well as portability on the software side and compatibility on the hardware side. Adve and Gharachorloo discuss various aspects and issues related to memory consistency models and provide a tutorial introduction to several models [1]. The most simple and intuitive model is Sequential Consistency (SC) [14] where all memory operations are assumed to be serialized in a way consistent with the program order in each thread. There is much published work that formally describes SC and other memory consistency models, using various frameworks designed in accordance with the different perspectives and purposes of the respective authors [6,7,11,12,15,19,20,22]. Our work is based upon the framework of axioms presented by Sindhu et al [19].

The work of verifying that a system complies with a given memory consistency model can be performed at almost all levels of abstraction and with different sets of components in the system considered; e.g. verification can be performed at protocol level, at architecture level, or at implementation level; with a model of the processor or memory system alone, or with a complete multiprocessor system; by using abstract models or real hardware. Having a system verified to be correct at a high level, however, does not guarantee that its implementation will actually work correctly as bugs may be later introduced at the lower level, or the abstraction at the high level may not contain sufficient detail to expose all corner cases. Our approach is therefore to perform end-to-end checks on a complete system, which may consist of real multiprocessor systems running commercial operating systems. We run test programs on this system, and verify the test execution results. The disadvantage with this approach is that the problem space is virtually infinite, and there is usually limited controllability and observability over the events in the system. Nevertheless, dynamically testing a complete system can potentially uncover bugs at all levels. In our methodology, pseudo-randomly generated programs with data races are run on the system, their execution results are observed and then checked for validity under the defined memory model of the system. By not necessarily assuming any observability more than what is visible to a programmer, we are able to use our methodology in both pre-silicon and post-silicon validation.

1.1 Related Work
The problem of verifying that the execution of a multithreaded program is sequentially consistent was first formally defined and studied by Gibbons and Korach [8]. The problem is denoted as VSC (Verifying Sequential Consistency); it has a number of variations, two of which are VSC-read and VSC-conflict. VSC-read is the VSC problem with additional information mapping each read operation to the corresponding write operation which created the read value. VSC-conflict is the VSC-read problem enhanced with information about the total order of write operations to each individual memory location (but not the order of writes between different locations). It has been shown that the VSC and VSC-read problems for an unlimited number of processors are NP-complete, while VSC-conflict is in P. For the VSC-read problem with a fixed number of processors k, an algorithm based on searching a frontier graph has time complexity $O(n^k)$. Cantin et al established similar complexity

SPAA'05, July 18–20, 2005, Las Vegas, Nevada, USA.

ACM 1-58113-986-1/05/0007.

results related to the problem of Verifying Memory Coherence (VMC), where only one memory location is involved [5]. Cain and Lipasti present a distributed algorithm that dynamically checks the VSC-conflict problem using vector clocks ([2,13]) with the assumption of a hardware observer [3]. The algorithm consists of an on-line checker that runs concurrently with the program it is checking. Vector clocks are also employed in other works which verify memory consistency at the architectural level [6, 21].

1.2 Contributions

Extending the study of the VSC problems to the Total Store Order (TSO) memory model or to more relaxed memory models is not trivial and, to our knowledge, there has been no previous work towards such an extension. Following the same terminology used by Gibbons and Korach [8], we call these extensions the VTSO, VTSO-read, and VTSO-conflict problems. Our major contributions in this paper are:

1. We show that VTSO-conflict is in P by describing a polynomial time algorithm, along with a proof of correctness.
2. We describe an incomplete polynomial time algorithm for VTSO-read that is fast and useful in practice; it may miss some violations, but it will never report a false violation. We extend our previous work [10] with the concept of vector clocks to lower the bound on time complexity and greatly improve running time at the expense of increased memory usage.
3. We develop the algorithm for VTSO-conflict into a heuristic for the VTSO-read problem to sometimes obtain a complete total order and increase confidence in a "yes" answer; this heuristic also achieves better run time in practice. We present results about the success rate of this heuristic.

Our approach of attacking VTSO-read with these polynomial time algorithms has proven extremely effective and useful in validating the design of commercial multiprocessors. Although our algorithms are presented with the TSO memory model, the ideas can also be applied to other models ranging from SC to a more relaxed one such as Relaxed Memory Order (RMO).

The rest of this paper is organized as follows. Section 2 describes the TSO memory model in terms of its formal axioms. Section 3 defines the VTSO, VTSO-read, and VTSO-conflict problems. Section 4 presents algorithms for solving these problems. Section 5 briefly discusses the results, and Section 6 concludes the paper.

2. THE TSO MEMORY MODEL

We briefly reproduce the 6 axioms of the TSO memory model formally described by Sindhu et al [19] 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
$;$	operator for per processor program order
$\leq$	operator for global memory order

Two kinds of orders are used in the definition of these axioms (an order is defined as a relation that is reflexive, anti-symmetric and transitive): a per processor program order denoted by the character ; and a global memory order denoted by the character ≤. In the following axioms, loads are represented in the global order by the time at which their return value is effectively bound (i.e. cannot be changed) while stores are represented by the time at which the store is effectively visible to all processors in the system. The following are the 6 TSO axioms:

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: Atomicity requires that there be no intervening stores between the load and store components of an atomic operation.

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

Termination: All stores and swaps eventually terminate. This is formally specified by requiring that 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 ≤.

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

LoadOp: If an operation follows a load in ; then it must also follow the load in ≤.

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

StoreStore: If 2 stores appear in a particular order in ; then they must also appear in the same order in ≤.

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

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.

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

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 order. When such reordering happens and both operations access the same memory location, however, the Value axiom guarantees that the load must return the value from the overtaken store, and not any other value that it could possibly overwrite. This makes the result always appears consistent to the same processor while permits optimizations such as store buffers to locally bypass data from a store to a load, before the store is globally visible.

The TSO memory model as defined in SPARC V9 [22] is slightly different from the above axioms in 2 points:

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

It can be shown, however, that the 2 seemingly different definitions of the TSO model are essentially equivalent for verification purpose, i.e. any execution trace which satisfies the axioms of either system also satisfies the other. (The proofs are outside the scope of this paper and not included.) For ease of understanding and implementation, we will use the following (stricter) versions of the Order and Atomicity axioms per the definitions of the SPARC V9 architecture [22] throughout the rest of this paper:

Order: There is a total order over all memory operations, i.e. ≤ has a total order.

Atomicity: Atomicity requires that there be no intervening memory operations between the load and store components of an atomic operation.

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

In addition to above axioms, we add an axiom pertaining to memory barriers *M* [22]*:*

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

3. THE VTSO PROBLEMS

Employing the same terminology used by Gibbons and Korach [8], we define the following problems based on the TSO memory model.

3.1 VTSO

Instance: A multithreaded program with:

- Known dynamic memory operation sequences for each thread
- The memory location and the written value for each operation that has store semantics
- The memory location and the read value for each operation that has load semantics

Question: Are all the TSO axioms satisfied?

Note: Due to the fact that the Termination axiom does not really specify a bound on how long it takes for a written value to be eventually seen by other processors, this axiom cannot be completely checked using finite test cases. Thus, we will omit this axiom from consideration for the rest of the paper.

3.2 VTSO-read

The VTSO-read problem is the VTSO problem with additional information, called read-mapping, which maps each read operation to the corresponding write operation which created that read value. A VTSO problem where all written values are unique is in effect a VTSO-read problem.

It can be shown that VTSO and VTSO-read are NP-complete as one can always convert a multithreaded program written for the SC model into a program for the TSO model by inserting a memory barrier after every write operation to enforce the SC behavior. i.e. The VSC and VSC-read problems reduce to the VTSO and VTSO-read respectively.

3.3 VTSO-conflict

VTSO-conflict is the VTSO-read problem with additional information specifying the total order of write operations to each memory location.

4. ALGORITHMS

Although the ability to solve the VTSO problem is the most general solution, the lack of crucial information such as read-mapping makes it difficult to design an algorithm that performs well. Therefore, we limit our scope in this work only to the VTSO-read and VTSO-conflict problems.

In our algorithms, an execution result of a multithreaded program is represented by a directed graph, whose nodes represent operations and edges represent ordering relations in the global memory order ≤. Since ≤ is transitive, any path in the graph implies the existence of the ≤ relation between the source and destination of the path. We ignore reflexivity of ≤ by not explicitly adding an edge from each node to itself. Using directed graphs to model relations between memory operations for various purposes can also be found in prior work [4,7,15,16,18].

A synthetic node is added at the root of the graph acting 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; outgoing edges from any node in the set similarly leave from its last node. This automatically ensures that the Atomicity axiom holds for all relations embedded in the graph.

A read-mapping function *w* maps each load to the store which wrote that value. In our implementation, this mapping can be drawn from the fact that every store would write a distinct value. If there exists a load reading a value never written to that memory location, a failure is signaled at the outset. An inverse of this 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.

We first present a polynomial time algorithm for the VTSO-conflict problem as it is the easier problem because more information is available.

4.1 Algorithm for VTSO-conflict

Given a graph representing a program with its execution result, the read-mapping function *w*, and the total order of all stores for each location, edges are added using the following rules.

Static Edges: In the first step, program order edges are added to the graph according to the following 3 rules. These edges are independent of execution results:

A1: $L;Op \Rightarrow L \leq Op$ (LoadOp axiom)

A2: $S;S' \Rightarrow S \leq S'$ (StoreStore axiom)

A3: $S;M;L \Rightarrow S \leq L$ (Membar axiom)

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 two rules are added based on the load results.

A4: $\neg S;L \Rightarrow S \leq L$ (Value axiom)
This follows because S must be in one of the two store sets in the Value axiom for L.

A5: $S';L \Rightarrow S' \leq S$ (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'.

Value Ordering Edges: With the knowledge of the write order, we finally add these edges:

A6: $S \leq S'$ according to the known write order per location, which is a total order.

A7: $S \leq S' \Rightarrow L \leq S'$ (Value axiom) ; for all L reading the value written by S. We only need to consider S' that immediately follows S for that location.

Rule A7 enforces the Value axiom by making sure that S must be the most recent (*Max*-≤) store for L because every store ordered after S will be ordered after L also. (If not, $S' \leq L$ because there is a total order on all operations and it would be illegal for L to read the value written by S)

After all the edges have been added, a cycle exists *if and only if* there is a TSO violation.

Proof: The reasons to justify the existence of edges added by these rules are already given above. If a cycle exists in the graph, ≤ is not a valid order (anti-symmetry property is violated.) In other words, the Order axiom is violated and, hence, a TSO violation. If a cycle does not exist in the graph, a topological sort of the graph gives a valid total order on all operations that satisfies all the TSO axioms, as shown below (we leave the Termination axiom unchecked as noted earlier; it is vacuously satisfied in any case):

Order: The topological sort gives a total order on all operations.

Atomicity: Enforced by the way edges are added to/from the atomic pairs.

LoadOp, StoreStore, Membar: Satisfied by A1 to A3

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

For each L, let $S = w(L)$. We will show that:

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

Case1: $S;L$

Rule A5 ensures that if there exists another S' such that $S';L$, then $S' \leq S$ and S is still the $\underset{\leq}{Max}\{S_a^i | S_a^i ; L_a^i\}$

Rule A7 ensures that every S' ordered after S is excluded from $\{S_a^k | S_a^k \leq L_a^i\}$

Case2: $\neg\, S;L$

Rule A5 ensures that $\underset{\leq}{Max}\{S_a^i | S_a^i ; L_a^i\} \leq S$

Rule A4 ensures that $S \leq L$

Rule A7 ensures that S is the $\underset{\leq}{Max}\{S_a^k | S_a^k \leq L_a^i\}$

Therefore, $S = \underset{\leq}{Max}[\{S_a^k | S_a^k \leq L_a^i\} \cup \{S_a^i | S_a^i ; L_a^i\}]$ in both cases and the Value axiom holds true. ◊

It may appear at first glance that rule A5, which adds edges between stores, is not needed since a total write order per location is already provided. However, it is indeed required since the given write order may specify a relation which conflicts with the Value axiom, and this rule will catch the conflict.

Rule A4 and A5 are critical for VTSO-conflict, and they are different from the simple addition of $S \leq L$ in VSC-conflict. This is the consequence of allowing loads to overtake their preceding stores in program order in the TSO model. Another way to view this difference is that the Value axiom of SC has only one term, that is: $Val[L_a^i] = Val[\underset{\leq}{Max}\{S_a^k | S_a^k \leq L_a^i\}]$

Note, however, that these rules for VTSO-conflict will work unchanged for VSC-conflict since $S;L \Rightarrow S \leq L$ in SC.

Time Complexity: The time complexity of this algorithm is O(n), where n is the number of nodes in the graph (the total number of operations), assuming appropriate data structures. The number of edges that can be added by each rule is bounded by O(n), and therefore the total number of edges in the graph is also O(n); the final topological sort is O(n+e) where e is the number of edges. This results in total running time which is O(n). This also shows that the VTSO-conflict problem is a P problem.

4.2 Algorithm for VTSO-read

We now present a polynomial-time algorithm for the VTSO-read problem, which also appears in our previous work [10]; the following sections describe extensions to this basic algorithm. Given a graph representing a program with its execution result and the read-mapping function w, edges are added using the following rules:

Static Edges: Rule **R1, R2** and **R3** are the same as **A1, A2** and **A3** respectively.

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: Rule **R4** and **R5** are the same as **A4** and **A5** respectively.

Inferred Edges: Even though the write order is not known, we can still infer edges similar to *Value Ordering Edges* using these 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 because L cannot read the value written by S as it would have already been overwritten by S'. This rule can be viewed as another form of writing **A7**, by negating both terms on the left and the right of ⇒ (assuming ≤ has a total order) and swapping their positions.

R7: $S \leq S' \Rightarrow L \leq S'$ (Value axiom)
This is essentially the same as **A7**, except for the complication that, in the VTSO-read problem, we do not know which S' is the one immediately follows S in the per-location total order, and therefore we need to apply this rule for all currently applicable stores S'.

To apply rule R6, the set of all possible S' such that $S' \leq L$ can be found by traversing the graph backward from L to check all its predecessors known at that time. Similarly, to apply rule R7, traversing the graph forward from S will reach all S' such that $S \leq S'$. However, this forward and backward graph traversal depends on predecessors and successors of the nodes in global order, which is still in the process of being derived. To overcome this problem, 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. Figure 1 outlines the algorithm to iterate over all the rules.

```
Input: A per 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:

Add edges according to rules R1 to R5
[rule R6 and R7] - done in iterations
do
  for each load L
    S := w(L)
    recursively trace all store predecessors S' of L:
      if S' != S and they write to the same address then
        add edge S'→S
      end if
  end for
  for each store S
    recursively trace all store successors S' of S:
      if S' and S write to the same address then
        add edge L→S' for all loads L reading value written by S
      end if
  end for
until no more edges can be added
```

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

Time Complexity: A simple, pessimistic upper bound on the time complexity of this algorithm is $O(n^5)$, where n is the number of nodes in the graph (the total number of operations): 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 the graph for each pair (rule R6 and R7). Thus, the algorithm in this section has polynomial running time independent of the number of processors. However, it is incomplete, as will be explained in Section 4.5.

4.3 Example

Figure 2 illustrates an example (reproduced from our previous work [10]) of a 4-thread program outcome for the VTSO-read problem which violates TSO; it is correctly detected as a violation by the algorithm described in the previous section. There are 2 memory locations involved: A and B. The notation for this example and the following examples in this paper is: S[A]#1 refers to a store which writes value 1 to location A, while L[B]=92 refers to a load to address B which reads value 92. Figure 3 illustrates graphically how a cycle is formed in the analysis graph.

- First, static edges E1, E2, and E3 are added using rules R1 and R2 which simply establish program order relationships using the LoadOp and StoreStore axioms.
- Next, observed edges E4 to E7 are added by applying rule R4 to all load nodes in the graph. Note that rule R4 does not create an edge from S[B]#92 to L[B]=92 on P2 because they are on the same processor.
- Next, observed edge E8 is added by applying rule R5 to L[A]=2 on P0
- Next, during application of rule R6 for the load L[B]=92 on P2, S[B]#91 is found to be one of its predecessors (this is due to the presence of edge E8); this lets us add inferred edge E9.
- Finally, tracing the predecessors of L[B]=91 on P3 leads us to S[B]#92, giving us the inferred edge E10 (rule R6).

P1	P2	P3	P4
S[B]#91	*S[A]#2*	*S[B]#92*	*L[B]=92*
S[A]#1		*L[A]=2*	*L[B]=91*
L[A]=2		*L[B]=92*	

Figure 2. An example program outcome which violates TSO

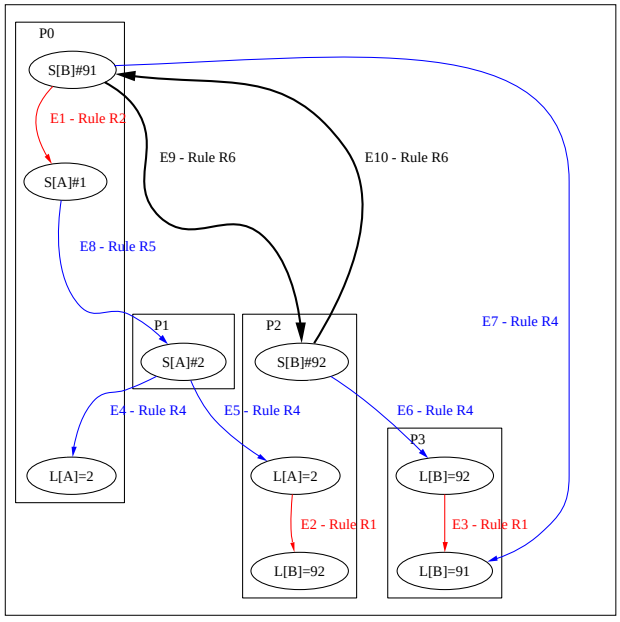

Figure 3. Inferred edges for the program in Figure 2

A cycle in the graph (shown in bold) is formed by edges E9 and E10 indicating a conflicting order between S[B]#91 and S[B]#92, a TSO violation.

4.4 Vector Clocks and VTSO-read

Vector clocks and timestamps are widely used techniques in reasoning about distributed computing [2,13]. They are used mainly for tracking the ordering of events or messages from other processes that each process currently perceives. We emphasize that vector clocks in our implementation are offline constructs used to reason about available ordering information at some point during the analysis; they are not actually present during test execution.

The concept of vector clocks can be applied directly to SC because program order implies memory order and, thus, given $Op^i \leq Op^j$, all operations from processor j after Op^j in program order are also ordered after Op^i from processor i in memory order. To find all successors of Op^i, therefore, we only need to keep track of the earliest Op^j in program order such that $Op^i \leq Op^j$ for all processors j. Furthermore, to find all $S' \leq L$ and $S \leq S'$ in the graph (the conditions of rule R6 and R7), it is sufficient to start at each store node and search only for its earliest successors, which could be loads or stores, that read or write to the same location but with values different from the one it writes. This observation helps us bound the graph traversal while we iterate over rule R6 and R7. For example, in rule R7, instead of traversing the whole graph to find all S' which succeed S in memory order, we only need to consider the earliest such S' in each thread. With a data structure representing a reverse time vector clock[1], and using pointers to actually link nodes rather than simply keeping timestamps, the time complexity for applying rule R7 to each store S in the graph is now $O(k)$ instead of $O(n+e)$ where k is the total number of processors, n is the number of nodes, and e is the number of edges.

However, in the TSO memory model, program order does not imply memory order since a load can overtake its preceding stores. Nevertheless, program order among stores implies memory order, and similarly for loads; therefore, we can split the instruction stream of one TSO processor into two *virtual SC* processors; one contains only stores and the other contains only loads. Note that program order $L;S$ also implies $L \leq S$ in TSO and we shall represent this with an oracle edge between these L and S which are now separated in the two *virtual SC* processors.

Figure 4 outlines the modified algorithm for rule R6 and R7.

Time Complexity: The number of iterations, bounded by the total number of possible edges, is $O(n^2)$. In each iteration, there are $O(n)$ stores whose vector clocks will be traced with $O(k)$ time complexity each. This totals to $O(kn^3)$.

[1] Reverse time vector clocks track successors, while vector clocks track predecessors.

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 $x \le x.rtvc[i]$. Initial *rtvc[]* for all nodes are precomputed with backward topological sort.

[rule R6 and R7] - done in iterations
do
  for each store *S*
    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 $w(L) \neq S$
        [rule R6]
        add edge $S \rightarrow w(L)$ if not already $S \le w(L)$
        update *S.rtvc[]*
      else (virtual SC processor *i* contains only stores and membars)
        S' := first store that accesses same location as *S* and *x*;*S'*
        [rule R7]
        for all loads *L* such that $w(L)=S$
          add edge $L \rightarrow S'$ if not already $L \le S'$
          update *L.rtvc[]*
        end for
      end if
    end for
  end for
until no more edges can be added

Figure 4. High level description of the iteration over R6 & R7 with Vector Clocks

4.5 Heuristic for VTSO-read

In the absence of cycles in the graph, the polynomial time algorithms of Figure 1 and Figure 4 create a global order relation which is consistent with the LoadOp, StoreStore, Membar, Value and Atomicity axioms. However, the algorithms are incomplete because they do not explicitly ensure that the Order axiom is satisfied. To satisfy the Order axiom, we would have to identify unordered writes at the end of our algorithms and search for a combination of relations between them which is compatible with the results; this search could make the runtime exponential in the worst case, and the analysis time impractically large. By not explicitly enforcing the Order axiom, our algorithms trade off accuracy for reasonable analysis time.

Figure 5 illustrates a case where an existing relation is not inferred by our analysis algorithms; the edges in the graph are depicted at the point when the algorithm of Figure 1 (or Figure 4) has reached a fixed point and terminated. Notice that S[A]#1 and S[A]#2 are left unordered. 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 3 or 4, can survive after S[A]#2 in location B, the two loads from location B must read the same value. While this example is not yet a missed TSO violation, adding a similar, mirrored set of nodes to a different location C (two stores to C ordered before S[A]#1, and two loads to C ordered after S[A]#2) creates an instance of a TSO violation which is missed by our algorithms.

To increase the probability of finding a valid total order and thereby remove the source of incompleteness, we adopt the following heuristic. After each complete iteration of applying rules R6 and R7 to all the nodes in the graph, we can hypothetically perform a topological sort and extract the resulting

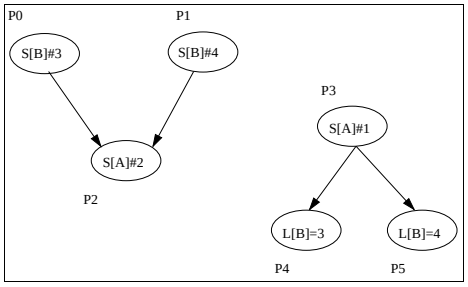

Figure 5. An example when the algorithm misses an edge

write order per location. Once a possible write order per location is available, we can apply the fast linear time algorithm in Section 4.1, assuming the VTSO-conflict problem. We actually combine these two steps into one in our implementation by applying rule A7 as soon as a new write order is assumed during the topological sort. Of course, if the heuristic finds a TSO violation, we cannot conclude that it is real as it may be the consequence of the improperly assumed write order; we must continue with the original algorithm, and if the original algorithm has reached a fixed point, then the analysis stops without a total ordering. In practice, as more edges get added due to rule R6 and R7 they tend to capture the essence of the memory ordering such that the heuristic has a higher probability of finishing with a valid total operation order (TOO). If this happens, our polynomial time algorithm has completely certified that the program outcome is valid under the TSO axioms; in addition, the iteration over rule R6 and R7 can stop early resulting in improvement in overall run time. If the heuristic does not find a total ordering, we still flag the test as a pass, since such an ordering is assumed to exist though we were not able to find it easily.

We find that, in practice, this heuristic not only helps improve completeness of our polynomial time algorithm, it also improves its running time because we often find a valid total order even before the previous algorithm has reached a fixed point. Consider again the completeness of analysis for the missing edge in Figure 5. Although the edge from S[A]#1 to S[A]#2 was missed by the analysis algorithm, by applying rule A7 as soon as one of the stores S[B]#3 and S[B]#4 is picked, we ensure S[A]#1 is picked before S[A]#2 in the heuristic, effectively discovering the missing edge.

4.6 Other Memory Models

The analysis algorithms described in this section can be modified to verify other memory models as well. For example, in SC, all the rules remain the same (as noted in Section 4.1), except for the additional requirement between stores and loads on the same processor. Therefore, the only difference lies in the initial set of edges determined from program order and the application of the remaining rules remains the same. For more relaxed memory models, more than 2 virtual SC processors may be required for one original instruction steam. For example, in Relaxed Memory Order (RMO), only program order among stores to the same location implies memory order, and therefore, the number of virtual SC processors depends on the number of shared memory locations used by the program.

5. RESULTS

We have implemented these algorithms as a key enhancement to our previous work in the context of TSOtool, a program that generates pseudo-random, multithreaded test programs with aggressive data races and analyzes their execution results to

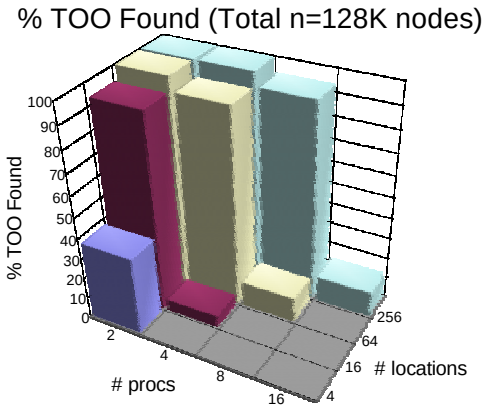

Figure 6. Effectiveness of the heuristic in Section 4.5

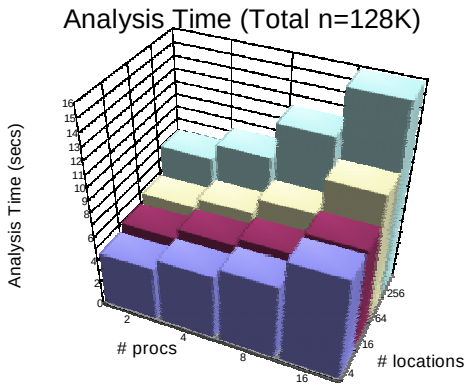

Figure 7. Analysis Time

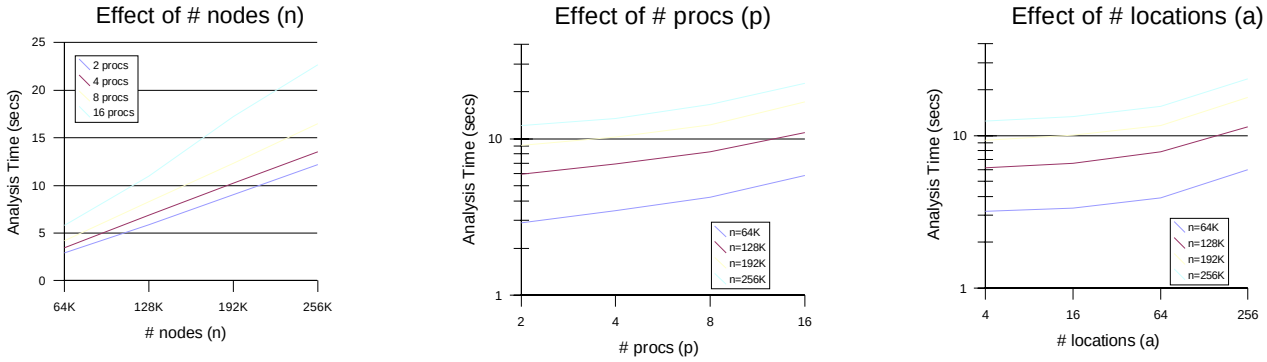

Figure 8. Analysis Time vs. *n* (averaged over *a*), *p* (averaged over *a*), and *a* (averaged over *p*)

detect TSO violations [10]. It is used in many phases of design and validation of new microprocessors in Sun Microsystems, and has been successful in finding hundreds of bugs in the design of nine different microprocessors, and shared memory multiprocessor systems based on them. All stores in TSOtool generated test programs write unique values, ensuring that the analysis has to deal only with the VTSO-read or the VTSO-conflict problems. Whenever possible, with sufficient observability about per-location write-order (for example, in simulation environments), the tool works on the the VTSO-conflict problem and can therefore employ the algorithm in Section 4.1 to generate complete results in linear time; however, in the general case, there is limited observability (for example, on real multiprocessor systems or hardware accelerated simulators), and TSOtool employs the algorithm in Section 4.2 along with the vector clocks extension in Section 4.4 for the VTSO-read problem and the TOO heuristic in Section 4.5. Analysis time is the main bottleneck in the number of tests which can be run on real multiprocessor systems: test threads are pre-generated, so selecting a set of test threads and running a 128K operation test takes only a few milliseconds, while analyzing the result takes a few orders of magnitude more time.

For the VTSO-read problem, applying the concept of vector clocks to the analysis algorithm gives us a speedup of about 30X over our original implementation of TSOtool, allowing us to dramatically increase our test throughput on large multiprocessor systems. Applying the heuristic in Section 4.5 to the problem yields approximately another 2X speedup whenever it is able to stop the iteration of rules R6 and R7 early because a valid TOO is found before the analysis algorithm has reached a fixed point.

To understand how well these algorithms perform in practice, we carried out experiments on an otherwise unloaded 24-way 1.2GHz UltraSparc-III+-based Sun multiprocessor system. We used TSOtool to generate random multithreaded programs with the following instruction mix: 33.3% loads, 33.3% stores, 30% atomics, 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 then analyzed using one of the processors on the same machine with the algorithms described in this paper (the analysis itself is serial, not parallel.) For each tuple (*n*,*p*,*a*), 32 pseudo-random programs were generated with different seeds, then executed, and analyzed. In Fig. 7 and Fig. 8, the average analysis time over all 32 runs is plotted for each tuple.

The percentage of times that the heuristic found a valid TOO while analyzing different programs for *n*=128K nodes is shown in Figure 6, and the corresponding analysis time is shown in Figure 7. The heuristic tends to successfully find a valid TOO less frequently when the number of locations is relatively small compared to the number of processors. This is due to increased access contention, resulting in more store values that are never read, so the heuristic has relatively less information in order to make a successful guess. Figure 8 illustrates the effect of *n*, *p*, and *a*, respectively. Note that the analysis time scales almost

linearly with all these variables although the bound is $O(kn^3)$ in theory. However, the size of our data structure also scales with all *p*, *a*, and *n*, and a large memory footprint can adversely affect performance.

Overall, our analysis algorithm runs in the order of seconds while the original algorithm runs in minutes when compared at n=64K.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we extend the study of the VSC problems to a more relaxed memory model, specifically, the TSO model and the corresponding VTSO problems. Although the discussion is based on the TSO model, the presented ideas can be applied to other memory models as long as they can be similarly defined in terms of a set of formal axioms.

Our polynomial time algorithm for VTSO-read, along with an extension employing vector clocks and a heuristic to improve completeness, has been useful and effective in quickly analyzing multithreaded program executions to detect bugs in the design of real, complex multiprocessor systems. In future, it would be valuable to further analyze the algorithm's quality in terms of completeness and gain insights into cases where it is incomplete despite our heuristic. Assistance in finding the root-cause of a detected violation will also be very helpful, since errors detected by this tool tend to be very subtle.

7. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments, Robert Cypher for many helpful discussions regarding this work; Ravi Hosabettu for a related proof on atomic transactions; Joseph Lu, Shrenik Mehta, Sridhar Narayanan, Mike Splain, and for their help in the initial stages of TSOtool development. Durgam Vahia, Aleksandr Gert, Rohit Kumar and Gopal Reddy have contributed substantially to the current implementation of TSOtool.

8. REFERENCES

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

[2] R. Baldoni and M. Raynal, *Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems*, IEEE Distributed Systems Online, Vol. 3, No. 2, 2002

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

[4] H.W. Cain, M.H. Lipasti and R. Nair, *Constraint Graph Analysis of Multithreaded Programs*, in Proceedings of the Twelfth International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003

[5] J.E. Cantin, M.H. Lipasti and J.E. Smith, *The Complexity of Verifying Memory Coherence*, in Proceedings of the Fifteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2003

[6] A.E. Condon, M.D. Hill, M. Plakal, and D.J. Sorin, *Using Lamport Clocks to Reason About Relaxed Memory Models*, in Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture (HPCA), 1999

[7] A.E. Condon and A.J. Hu, *Automatable verification of sequential consistency*, in Proceedings of the Thirteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA), 2001

[8] P. B. Gibbons and E. Korach, *The complexity of Sequential Consistency*, in Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing (SPDP), 1992

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

[10] S. Hangal, D. Vahia, C. Manovit, J. Lu, S. Narayanan, *TSOtool: A Program to Verify Multiprocessor Memory Systems Using the Memory Consistency Model*, in Proceedings of the 31st International Symposium on Computer Architecture (ISCA), 2004

[11] T.A. Henzinger, S. Qadeer, and S.K. Rajamani, *Verifying Sequential Consistency on Shared-memory Multiprocessor Systems*, in Proceedings of the 11th International Conference on Computer-Aided Verification (CAV), 1999

[12] L. Higham, J. Kawash, and N. Verwaal, *Defining and Comparing Memory Consistency Models*, in Proceedings of 9th International Conference on Parallel and Distributed Computing and Systems (PDCS), 1997

[13] L. Lamport, *Time, Clocks and the Ordering of Events in a Distributed System*, Communications of the ACM, 21(7):558-565, July 1978

[14] L. Lamport, *How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, C-28(9):241-248, September 1979

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

[16] D. H. Linder and J.C. Harden, *Access Graphs: A Model for Investigating Memory Consistency*, IEEE Transactions on Parallel and Distributed Systems, Vol. 5, No. 1, January 1994

[17] R. Nalumasu, R. Ghughal, A. Mokkedem, and G. Gopalakrishnan, *The `Test Model-checking' Approach to the Verification of Formal Memory Models of Multiprocessors*, in Proceedings of the 10th International Conference on Computer-Aided Verification (CAV), 1998

[18] S. Qadeer, *On the verification of memory models of shared-memory multiprocessors*, in Proceedings of the 12th International Conference on Computer Aided Verification (CAV), 2000

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

[20] R.C. Steinke and G.J. Nutt, *A Unified Theory of Shared Memory Consistency*, Journal of the ACM (JACM), Volume 51, Issue 5, September 2004

[21] D.J. Sorin, M. Plakal, M.D. Hill, and A.E. Condon, *Lamport Clocks: Reasoning About Shared-Memory Correctness*, Technical Report CS-TR-1367, University of Wisconsin-Madison, March 1998

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