Fast and Generalized Polynomial Time Memory Consistency Verification

Amitabha Roy, Stephan Zeisset, Charles J. Fleckenstein, and John C. Huang

Intel Corporation
{amitabha.roy, stephan.zeisset, chuck.fleckenstein}@intel.com,
jhuangtw@umich.edu

Abstract. The problem of verifying multi-threaded execution against the memory consistency model of a processor is known to be an NP hard problem. However polynomial time algorithms exist that detect almost all failures in such execution. These are often used in practice for microprocessor verification. We present a low complexity and fully parallelized algorithm to check program execution against the processor consistency model. In addition our algorithm is general enough to support a number of consistency models without any degradation in performance. An implementation of this algorithm is currently used in practice to verify processors in the post silicon stage for multiple architectures.

1   Introduction

Verifying processor execution against its stated memory consistency model is an important problem in both design and silicon system verification. Verification teams for a microprocessor are often concerned with the memory consistency model visible to external customers such as system programmers. In the context of multi-threading, both in terms of Simultaneous Multi Threading(SMT) and Chip Multi Processing(CMP), Intel[®][1] and other CPU manufacturers are increasingly building complex processors and SMP platforms with a large number of execution threads. In this environment the memory consistency model of microprocessors will come under close scrutiny, particularly by developers of multi-threaded applications and operating systems. Allowing any errors in implementing the consistency model to show up as customer visible is thus unacceptable. The problem we are concerned with is that of matching the result of executing a random set of load store memory operations distributed across processors, on a set of shared locations, against a memory consistency model. The algorithm should flag an error if the consistency model does not allow the observed execution results. This forms the basis for Random Instruction Test(RIT) generators such as TSOTOOL[2] [1] and Intel's Multi Processor(MP) RIT environment. The Intel MP RIT Tool incorporates the algorithm in this paper. Formally, we concentrate on variations of the VSC (Verifying Sequential Consistency) problem [2]. The VSC problem is exactly the problem described above, when restricted to sequential consistency. The general

---

[1] Intel[®]is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[2] Other names and brands may be claimed as the property of others.

VSC problem is NP complete [3]. The general coherence problem has also been shown to be NP complete [4]. A formulation of VSC for more general memory consistency models was done in [1] where a polynomial time algorithm was presented for verifying a memory consistency model at the cost of correctness, although the incorrect executions missed were shown to be insignificant for the purpose of CPU verification. That work focused almost exclusively on the Total Store Order(TSO) memory consistency model and presented a worst case $O(n^5)$ algorithm. In this work, we present an efficient implementation of the basic algorithm in [1]. Our key contribution is to reduce the worst case complexity to $O(n^4)$ for *any* memory consistency model using $\Theta(n^2)$ space. Although the work in [5] has reduced the complexity to $O(kn^3)$ where k is the number of processors, that algorithm assumes the TSO memory consistency model and does not generalize to other models. Our motivation for generalizing and improving it is Intel's complex verification environment, where microprocessors support as many as five different consistency models at the same time. The primary objectives of our algorithm design are simplicity, performance and seamless extendibility in the implementation to any processor environment, including the Itanium®[3]. Another goal is enhanced support for debugging reported failures, which is crucial to reducing time to market for complex multi processors.

The algorithm we have developed is currently implemented in Intel's in house random test generator and is used by both the IA-32 and Itanium verification teams. We also present scalability results and a processor bug that was caught by the tool using this algorithm.

2   Memory Consistency

Consider a set of processors each of which executes a stream of loads and stores. These are done to a set of locations shared across the processors. We are concerned with a global ordering of all the loads and stores, which when executed serially leads to the same result. The strictest consistency model is the sequential consistency (SC) model which insists that the only valid orderings are those that do not relax per processor program order between the memory operations. Relaxing restrictions between operations such as stores and loads leads to progressively weaker models such as Total Store Order (TSO) and Release Consistency (RC). All these are surveyed in [6]. We point out that in these orderings we refer to load executions and store executions. A load is considered performed(or executed) if no subsequent store to that location(on any processor) can change the load return value. A store is considered performed(or executed) if any subsequent load to that location (on any processor) returns its value. These are definitions from [7]. Any instruction on a modern pipelined processor has a number of phases and some, such as instruction fetch and retirement, occur in strict program order without regard to the memory consistency model. We are concerned only with ordering the load and store execution phases for instructions referring to memory.

---

[3] Itanium®is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

2.1   Formalism

The terminology used in this paper is similar to [1]. We use ; to denote program order and $\leq$ to denote global order. Thus $A; B$ and $A \leq B$ mean that B follows A in program order and global order respectively. The fundamental operations in our test consist of L_a^i and S_a^i which are loads and stores respectively to location a by processor i. We also consider $[L_a^i; S_a^i]$ which is an atomic load store operation. Examples are XCHG in IA-32 [8] and FETCHADD in Itanium [9]. We use $val(L_a^i)$ to denote the load return value of a load operation and $val(S_a^i)$ to denote the value stored by a store operation.

For any location a we define the type of a location to be $Type(a) \in \{WB, WT, WP, UC, WC\}$. The type of a location is the *memory* type of the location. IA-32 [8] supports all five memory types, Write Back (WB), Write Through (WT), Write Protect (WP), Write Combining(WC) and Uncacheable. Itanium [9] supports only three, WB, WC and UC. In addition to cacheability and write through implications of these memory types, they also affect the consistency model.

2.2   Axioms and Orders

Both $\leq$ and ; are transitive, reflexive and antisymmetric orders. The program order is limited to operations on the same processor while the global order covers all operations across all processors. We also define $A < B$ to mean $A \leq B$ and $A \neq B$.

We define the following axiom to support atomic operations.

Axiom 1 (Atomic Operations). $[L_a^i; S_a^i] \Rightarrow (L_a^i \leq S_a^i) \bigwedge (\forall S_b^j : (S_b^j \leq L_a^i) \bigvee (S_a^i \leq S_b^j))$

As a result of this, we can treat atomic operations as a single operation for verification. We assume the following two axioms to hold, the bare minimum to be able to use the basic algorithm proposed in [1].

Axiom 2 (Value Coherence). $val[L_a^i] \in \{val[\overset{Max}{\leq} S_a^k | S_a^k < L_a^i], val[\overset{Max}{;} S_a^i | S_a^i; L_a^i]\}$

The value returned by a read is from either the most recent store in program order or the most recent store in global order. This is intuitive for a cache coherent system. Note that the most recent store in program order may not be a preceding store in global order. This is because many architectures including Intel ones can support the notion of store forwarding, which allows a store to be forwarded to local loads before it is made globally visible. Also, in the test a load may occur before any store to that location in which case it returns the initial value of that location. Such cases are handled by assuming a preliminary set of stores that write initial values to locations. The store values to a location and initial value of the location are chosen to be unique by the test generator. This allows the axiom to be applied after the test is completed to link a load to the store that it reads.

Axiom 3 (Total Store Order). $\forall S_a^i, S_b^j ((S_a^i \leq S_b^j) \bigvee (S_b^j \leq S_a^i))$.

Unlike [1], we have avoided imposing any additional constraints between operations on the same processor. Rather, we allow these constraints to be dynamically specified. This allows us to parameterize the same algorithm to work across CPU architectures (Itanium and IA-32) and processor generations (Intel NetBurst®[4] and P6 in the case of IA-32).

Define $Ops = \{L, S, X\}$ to be the allowed types of an operation. Thus we can define $Type(L_a^i) = L$, $Type(S_a^i) = S$ and $Type([L_a^i; S_a^i]) = X$. We also define $Loc(Op)$ to return the memory location used by the operation. For example $Loc(L_a^i) = a$.

We can then define the constraint function
$f : (Ops X \{WB, WP, WT, WC, UC\})^2 \rightarrow \{0, 1\}$. This is used to impose the dynamic set of constraints:

Definition 1 (Local Ordering). *$[O_1; O_2$ and*
$f((Type(O_1), Type(Loc(O_1)), (Type(O_2), Type(Loc(O_2)))) = 1] \Rightarrow O_1 \leq O_2$
If the LHS of the implication is satisfied we call O_1 and O_2 as locally ordered memory operations.

As an example, from [8] we know that Write back stores do not bypass each other. Hence f((S, WB),(S,WB))=1. However, write combining stores are allowed to bypass each other and hence f((S, WC), (S,WC))=0. There are other more subtle orderings which vary between processor generations and in this case we obtain appropriate ordering functions from the CPU architects or designers.

3   Algorithm

Our objective is an algorithm that takes in the result of an execution and flags violation of the memory consistency model. The basic algorithm in [1] that we extend uses constraint graphs to model the execution. There have been similar approaches in the past too, such as [10] and an approach to the same problem using Boolean satisfiability solvers [11], which models write atomicity accurately, but can handle only much shorter executions than our method can handle. We model the execution as a directed graph G=(V, E) where the nodes represent memory operations and the edges represent the $\leq$ global order. However, as in [1], we do not put self edges although the relation is reflexive. Thus if $O_1 \leq O_2$ then we add an edge from the node for O_1 to that for O_2. For brevity, we refer to operations and their corresponding nodes by the same name. $A \rightarrow B$ means there is an edge from A to B while $A \rightarrow_P B$ means there is a path from A to B.

Based on the per processor ordering imposed by our ordering function f, we can immediately add static edges to the graph.

Rule 1 (Static Edges). *For every pair of nodes O_1 and O_2 such that they are locally ordered by definition 1, add the edge $O_1 \rightarrow O_2$.*

---

[4] Intel NetBurst®is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

After execution of the test, we determine a function $Reads$ in a preprocessing step (operating on loads) such that $Reads(L_a^i) = S_a^j$ if L_a^i reads S_a^j. Otherwise (the case where the initial value for the location is read), $Reads(L_a^i) = Sentinel$, a special sentinel node. We add edges from $Sentinel$ to all other store nodes in the graph. This is the same construction as described in [1]. From the value axiom we know that any read that returns the value of a remote write must have occurred after the remote write has been globally observed. This allows us to add observed edges to the graph based on the values returned by the loads in the test. Note that for the rules below we treat an atomic operation as both a load and a store.

Rule 2 (Observed Edge). *For every load L_a^i, if $Reads(L_a^i) = S_a^j$ where $i \neq j$, or if $Reads(L_a^i) = Sentinel$, add the edge $Reads(L_a^i) \rightarrow L_a^i$. Note that since stores to the same location write unique values and all locations are initialized to hold unique values, value equivalence means that the load must have read that store.*

The next few set of edges are essentially inferred from the value axiom. Hence they are called inferred edges.

Rule 3 (Inferred Edge 1). *If $Reads(L_a^i) = S_a^j$ and $i \neq j$ then for every S_a^i such that $S_a^i; L_a^i$ add the edge $S_a^i \rightarrow S_a^j$. This follows from the value axiom since the alternative global order would mean the load should read the local store.*

Rule 4 (Inferred Edge 2). *If $Reads(L_a^i) = S_a^j$ then for every S_a^k such that $S_a^k \rightarrow_P L_a^i$ and $S_a^k \neq S_a^j$, add the edge $S_a^k \rightarrow S_a^j$. This follows from the value axiom since the alternative global order would mean that the load should read S_a^k.*

Rule 5 (Inferred Edge 3). *If $Reads(L_a^i) = S_a^j$ then for every S_a^k such that $S_a^j \rightarrow_P S_a^k$ add the edge $L_a^i \rightarrow S_a^k$. This follows from the value axiom since the alternative global order would mean that the load should read S_a^k.*

3.1 Basic Algorithm

The basic algorithm described in [1] can now be summarized as follows:

1. Compute the $Reads$ function in a preprocessing step.
2. Apply rule 1 to add all possible edges.
3. Apply rule 2 to add all possible edges.
4. Apply rules 3, 4 and 5.
5. If any edges were added in step 4 go back to step 4 else go to step 6
6. Check the graph for cycles. If any are found, flag an error.

An example of this algorithm applied to an execution is shown in Figure 1. We use the notation $S[X]\#V$ for write V to location X, and $L[X] = V$ for read from location X returns value V.
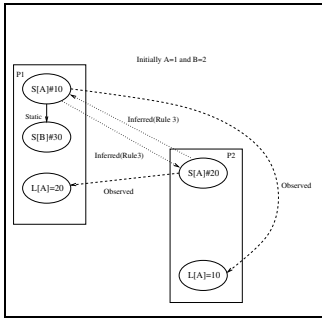
Fig. 1. Example of an incorrect execution with graph edges added

Computing the $Reads$ function is $O(n^2)$ since we need to examine all pairs of loads and stores. Steps 2 and 3 are of cost $O(n^2)$ since we examine all pairs of nodes. Step 4 involves determining the relationship $A \rightarrow_P B$ for $O(n)$ nodes. This costs $O(n^2)$ for each node (assuming a depth first search, as one of the obvious options) and hence $O(n^3)$ overall. Since the fixed point iteration imposed by steps 4 and 5 may loop for at most $O(n^2)$ adding one edge on each iteration, we have a worst case complexity of $O(n^5)$. The detailed analysis is in [1]. There has been a subsequent improvement published in [5] that reduces the complexity to $O(kn^3)$. Its correctness requires that there are a constant number of ordered lists on each processor. This is true because all loads and all stores are ordered on a processor in the TSO consistency model that they have considered. Unfortunately this does not hold true for both the IA-32 [8] and Itanium [12] memory models for various memory types (consider WC stores). Hence the formulation in [5] is not general enough.

3.2   Graph Closure

The primary contributor to the $O(n^5)$ complexity is deciding whether $A \rightarrow_P B$ holds. All other operations can be efficiently implemented and do not seem to hold any opportunity for improvement, given our goal of generality. Hence, we decided to focus on the problem of efficiently determining $A \rightarrow_P B$. A solution is to compute the transitive closure of the graph. We first label all the nodes in the directed graph under consideration, $G = (V, E)$ by natural numbers using the bijective mapping function $g : V \rightarrow \{1..n\}$ where $| V | = n$. We can then represent E by the familiar n square adjacency matrix A such that $(U, V) \in E \Leftrightarrow A[g(U), g(V)] = 1$.

For transitive closure of the graph we seek the closed form of the adjacency matrix A such that $U \rightarrow_P V \Leftrightarrow A[g(U), g(V)] = 1$. A well known algorithm for computing the transitive closure of a binary adjacency matrix is Warshall's algorithm[13]. Before giving Warshall's algorithm, we first define some convenient notation and functions to transform the connectivity matrix. $AddEdge(x, y)$ stands for : set $A[x, y] = 1$. $Subsume(x, y)$ is defined as $\forall z$ such that $A[y, z] = 1$, $AddEdge(x, z)$. The subsume function causes all neighbors of node $g^{-1}(y)$ to also become neighbors of node $g^{-1}(x)$ in the adjacency matrix representation.

Incremental Graph Closure: Although Warshall's algorithm will compute the closed form of the adjacency matrix, any edge added by $AddEdge$ will cause the matrix to lose this property since new paths may be available through the added edge. Hence we need an algorithm which when given a closed adjacency matrix and some edges added *efficiently* recomputes the closure.

Warshall's Algorithm:	Incremental Warshall's Algorithm:
for all $j \in \{1..N\}$	for all $j \in \{1..N\}$
for all $i \in \{1..N\}$	for all $i \in \{1..N\}$
if($A[i,j] = 1$)	if($A[i,j] = 1$ and
$Subsume(i,j)$	($Changed[j] = 1$ or $Changed[i] = 1$))
end if	$Subsume(i,j)$
end for	end if
end for	end for
	end for

We assume that when adding edges to any node U, we mark that node as changed by setting the corresponding bit in the change vector $Changed[g(U)] = 1$. We can now rerun Warshall's algorithm *restricted* to only those nodes which have either changed themselves, or are connected in the current adjacency matrix to a changed node. This is shown in pseudo-code as incremental Warshall's algorithm. A correctness proof can be found in [14].

Complexity: An important observation is that the complexity of the incremental update is $O(mn^2)$ where the number of changed nodes is $O(m)$. This is because the subsume step takes $O(n)$ and for each node, $Subsume$ can only be called at worst $O(m)$ times, if it is connected to all the changed nodes. At worst all $O(n)$ nodes satisfy the precondition for subsume and hence the $O(mn^2)$ complexity.

3.3   Final Algorithm

We describe algorithms to implement the rules for adding observed and inferred edges in Table 1. Recall that our graph is G=(V, E) and the vertices correspond to memory operations in the test. Also, for ease of specification we have allowed atomic read modify write operations to be treated as both stores $Type(Op) = S$ and loads $Type(Op) = L$.

The ordering of for loops is not arbitrary as it may appear but rather has been carefully chosen to aid in parallelization as we demonstrate in section 4.

We now state the final algorithm used to verify the execution results. A benefit of our approach is that checking the graph for cycles is simply checking whether $\exists i \, A[i,i] = 1$ since a cycle results in a self loop due to the closure. Additionally, note that we have merged the preprocessing step that links loads to the stores they read, into the step to compute observed edges.

1. Apply rule 1 to add all possible edges.
2. Apply rule 2 to add all possible edges.
3. Apply Warshall's algorithm to obtain the closed adjacency matrix.
4. Apply rules 3, 4 and 5.
5. If any edges were added in step 4 go to step 6 else go to step 8.
6. Apply the incremental Warshall's algorithm to recompute closure and reset the changed vector.
7. Go to step 4.
8. Check the graph for cycles. If any are found, flag an error.

Table 1. Pseudcode of Algorithm for Adding Edges

Algorithm for adding edges:

Static Edges:
for all $O_1 \in V$
  for all $O_2 \in V$ such that $O_1 \neq O_2$
    If O_1 is locally ordered after O_2 as per definition 1then
    $AddEdge(g(O_2), g(O_1))$
  end for
end for

Observed Edges:
for all $O_1 \in V$ such that $type(O_1) = L$
  for all $O_2 \in V$ such that $type(O_2) = S$
    If $val(O_1) = val(O_2)$
      set $Reads(O_1) = O_2$
      If O_2 is on a different CPU from O_1 then $AddEdge(g(O_2), g(O_1))$
    end If
  end for
  If no corresponding store is found for this load then
  $AddEdge(g(Sentinel), g(O_1))$ and set $Reads(O_1) = Sentinel$
end for

Inferred Edge 1:
for all $O_1 \in V$ such that $type(O_1) = L$
  for all $O_2 \in V$ such that $type(O_2) = S$ and $O_2; O_1$ and $O_2 \neq Reads(O_1)$
    If O_2 is on a different CPU from O_1 then
    $AddEdge(g(O_2), g(Reads(O_1)))$ and set $Changed[g(O_2)] = 1$
  end for
end for

Inferred Edge 2:
for all $O_1 \in V$ such that $type(O_1) = L$
  for all $O_2 \in V$ such that $type(O_2) = S$ and $A[g(O_2), g(O_1)] = 1$
  and $O_2 \neq Reads(O_1)$
    $AddEdge(g(O_2), g(Reads(O_1)))$ and set $Changed[g(O_2)] = 1$
  end for
end for

Inferred Edge 3:
for all $O_1 \in V$ such that $type(O_1) = S$
  for all $O_2 \in V$ such that $type(O_2) = L$ and $A[g(Reads(O_2)), g(O_1)] = 1$
    $AddEdge(g(O_2), g(O_1))$ and set $Changed[g(O_2)] = 1$
  end for
end for

3.4   Complexity

The analysis of complexity is straightforward. Each of steps 1 and 2 take $O(n^2)$ since they examine all pairs of nodes. Step 3 takes $O(n^3)$ as is shown in [13]. Each iteration of Step 4 again takes $O(n^2)$ because we examine all pairs of nodes. Note that checking $A \rightarrow_P B$ is now $O(1)$ thanks to the closed adjacency matrix. There are at most $O(n^2)$ edges to be added and hence the worst case complexity for Step 4 is $O(n^4)$. The

remaining analysis is step 6. For this we note that the complexity is also $O(mn^2)$ when considered over *all* invocations. Since $m = O(n^2)$ (bounded above by the number of edges we can possibly add and thereby change nodes), we have $O(n^4)$ as the worst case complexity for step 6. Cycle checking in step 8 is simply $O(n)$ due to the closed form of the adjacency matrix. Thus the overall complexity is $O(n^4)$ which meets our stated goal. Our overall space requirements are clearly $\Theta(n^2)$ due to the adjacency matrix.

4 Parallelization

One of the ways to mitigate the expense of an $O(n^4)$ algorithm is parallelization. With a test size of hundreds of memory operation per CPU, result validation time can easily overwhelm the verification process. For example consider a 4 way SMP platform with hyperthreaded processors with a total of 8 threads and hence 800 operations. The way we have arranged the algorithm and data structures allows us to easily do loop parallelization [15].

Fig. 2. Example of an actual processor bug

The phases of the algorithm are Warshall's algorithm, incremental graph closure and the rule algorithms given in section 3.3. The key observation is that in each case we always have no more than two nested for loops and there are no data dependences between iterations of the inner loop. The latter is true because no two iterations change the same node in the graph and hence never write to the same element in the adjacency matrix. We are not worried about considering edges added in previous iterations of the inner for loop of step 4 (of the algorithm in 3.3) because such edges are considered in subsequent iterations, since we iterate to a fix point. Also the same element in the $Changed$ vector is not accessed by two different inner loop iterations. Hence we can parallelize by distributing different iterations of the inner for loop in each step across processors. Since each inner for loop iterates over all nodes in the graph, this leads to a convenient data partitioning. We allocate each CPU running the verification algorithm a disjoint subset of nodes in the graph. Each CPU executes the inner for loop in each phase only on nodes that it owns. Note that each CPU still needs to synchronize with all other CPUs after completion of the inner for loop in each case (this is similar to the INDEPENDENT FORALL construct in High Performance Fortran).

5 Implementation

Intel's verification environment spans both architecture validation (Pre Silicon on RTL models) as well as extensive testing post silicon with the processor in an actual platform [16]. The algorithm described in this paper has been implemented in an Intel RIT generator, used by verification teams across multiple Intel architectures (Itanium, IA-32
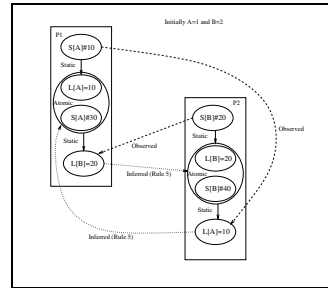
```
Algorithm PrintSomeCycle:
PossibleStart={g⁻¹(i) | A[i,i] = 1}
while PossibleStart is not empty
  StartNode=any node in PossibleStart
  PossibleStart=PossibleStart -{StartNode}
  CurrentList={g⁻¹(i) | A[i,i] = 1} - StartNode
  GetCycleEdge(startNode,startNode)
end while
Function GetCycleEdge:
GetCycleEdge(node Start, node Current)
If Algorithm(Current, Start) returns true
  print edge (Current, Start)
  PossibleStart=PossibleStart -{Current}
  return true
end If
for each node nextNode in CurrentList
  If Algorithm(Current, nextNode) returns true
    CurrentList = CurrentList - {nextNode}
    If GetCycleEdge(Start, nextNode) returns true
      print edge (Current, nextNode)
      PossibleStart=PossibleStart -{Current}
      return true
    end If
  end If
end for
return false
```

Fig. 3. Debug Algorithm

and 64-bit IA-32). Although in the architecture validation (pre silicon on RTL simulators) environment direct visibility into load and store execution allows simpler tools to be built, it has been used in a limited fashion to generate tests that are subsequently run on RTL simulators. The results are then checked by the algorithm to find bugs. The greatest success of the tool has been in the Post Silicon Environment, where the execution speed available (compared to RTL simulations) allows the tool to quickly run a large number of random tests and discover memory ordering issues on processors. In figure 2 we show an example of an incorrect execution corresponding to an actual bug found by this tool. The problem was subsequently traced to incorrect design in the CPU of the locking primitive for certain corner cases.

In the Post Silicon environment the tool has been written to run directly on the Device Under Test(DUT). This was made possible by running it as a process on a deviceless Linux kernel which is booted on the target. The primary advantage of this model is speed and adaptability where the RIT tool directly detects its underlying hardware, generates and executes the appropriate tests and then verifies the result with no communication overhead.Another not so apparent but important advantage is *scaling*. As we anticipate future processors to increase the number of available threads, the tool scales seamlessly by not only running tests on the increased number of threads but also using all available threads to run the checking algorithm itself. This is also the reason why we have paid so much attention to parallelization in this work. That is to allow the algorithm to bootstrap on future generations of multi threaded processors. We point out here that the test generation phase is also parallelized in the tool to make optimal use of resources and achieve the best speedup.

Implementation Environment: The algorithm is implemented in C and architecture dependent assembly that runs on a scaled down version of the Linux kernel. We have chosen to use the Linux process model (avoiding other threading models for simplicity) with shared memory segments for inter process communication. We have hand parallelized the loops using the data distribution concepts described in section 4. This allows

us to use off the shelf compilers such as those in standard Linux distributions and work across all the platforms that Linux supports.

Exploiting SIMD: The key kernel used in the iterative phase of our algorithm is $Subsume$. This is called at least once for every edge added to the graph and improving its performance is clearly beneficial. The implementation for $Subsume(x, y)$ is $\forall z \in \{1..n\} A[x, z] = A[x, z] \vee A[y, z]$. Another way of looking at it is as the logical 'OR' of two binary vectors $A[x, .] = A[x, .] \vee A[y, .]$. This could have taken as many as n operations in the most obvious implementation, but we instead chose to use Single Instruction Multiple Data (SIMD) extensions available in both the IA-32 [8] and Itanium [9] instruction sets. These enable us to perform the subsume operation upto 128 bits at a time providing a 128 times speedup to the implementation of $Subsume$. This is also the only place in our tool where we have IA-32 and Itanium specific verification code. The option to use SIMD to speedup the algorithm is really a consequence of the carefully selected data structures and the time consuming graph manipulations being reduced to a single well defined kernel.

Extendibility: We support multiple architectures in our implementation by having as much architecture independent code as possible. This means we need to only recompile the tool to target different architectures. In addition we have made the tool independent of the memory consistency model it is verifying by taking as input to the tool a description of the local ordering rules, as described in definition 1 in a standard format rulefile. This allows us to verify different consistency models (Itanium and different generations of IA-32) and adapt to changes in the consistency models that may happen in the future.

Debug Support: A critical requirement in CPU verification is that failures should be root caused to bugs as soon as possible. Ease of debugging failures is very important in all of Intel's verification methodologies. A failure in our case is a cycle in the graph. The problem with our algorithm formulation is that the final cycle is detected only in terms of *which* nodes are participating in the cycle. There is no way to determine from the closed form adjacency matrix what is the *ordering* of nodes in the cycle. Also the nature of the basic algorithm often leads to more than one cycle in long tests. To work around this problem without sacrificing algorithm efficiency we use a backtracking algorithm described in Figure 3 that prints all the detected cycles. The only change we need to make to the algorithm described in section 3.3 is that it takes as parameter an edge e. Whenever the $AddEdge$ function adds the edge e during execution of the algorithm we return true indicating that this edge is actually added by one of the rules in the algorithm. We also return the reason for addition of this edge which allows all edges to be labelled with the corresponding rule, a good aid to debug. Note that the backtrack though costly is only run in case of failure which should be rare.

6   Performance and Scaling

We include some performance data to support our claims of efficient algorithm design. In figure 4(a) we show how the cost of running the algorithm grows with increasing number of nodes. Clearly the algorithm scales well. In figure 4(b) we show how the

speedup increases when we use more processors to run the algorithm while keeping the problem size (number of graph nodes) same. The near to linear speedup (ideal) indicates that the parallelization decisions have been correctly made and load balance the problem well among different processors. All the presented scalability data was taken on an 8 way 1.2 Ghz Intel®Xeon®[5] processor platform running Linux.
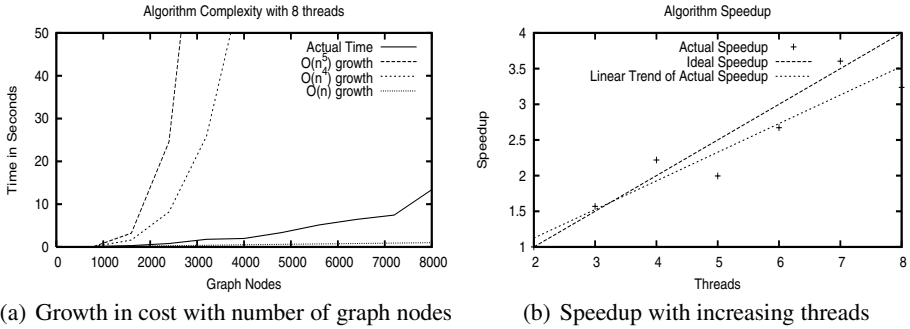

(a) Growth in cost with number of graph nodes   (b) Speedup with increasing threads

Fig. 4. Algorithm Performance

7   Limitations

Although our algorithm is general enough to cover the memory consistency models we need to check for at Intel, it has certain limitations and assumptions stated here. We assume that all stores in the test to the same location write unique values. Thus we are never in a position where we need to reconcile a load with multiple stores for rule 2. This does not affect our coverage of the logic that is responsible for maintaining memory ordering, since that logic has no dependency on the actual data values
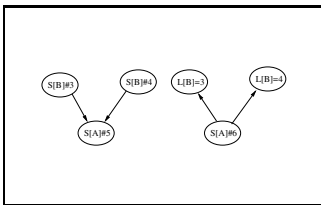

Fig. 5. A missed edge

The algorithm assumes store atomicity, which is necessary for Axiom 3. However it supports slightly relaxed consistency models which allow a load to observe a local store which precedes it in program order, before it is globally observed. Thus we cover all coherence protocols that support the notion of relaxed write atomicity which can be defined as : No store is visible to any *other* processor *before* the execution point of the store. Based on our discussion with Intel microarchitects we determined that all IA-32 and *current* generations of Itanium microprocessors support this due to identifiable and atomic global observation points for any store. This is mostly due to the shared bus and single chipset. For Itanium we can still adapt to the case where stores are not atomically observed by other processors by checking only store releases

---

[5] Intel®Xeon®is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

[12]. Another approach is to split stores into one for each observing processor and appropriately modify rule 2. This would lead to a worse case degradation of checking performance by a factor equal to the number of processors.

Last, the algorithm does approximate checking only (since it is a polynomial time solution to an NP-Hard problem). It does not completely check for Axiom 3, since it does not attempt to order all stores and thereby find additional inferred edges which could lead to a cycle. An example taken from [1] is shown in 5. The algorithm is unable to deduce the ordering from $S[A]\#6$ to $S[A]\#5$ although that is the only possibility given that the loads to location B read different values. Adding a similar mirrored set of nodes, 2 stores to location C before $S[A]\#6$ and two loads from location C after $S[A]\#5$ give an example violation of the TSO model which is missed by this algorithm. However, we hypothesize that only a small fraction of bugs actually lead to such cases and these are ultimately found by sufficient random testing which will show them up in a form the algorithm can detect, another reason why we place so much emphasis on test tool performance.

8 Conclusion

We have described an algorithm that does efficient polynomial time memory consistency verification. Our algorithm meets its stated goals of efficiency and generality. It is implemented in a tool that is used across multiple groups in Intel to verify increasingly complex microprocessors. It has been appreciated across the corporation for finding a number of bugs that are otherwise hard to find and point to extremely subtle flaws in implementing the memory consistency model. We hope to work further in decreasing the cost of the algorithm by by studying the nature of the graphs generated and considering more fine grained parallelization opportunities.

References

1. Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, and Juin-Yeu Joseph Lu. Tsotool: A program for verifying memory systems using the memory consistency model. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, page 114, Washington, DC, USA, 2004. IEEE Computer Society.
2. Phillip B. Gibbons and Ephraim Korach. The complexity of sequential consistency. In *SPDP:Proceedings of the Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 317–325, 1992.
3. Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence. In *SPAA '03: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 254–255, New York, NY, USA, 2003. ACM Press.
4. Cantin, J. F., Lipasti, M. H., and Smith, J. E. "The Complexity of Verifying Memory Coherence". In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, pages 254 – 255, San Diego, 2003.

5. Chaiyasit Manovit and Sudheendra Hangal. Efficient algorithms for verifying memory consistency. In *SPAA'05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, pages 245–252, New York, NY, USA, 2005. ACM Press.

6. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

7. Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

8. *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, 2005. URL: `http://www.intel.com/design/pentium4/manuals/index_new.htm`.

9. *Intel Itanium Architecture Volume 1:Application Architecture*. Intel Corporation, 2005. URL: `http://www.intel.com/design/itanium/manuals/iiasdmanual.htm`.

10. Harold W. Cain, Mikko H. Lipasti, and Ravi Nair. Constraint graph analysis of multithreaded programs. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 4, Washington, DC, USA, 2003. IEEE Computer Society.

11. Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj. Qb or not qb: An efficient execution verification tool for memory orderings. In *CAV*, pages 401–413, 2004.

12. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. Intel Corporation, 2005. URL: `http://www.intel.com/design/itanium/downloads/251429.htm`.

13. Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

14. Charles J. Fleckenstein John C. Huang Amitabha Roy, Stephan Zeisset. Fast and Generalized Polynomial Time Memory Consistency Verification. Technical Report arXiv:cs.AR/0605039, May 2006.

15. Utpal K. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Norwell, MA, USA, 1994.

16. Bob Bentley. Validating the intel pentium 4 microprocessor. In *DAC '01: Proceedings of the 38th conference on Design automation*, pages 244–248, New York, NY, USA, 2001. ACM Press.