

QB or not QB: An Efficient Execution Verification Tool for Memory Orderings^{*}

Ganesh Gopalakrishnan, Yue Yang, and Hemanthkumar Sivaraj

School of Computing, University of Utah
{ganesh,yyang,hemanth}@cs.utah.edu

Abstract. We study the problem of formally verifying shared memory multiprocessor executions against memory consistency models—an important step during post-silicon verification of multiprocessor machines. We employ our previously reported style of writing formal specifications for shared memory models in higher order logic (HOL), obtaining intuitive as well as modular specifications. Our specification consists of a conjunction of rules that constrain the *global visibility order*. Given an execution to be checked, our algorithm generates Boolean constraints that capture the conditions under which the execution is legal under the visibility order. We initially took the approach of specializing the memory model HOL axioms into equivalent (for the execution to be checked) quantified boolean formulae (QBF). As this technique proved inefficient, we took the alternative approach of converting the HOL axioms into a program that generates a SAT instance when run on an execution. In effect, the quantifications in our memory model specification were realized as iterations in the program. The generated Boolean constraints are satisfiable if and only if the given execution is legal under the memory model. We evaluate two different approaches to encode the Boolean constraints, and also incremental techniques to generate and solve Boolean constraints. Key results include a demonstration that we can handle executions of realistic lengths for the modern Intel Itanium memory model. Further research into proper selection of Boolean encodings, incremental SAT checking, efficient handling of transitivity, and the generation of unsatisfiable cores for locating errors are expected to make our technique practical.

1 Introduction

In many areas of computer design, formal verification has virtually eliminated logical bugs escaping into detailed designs (including silicon). However, in areas where the system complexity is high, and global interactions among large collections of subsystems govern the overall behavior, formal verification cannot yet cope with the complex models involved. The verification of multiprocessor machines for conformance to shared memory consistency models [1] is one such area. This paper focuses on verifying whether multiprocessor executions violate

^{*} Supported by NSF Grant CCR-0081406 and SRC Contract 1031.001

memory ordering rules. These executions may be obtained from multiprocessor simulators or from real machines. The current practice is to employ well-chosen test programs to obtain a collection of “interesting” executions from machines and simulators. These executions are then examined using *ad hoc* “checker” programs. Our contribution is to make the second step formal.

It is crucially important that multiprocessor machines and simulators conform to their memory models. Future high performance operating systems will exploit relaxed memory orderings to enhance performance; they will fail if the multiprocessor deviates from its memory model. However, as far as we know from the literature, none of the existing methods can *verify* executions against formal descriptions of industrial memory models. A tool such as what we propose can also help designers comprehend a given memory model by executing critical code fragments. Given that industrial memory models are extremely complex, an efficient execution verification facility is very important in practice.

In this paper, we show that Boolean satisfiability (SAT) based tools can be developed for verifying executions of realistic lengths. Our current work is aimed at the Intel Itanium memory model [2]; the technique is, however, general. Given a shared memory multiprocessor execution, e , and a formal specification of the memory model as logical axioms, r , we offer a formal technique to verify whether e is allowed under r . By the term *execution*, we mean multiprocessor assembly programs over *loads*, *stores*, *fences*, and other memory operations, with the *loads* annotated with returned values. The actual assembly program run on a machine may consist of instructions other than *loads* and *stores*; it may, for instance, include branches and arithmetic operations. In those cases, e retains a dynamic trace of just the *load* and *store* group of instructions, with the *loads* annotated with their returned values. In this paper, we will depict e in the form of assembly programs consisting of only *load* and *store* instructions, with the *loads* annotated with the returned values (such annotated programs are called “litmus tests”). We do not discuss here how such dynamic traces can be obtained.

Gibbons and Korach [3] have shown that the problem of checking executions against sequential consistency [4] is NP-complete. Generalizing this result, Cantin [5] has shown that the problem of checking executions against memory ordering rules that contain coherence as a sub-rule is NP-hard. Since the Itanium memory model contains coherence, and since executions serve as polynomial certificates that can be checked against the rules in polynomial time, we have an NP-complete problem at hand. Despite these results, we initially found it natural to employ a quantified boolean formula (QBF) [6] satisfiability checker. This is because of two facts: (i) the Itanium memory model is quite complex, and to write a formal specification in a declarative and intuitive style, we employed higher order logic (HOL) [7–9]; (ii) since we wanted to have a trustworthy checking algorithm, we took the approach of *specializing* the HOL description to a QBF description so as to check e . This specialization is natural, given that r captures the memory model in terms of quantifiers that range over program counters, addresses, and data values that have arbitrary ranges, whereas e has these quantities occurring in it over a finite range. However, the direct use of a

```

P0: st a,1;          ld r1,a <1>; st b,r1 <1>;
P1: ld.acq r2,b <1>; ld r3,a <0>;

```

```

[{id=0;proc=0;pc=0;op=St;var=0;data=1;wrID=0;
  wrType=Local;wrProc=0;reg=-1;useReg=false};
{id=1;proc=0;pc=0;op=St;var=0;data=1;wrID=0;
  wrType=Remote;wrProc=0;reg=-1;useReg=false};
{id=2;proc=0;pc=0;op=St;var=0;data=1;wrID=0;
  wrType=Remote;wrProc=1;reg=-1;useReg=false};
{id=3;proc=0;pc=1;op=Ld;var=0;data=1;wrID=-1;
  wrType=-1;wrProc=-1;reg=0;useReg=true};
{id=4;proc=0;pc=2;op=St;var=1;data=1;wrID=4;
  wrType=Local;wrProc=0;reg=0;useReg=true};
{id=5;proc=0;pc=2;op=St;var=1;data=1;wrID=4;
  wrType=Remote;wrProc=0;reg=0;useReg=true};
{id=6;proc=0;pc=2;op=St;var=1;data=1;wrID=4;
  wrType=Remote;wrProc=1;reg=0;useReg=true};
{id=7;proc=1;pc=0;op=LdAcq;var=1;data=1;wrID=-1;
  wrType=-1;wrProc=-1;reg=1;useReg=true};
{id=8;proc=1;pc=1;op=Ld;var=0;data=0;wrID=-1;
  wrType=-1;wrProc=-1;reg=2;useReg=true} ]

```

Fig. 1. The execution of a multiprocessor assembly program, and the tuples it generates

QBF solver[10] proved to be of impractical complexity. Therefore, we pursue the following alternative approach. We first derive a mostly¹ applicative functional program p from r . Program p captures the quantifications present in r via iterative loops (tail-recursive calls). It also stages the evaluation of the conditionals in an efficient manner. Such a program p , when run on execution e , evaluates all the *ground* constraints (constraints without free variables) efficiently by direct execution, and generates non-ground constraints in the form of a SAT instance b which is satisfiable if and only if e is allowed under r . Further we demonstrate that the derivation of p can be automated in a straightforward manner.

Related Work: Park et.al. [11] wrote an operational description of the Sparc V9 [12] memory models in Murphi [13] and used it to check assembly language executions. It is our experience is that this approach does not scale beyond a dozen or so instructions; it is also our experience that specifications for memory models as intricate as the Itanium are very difficult to develop using the operational approach [14]. Since our HOL specification follows the axiomatic style used in Intel’s description [2], it can be more easily trusted. It also can be formally examined using theorem provers to enhance our confidence in it. In [15], we show that a whole range of memory models can be described in the same HOL style as we use here.

¹ The only imperative operations are file I/O.

Yu [16] captured memory ordering rules for the Alpha microprocessor [12] as first-order axioms. Given an execution to be checked, they generated verification conditions for the decision procedure Simplify [17]. We believe that the use of SAT for this application will scale better.

In previous work [18], we presented the higher order logic specification of the Itanium memory model and its realization as a *constraint logic program*. We also sketched an approach to generate Boolean satisfiability constraints. Three major problems remained: (i) the constraint logic program version was unable to handle more than about 20 tuples (a dozen or so instructions); (ii) the SAT version was extremely difficult to debug owing to it being retrofitted into a logic program; (iii) since the logic program did not exploit the nature of the higher order logic axioms, it took far more time to generate SAT instances than to solve them—often with a ratio of 200:1. The present work is an improvement in all these regards and also offers several new directions. In particular, it offers a reliable formal derivation scheme to obtain the SAT instance generation program. This program can handle much longer executions—about 300 tuples. The SAT instances generated from such executions can be solved using SAT tools in reasonable time, thanks to the care exercised in selecting the Boolean encoding method. We have also identified many avenues to scale the capacity of our tool further.

2 Overview of Our Approach

As a simple example, consider the litmus test shown in Figure 1. Processor P0 issues a store (`st`) instruction to location `a` with data value `1`. It then issues a load (`ld`) to location `a`, which fetches the value `1` into register `r1` (shown via the annotation `<1>`). It then stores the contents of register `r1` into location `b` (we show the value annotation `<1>` here also, as we can compute the value in `r1` at this point). Processor P1 issues a *load acquire* (`ld.acq`) instruction to begin with. This fetches value `1` from location `b` into register `r2`. It then performs an ordinary `ld` instruction, obtaining `0` from location `a` that is stored into `r3`. The only strongly ordered operation in this whole program is `ld.acq`. Itanium rules require that the visibility of `ld.acq` must be before the visibility of all the instructions following it in program order (i.e., `ld.acq` acts as a “one-way barrier”). The question now is: “is this execution legal?”

Modeling Executions Using Tuples: Following earlier approaches [2, 19], we employ a set of tuples to model executions. One tuple is employed to capture the attributes of each `ld` instruction, and $p + 1$ tuples are employed to model the attributes of each `st` or `st.rel` instruction, where there are p processors in the multiprocessor (Figure 1). In our example, each store generates three tuples, giving us a total of nine tuples². Of the $p + 1$ tuples modeling a store (`st`), one is a *local store* and the remaining p are global stores, one for each processor.

² We have written an assembler to generate tuples from annotated assembly programs.

$$\begin{aligned}
\text{legal}(ops) &= \\
&\exists order. \\
\text{StrictTotalOrder} & ops\ order \wedge \text{WriteOperationOrder} \quad ops\ order \wedge \\
\text{ItProgramOrder} & ops\ order \wedge \text{MemoryDataDependence} \quad ops\ order \wedge \\
\text{DataFlowDependence} & ops\ order \wedge \text{Coherence} \quad ops\ order \wedge \\
\text{ReadValue} & ops\ order \wedge \text{AtomicWBRelease} \quad ops\ order \wedge \\
\text{SequentialUC} & ops\ order \wedge \text{NoUCBypass} \quad ops\ order \\
\text{StrictTotalOrder } ops\ order &= \text{IrreflexiveOrder } ops\ order \\
&\wedge \text{TransitiveOrder } ops\ order \\
&\wedge \text{TotallyOrdered } ops\ order \\
\text{Irreflexive } ops\ order &= \forall(i \in ops). \forall(j \in ops). (i.\text{id} = j.\text{id}) \Rightarrow \neg order\ i\ j \\
\text{Transitive } ops\ order &= \forall(i \in ops). \forall(j \in ops). \forall(k \in ops). \\
&\quad (order\ i\ j \wedge order\ j\ k \Rightarrow order\ i\ k) \\
\text{TotallyOrdered } ops\ order &= \forall(i \in ops). \forall(j \in ops). (i \in ops) \wedge (j \in ops). \wedge \neg(i.\text{id} = j.\text{id}) \\
&\quad \Rightarrow order\ i\ j \vee order\ j\ i \\
\text{ItProgramOrder } ops\ order &= \forall(i \in ops). \forall(j \in ops). \\
&\quad \text{ordByAcquire } i\ j \vee \text{ordByRelease } i\ j \vee \text{ordByFence } i\ j \\
&\quad \Rightarrow order\ i\ j \\
\text{ordByAcquire } i\ j &= \text{ordByProgram } i\ j \wedge (i.\text{op} = \text{LdAcq}) \\
\text{ordByProgram } i\ j &= (i.\text{proc} = j.\text{proc}) \wedge i.\text{pc} < j.\text{pc} \\
\text{ReadValue } ops\ order &= \forall(j \in ops). \\
&\quad (\text{isRd } j \Rightarrow \\
&\quad \quad \text{validLocalWr } ops\ order\ j \\
&\quad \quad \vee \text{validRemoteWr } ops\ order\ j \\
&\quad \quad \vee \text{validDefaultWr } ops\ order\ j) \\
&\quad \wedge (\text{isWr } j \wedge j.\text{useReg} \Rightarrow \text{validRd } ops\ order\ j) \\
\text{validRd } ops\ order\ j &= \\
&\quad \exists(i \in ops). \text{isRd } i \wedge (i.\text{reg} = j.\text{reg}) \wedge \text{ordByProgram } i\ j \\
&\quad \wedge (i.\text{data} = j.\text{data}) \\
&\quad \wedge \neg(\exists(k \in ops). \text{isRd } k \wedge (k.\text{reg} = j.\text{reg}) \\
&\quad \quad \wedge \text{ordByProgram } i\ k \wedge \text{ordByProgram } k\ j) \\
\text{atomicWBRelease } ops\ order &= \\
&\quad \forall(i \in ops). \forall(j \in ops). \forall(k \in ops). \\
&\quad (i.\text{op} = \text{StRel}) \wedge (i.\text{wrType} = \text{Remote}) \\
&\quad \wedge (k.\text{op} = \text{StRel}) \wedge (k.\text{wrType} = \text{Remote}) \\
&\quad \wedge (i.\text{wrID} = k.\text{wrID}) \wedge (\text{attr_of } i.\text{var} = \text{WB}) \\
&\quad \wedge order\ i\ j \wedge order\ j\ k \\
&\quad \Rightarrow (j.\text{op} = \text{StRel}) \wedge (j.\text{wrType} = \text{Remote}) \\
&\quad \quad \wedge (j.\text{wrID} = i.\text{wrID})
\end{aligned}$$

Fig. 2. Excerpts from the Itanium Ordering Rules (For the full spec, see [18])

		t0	t1	t2	t3
t0	$[ord_{01}, ord_{00}]$	t0	0	1	
t1	$[ord_{11}, ord_{10}]$	t1	0		
t2	$[ord_{21}, ord_{20}]$	t2		0	
t3	$[ord_{31}, ord_{30}]$	t3			0

Fig. 3. Illustration of the $nlogn$ (left) and nn (right) methods

For example, consider the tuples with `id=0`, `id=1`, and `id=2`. These are tuples coming from the store instruction of P0 (`proc=0`), have program counter `pc=0`, employ variable `var=0`, and have `data=1`. The `wrID=0` says that these store tuples come from the store instruction with `id=0`. To distinguish where these stores are observed, we employ the `wrProc` attribute, the values of which are 0, 0, and 1 respectively. Notice that the tuple with `id=0` has `wrType=Local`, and the one with `id=1` has `wrType=Remote`. (“Remote” means “global” in the parlance of [2]). Notice that we employ two tuples, namely the ones with `id=0` and `id=1`, both for the local processor `proc=0` (P0). This is to facilitate modeling the semantics of *load bypassing*—the ability of a processor to read its own store early. For details, please see [2, 18].

The modeling details associated with *load* instructions are much simpler. We simply employ one tuple per `ld` or `ld.acq` instruction. The `useReg` field captures whether a register is involved, and the `reg` field indicates which register is involved. All fields with `-1` are don’t-cares.

Overview of the Itanium Ordering Rules: Figure 2 provides roughly a fourth of the Itanium ordering rules from our full specification. The legality of an execution is checked by `legal ops`, where `ops` is the collection of tuples obtained from an execution, such as in Figure 1. Note how `order`, a binary relation, is passed around and constrained by all the ordering rules. Basically, the definition consists of four distinct parts: (i) **StrictTotalOrder**, which seeks one arrangement of the tuples into a strict total order, (ii) **ReadValue**, which checks that all reads in this strict total order either return the value associated with the most recent (in the strict total order) write to the same location, or the initial store values, if there is no write to that location, (iii) **ItProgramOrder**, which is weakened program order that orders instructions only if one of them is an *acquire*, a *release* or a *fence*, and (iv) all the remaining rules which try to recover some modicum of program order. For instance, an instruction `i` is ordered before an instruction `j` if `i` is of type `ld.acq`, as captured by the **ordByAcquire** rule.

This style of specification, adopted by [2], makes it easier to contrast it with sequential consistency. For instance, if we change **ItProgramOrder** into a regular program order relation, and retain **ReadValue** and **StrictTotalOrder**, we obtain sequential consistency. Since the combination of **ItProgramOrder** and the rules mentioned in (iv) above is weaker than the regular program order relation, the Itanium memory model allows *more* solutions under **StrictTotalOrder** than with regular sequential consistency. Hence the Itanium memory model is weaker than sequential consistency. However, the variety of instructions allowed under Itanium is more than just *load* and *store*. Hence, we can only hope to make qualitative comparisons between these models.

Overview of Boolean Encoding: As far as the relation `legal` goes, `ops` of Figure 1 is to be viewed as a set of tuples. Notice that **StrictTotalOrder** seeks

to arrange the elements of *ops* into a strict total order such that the remaining constraints are met (the arrangement of the elements of *ops* is captured in the *order* relation). Total ordering among n tuples can be encoded using auxiliary Boolean variables in two obvious ways (Figure 3, also see [20] where these are called the *small domain* and the e_{ij} approaches): (i) the *nlogn* approach, in which a bit-vector of $\log(n)$ Boolean variables of the form $[ord_{i,j-1} \dots ord_{i,0}]$ are augmented to the i th tuple (example tuples are shown as **t0** through **t3** in the figure). Here, n is the number of tuples, assumed to be a power of 2, and $j = \log_2(n)$; (ii) the *nn* approach, in which n^2 Boolean variables (denoted by $matrix_{ij}$, with $0 \leq i, j < n$) are introduced to represent how tuples are ordered. In the *nlogn* approach, **StrictTotalOrder** is implemented by the constraint $[ord_{i,\log_2(n)-1}, \dots, ord_{i,0}] \neq [ord_{j,\log_2(n)-1}, \dots, ord_{j,0}]$ for all $i \neq j$. In the *nn* approach, **StrictTotalOrder** is implemented via its constituents: **irreflexive**, **transitive**, and **totallyOrdered**. Constraint **irreflexive** is encoded by setting the diagonal elements of the matrix to 0. Constraint **transitive** is encoded by generating the formula $(matrix_{ij} \wedge matrix_{jk}) \Rightarrow matrix_{ik}$. Constraint **totallyOrdered** is encoded by generating the formula $matrix_{ij} \vee matrix_{ji}$.

The size of the formula which encodes **StrictTotalOrder** for the *nn* method is far greater than for the *nlogn* method. This is largely because of the transitivity axiom where we go through every triple of tuples and generate the transitivity clause. We plan to investigate other methods discussed in [20]. One key difference between our work and that of [20] is that in their setting, a collection of first-order equational formulae (or more generally speaking, formulae in *separation logic* involving $=$, \geq , and $<$) are to be checked for validity. In doing so, transitivity is applied over the given set of equations. In our case, we are *solving* for an *order* over the tuples. The number of these tuples is expected to be far higher. In a sense, our method searches for the few permutations of the given sequence of tuples that are consistent with the memory ordering rules. We hope to investigate lazy approaches to handling transitivity as discussed in Section 5.

A significant advantage of the *nn* method over the *nlogn* method in our context is that it generates much smaller formulae for the rest of the constraints other than transitivity. For example, suppose while processing a memory ordering rule we have to specify that some tuple, say **t0**, appears before another tuple, say **t3**, in any allowed total order. This encoding is achieved by $[ord_{01}ord_{00}] < [ord_{31}ord_{30}]$ in the *nlogn* method, while simply achieved by asserting $matrix_{03}$ in the *nn* method (see Figure 3 for a ‘1’ in the matrix). These trade-offs are studied in Section 4. In effect, we found that despite the use of n^2 variables as opposed to $n \log(n)$, the *nn* method is more efficient during SAT checking. Similar results are obtained in [20] where SAT-checking is often faster under their e_{ij} method (similar to our *nn* method) than their small domain method (similar to our *nlogn* method).

In post-silicon verification, tests on multiprocessor machines are run multiple times in the hope of obtaining different load values due to non-deterministic interleavings. This naturally fits with the use of incremental SAT methods for execution verification.

3 Program Derivation from Memory Ordering Rules

We provide an example of how one rule of Itanium, namely **atomicWBRelease**, is transformed into a program; all other rules are handled similarly. The initial specification is in Figure 2. Recall from Section 1 that for every store instruction, we generate $p + 1$ stores, of which p are considered ‘remote stores.’ Rule **atomicWBRelease** says that all these remote stores form an ‘atomic packet’ in the sense that any other event e is strictly before or strictly after all the events in this packet. Notice how it is specified by the following axiom which says: if j is an event ‘trapped’ between i and k , then j also belongs to the atomic packet of all remote stores. (A note about our notation: we use the generic *order* relation to denote a total order over the set of tuple operations **ops**. When it comes to specifically generating the Boolean constraints, we choose *ord* or *matrix* depending on the encoding method used. This difference shows up in Table 1(e) in part b_2 of the results.)

We now pre-process this specification by applying the contrapositive rule. The general idea is to bring ground constraints to the antecedent so that we can evaluate them through direct execution. The SAT instances can then be generated from the consequent part. The result of this step is a formula with three outermost quantifiers (Figure 4, before *Quantifier Scope Reduction*). If we translate this directly into loops, we will obtain a very inefficient program. The *Quantifier Scope Reduction* step takes advantage of the limited scope of various sub-formulae and rewrites the quantified expression into a series of staged quantifications. This will allow many iterations of outer loops to be cut-off early, thus not suffering from the full brunt of the $O(n^3)$ complexity. This dramatically reduced our SAT-generation time. For example, $i.op = \mathbf{StRel}$ depends only on i , and so the inner loops are not called for all those instructions that do not pass this test.

The last stage of our translation (*SAT-generation program sketch*) obtains a series of tail-recursive functions capturing the semantics of the quantified expression. Here, *foldr* reduces a given list of arguments (generated by *map*) using conjunction; this is because conjunction is the explicitly provided second operation ‘&’ for *foldr*. The list that is reduced is obtained by mapping the function (**fn** $i \rightarrow e(i)$) (a Lambda abstraction) on the given list. Forms such as $f(i)(j)$ are employed as opposed to $f(i, j)$ to signify *currying* [21]. The main difference between the sketch we provide and the actual Ocaml [22] code we employ is that the latter emits constraints on-the-fly to a file instead of building an expression tree using **foldr** as shown in our sketch.

4 Results

Our program handles all the 17 litmus tests given in [2] except a few that involve partial word writes that are currently omitted. These tests ran considerably faster than those in [18].

Next, we considered executions with 32, 64, and 128 tuples in our experiments. The complexity of our algorithm depends primarily on the number of

(a).SAT generation times for $nlogn$ encoding (parts b_1 and b_2).

#tuples	Part b_1			Part b_2		
	time (secs)	#vars	#clauses	time (secs)	#vars	#clauses
32	0.219	20,992	68,448	1.635	92,316	258,632
64	1.213	101,184	330,624	17.178	852,632	2,387,664
128	5.748	472,320	1,544,320	179.026	7,777,200	21,775,520

(b). SAT generation times for nn encoding

#tuples	Part b_1			Part b_2		
	time (secs)	#vars	#clauses	time (secs)	#vars	#clauses
32	0.509	67,552	233,376	0.100	8,044	22,760
64	4.311	532,416	1,851,200	0.967	63,832	179,792
128	34.255	4,226,944	14,745,216	9.095	509,104	1,431,200

(c). ‘Monolithic’ gives the SAT solver execution time for the full SAT instance. Column Part b_1 gives the SAT time for part b_1 . Part b_2 gives the time for SAT after resuming from the checkpoint and adding the new constraints.

#tuples	$nlogn$ encoding			nn encoding		
	monolithic	Part b_1	Part b_2	monolithic	Part b_1	Part b_2
32	9.61	0.6	4.3	0.33	0.69	0.05
64	247.17	29.53	37.6	2.73	6.17	0.5
128	aborted	1341.85	aborted	164.8	145.64	351.1

(d). $nlogn$ encoding: 1-Cl, 2-Cl, and 3-Cl give the percentage of clauses with one, two and three literals.

#tuples	Part 1			Part b_2		
	1-Cl (%)	2-Cl (%)	3-Cl (%)	1-Cl (%)	2-Cl (%)	3-Cl (%)
32	1.449	46.376	52.173	0.064	71.387	28.547
64	1.219	46.341	52.439	0.024	71.419	28.555
128	1.052	46.315	52.631	0.010	71.430	28.559

(e). nn encoding: 1-Cl, 2-Cl, and 3-Cl give the percentage of clauses with one, two and three literals.

#tuples	Part b_1			Part b_2		
	1-Cl (%)	2-Cl (%)	3-Cl (%)	1-Cl (%)	2-Cl (%)	3-Cl (%)
32	14.479	57.013	28.506	0.738	70.685	28.576
64	14.382	57.078	28.539	0.329	71.006	28.664
128	14.333	57.110	28.555	0.154	71.143	28.702

Table 1. Result Tables

Applying contrapositive

```

atomicWBRelease(ops, order) =
  ∀(i ∈ ops). ∀(j ∈ ops). ∀(k ∈ ops).
  (i.op = StRel) ∧ (i.wrType = Remote)
  ∧ (k.op = StRel) ∧ (k.wrType = Remote)
  ∧ (i.wrID = k.wrID) ∧ (attr_of i.var = WB)
  ∧ ¬((j.op = StRel) ∧ (j.wrType = Remote) ∧ (j.wrID = i.wrID))
  ⇒ ¬(order(i, j) ∧ order(j, k))

```

Quantifier Scope Reduction

```

atomicWBRelease(ops, order) =
  ∀(i ∈ ops).
  (i.op = StRel) ∧ (i.wrType = Remote)
  ∧ (attr_of i.var = WB)
  ⇒ ∀(k ∈ ops).
    (k.op = StRel) ∧ (k.wrType = Remote) ∧ (i.wrID = k.wrID)
    ⇒ ∀(j ∈ ops).
      ¬((j.op = StRel)
        ∧ (j.wrType = Remote)
        ∧ (j.wrID = i.wrID))
      ⇒ ¬(order(i, j) ∧ order(j, k))

```

SAT-generation program sketch

```

atomicWBRelease(ops) = forall(i, ops, wb(i));
wb(i) = if(¬((attr_of i.var = WB) & (i.op = StRel) & (i.wrType = Remote)))
  then true else forall(k, ops, wb1(i)(k));
wb1(i)(k) = if(¬((k.op = StRel) & (k.wrType = Remote) & (i.wrID = k.wrID)))
  then true else forall(j, ops, wb2(i)(k)(j));
wb2(i)(k)(j) = if((j.op = StRel) & (j.wrType = Remote) & (j.wrID = i.wrID))
  then true else ¬(order(i, j) & order(j, k));
forall(i, S, e(i)) = foldr(map(fn i → e(i))(S), &, true)

```

Fig. 4. Sketch of SAT-generation Program Derivation

tuples, and far less on the remaining attributes of the tuples. Thus, checking 8 tuples over 2 processors has nearly the same complexity as checking 2 tuples over 8 processors. We selected the instruction mix heavily skewed towards *stores* to reflect a worst-case behavior (more rules pertain to stores than loads). All runs were on an AMD Athlon XP2100+ CPU (1.733 GHz, 1GB memory, Red Hat Linux V.9). We used the Satzoo incremental solver [23].

We evaluated two approaches, one generating and solving the constraints monolithically, and the other using partial evaluation. To motivate the latter approach, note that the constraints generated from **TotallyOrdered** depends on the number of tuples—and not on the contents of the tuples. Capitalizing on this fact, we pre-generated the constraints pertaining to **TotallyOrdered** for various lengths; call these constraints b_{1n} , where n represents the number of

tuples anticipated in a test program to be given in future. We then loaded these constraints into the SAT solver, and created the checkpoint of a runnable image of the SAT solver using the `ckpt` tool [24]³ to obtain *ckpt_n*. Later, when presented with a litmus test of length n , we only generated the *remaining* constraints (other than *TotallyOrdered*) for it; call the resulting constraints b_{2n} . We then ran *ckpt_{1n}* on b_{2n} . Table 1(a) provides the time to generate SAT instances for the *nlogn* encoding method for formula parts b_1 and b_2 . Table 1(b) provides these for the *nn* encoding method. Table 1(c) gives the SAT solving time for the *nn* and *nlogn* methods for a monolithic run, and for running parts b_1 and b_2 separately.

The results show that under the *nn* encoding, it takes longer to generate SAT instances for part b_1 , but considerably shorter for part b_2 . The main reason is that in our implementation, the number of clauses nc grows as $7n^3 + \dots$ and the number of variables as $2n^3 + \dots$ (later code improvements have brought down nc to $n^3 + \dots$). The SAT solving times are uniformly lower for the *nn* method. This is because of the preponderance of clauses with smaller numbers of literals, as shown in Tables 1(d) and 1(e). In particular, part b_2 of *nn* encoding has both lower number of clauses and a higher proportion of clauses with 1 or 2 literals than the *nlogn* encoding. To summarize: (i) The *nn* encoding is better in terms of SAT solving time. The SAT generation time is acceptably small till about 128 tuples. (ii) Verifying in two parts b_1 and b_2 can be advantageous for problems of reasonable sizes. The advantage is far more for the *nn* approach. (iii) Since the same test is re-run multiple times, partial evaluation and other incremental SAT techniques can play a crucial role in overall efficiency.

Recently we have run a more realistic test of 130 assembly language instructions⁴. These expanded into 239 tuples. Initially, since the constraint generation program could not handle the transitivity rule, we suppressed it, obtaining a SAT instance of 115,637 variables and 164,848 clauses. This SAT instance proved to be unsatisfiable. Upon deeper examination using the `Zcore` program (distributed with the latest Zchaff [25]), we discovered an unsatisfiable core of 9 clauses. By analyzing these clauses, it was discovered that the error we detected resulted from us forgetting to initialize the memory state prior to the test. Further experiments with these and other realistic tests are underway, and our latest results will be presented on our webpage [26].

To sum up, proper handling of transitivity is crucial to scale our tool further. Recent code optimizations have allowed us to handle this realistic example *without* suppressing transitivity. However, the complexity of transitivity still lurks—in the 400 tuples and above range as of now. Also, the use of unsatisfiability core generation tools can be of considerable help in finding the root cause of violations.

³ We resorted to binary checkpoints—as opposed to clause checkpoints—because the source code of Satzoo was not available.

⁴ We are deeply indebted to Intel for providing us this test.

5 Concluding Remarks

We proposed a method for verifying shared memory multiprocessor executions where the reference semantics is that of shared memory consistency. We propose a method by which executions can be analyzed using programs that embody the shared memory consistency rules. The ground part of the constraints in these rules are evaluated by the program, and the non-ground parts are emitted as Boolean constraints to check.

Semaphores are currently omitted to retain focus on the overall scalability and usability of our tool. Partial-word writes are also not handled. These extensions are planned for the future. A rudimentary assembler has been written to generate tuples from value-annotated assembly programs. This assembler can model data and address dependencies. The Itanium memory model rules in HOL were hand-translated into a series of tail-recursive programs; this process is best automated to ensure correctness, using the transformation rules illustrated earlier.

If the generated SAT instance is satisfiable, the space of satisfying assignments will reveal the set of allowed executions. Future work will annotate the Boolean constraints (clauses) with the instructions as well as memory ordering rules that generate them. This way, if the SAT instance is unsatisfiable, the unsatisfiability core will reveal which instructions and which memory ordering rules are causing the execution to be invalid. Incremental SAT techniques will be of great importance to develop, as are hierarchical analysis methods that treat groups of instructions atomically.

Better methods for handling transitivity are needed. One approach would be to see if SAT returns a satisfying instance when transitivity is suppressed, and if so to selectively introduce transitivity on those elements corresponding to the SAT instance. The ability to analyze *symbolic* executions (where not all the execution results are ground) would also enhance the usability of our tool.

Acknowledgements: We thank our SRC mentor Kushagra Vaid and his colleagues at Intel for their discussions and comments on our work. Thanks also to Konrad Slind and Gary Lindstrom for their contributions to our research.

References

1. Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.
2. A Formal Specification of Intel(R) Itanium(R) Processor Family Memory Ordering, 2002. <http://www.intel.com/design/itanium/downloads/251429.htm>.
3. Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, August 1997.
4. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

5. Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. 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.
6. Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.
7. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
8. F.K. Hanna and N. Daeche. Specification and verification using higher-order logic. In *7th International Conference on Computer Hardware Description Languages and their Applications*, pages 418–419, 1985.
9. Michael Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In *Formal aspects of VLSI design*, 1986.
10. Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of International Conference on Computer Aided Design*, November 2002.
11. David L. Dill, Seungjoon Park, and Andreas Nowatzky. Formal specification of abstract memory models. In *Research on Integrated Systems*, pages 38–52. MIT Press, 1993.
12. David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9*. P T R Prentice-Hall, 1994.
13. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Computer Aided Verification*, pages 522–525, 1992.
14. Prosenjit Chatterjee and Ganesh Gopalakrishnan. Towards a formal model of shared memory consistency for Intel Itanium. In *ICCD*, pages 515–518, 2001.
15. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *International Parallel and Distributed Processing Symposium*, 2004.
16. Personal Communication with Yuan B. Yu.
17. G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
18. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Analyzing the intel itanium memory ordering rules using logic programming and SAT. In *CHARME*, pages 81–95, 2003. LNCS 2860.
19. Anne Condon, Mark Hill, Manoj Plakal, and David Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Fifth International Symposium On High Performance Computer Architecture (HPCA-5)*, January 1999.
20. Sanjit A. Seshia, Shuvendu K. Lahiri, and Randal E. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Design Automation Conference (DAC)*, pages 425–430, 2003.
21. Michael Gordon. *Programming Language Theory and Implementation*. Prentice-Hall, 1993.
22. www.ocaml.org.
23. Satzoo Incremental SAT Solver. Author: Niklas Een. Also competed in SAT’03. <http://www.math.chalmers.se/~een/Satzoo/An.Extensible.SATsolver.ps.gz>.
24. <http://www.cs.wisc.edu/~zandy/ckpt/>.
25. Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36, 2002. LNCS 2402.
26. http://www.cs.utah.edu/formal_verification.