Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming and SAT *

Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind

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

Abstract. We present a non-operational approach to specifying and analyzing shared memory consistency models. The method uses higher order logic to capture a complete set of ordering constraints on execution traces, in an axiomatic style. A direct translation of the semantics to a constraint logic programming language provides an interactive and incremental framework for exercising and verifying finite test programs. The framework has also been adapted to generate equivalent boolean satisfiability (SAT) problems. These techniques make a memory model specification executable, a powerful feature lacked in most non-operational methods. As an example, we provide a concise formalization of the Intel Itanium memory model and show how constraint solving and SAT solving can be effectively applied for computer aided analysis. Encouraging initial results demonstrate the scalability for complex industrial designs.

1 Introduction

Modern shared memory architectures rely on a rich set of memory-access related instructions to provide the flexibility needed by software. For instance, the Intel Itanium^{TM} processor family [1] provides two varieties of loads and stores in addition to fence and semaphore instructions, each associated with different ordering restrictions. A memory model defines the underlying memory ordering semantics (also known as memory consistency). Proper understanding of these ordering rules is essential for the correctness of shared memory consistency protocols that are aggressive in their ordering permissiveness, as well as for compiler transformations that rearrange multithreaded programs for higher concurrency and minimal synchronization. Due to the complexity of advanced computer architectures, however, practicing designers face a serious problem in reliably comprehending the memory model specification.

Consider, for example, the assembly code shown in Fig. 1 that is run concurrently on two Itanium processors (such code fragments are generally known as *litmus tests*): The first processor, `P1`, executes a *store* of datum `1` into address `a`; it then performs a *store-release*[1] of datum `1` into address `b`. Processor `P2` performs a *load-acquire* from `b`, loading the result into register `r1`. It is followed by an *ordinary load* from location `a` into register `r2`. The question arises: if all locations

* This work was supported by a grant from the Semiconductor Research Corporation for Task 1031.001, and Research Grants CCR-0081406 and CCR-0219805 of NSF.

```
P1                 P2

st      a,1;       ld.acq  r1,b;
st.rel  b,1;       ld      r2,a;
```

Fig. 1. A litmus test showing the ordering properties of store-release and load-acquire. Initially, a = b = 0. Can it result in r1 = 1 and r2 = 0? The Itanium memory model does not permit this result. However, if the load-acquire in P2 is changed to an ordinary load, the result would be allowed.

initially contain `0`, can the final register values be `r1=1` and `r2=0`? To determine the answer, the Itanium memory model must be consulted. The formal specification of the Itanium memory model is given in an Intel application note [2]. It comprises a complex set of ordering rules, 24 of which are expressed explicitly based on a large amount of special terminology. One can follow a pencil-and-pen approach to reason that the execution shown in Fig. 1 is not permitted by the rules specified in [2]. Based on this, one can conclude that even though the instructions in `P2` pertain to different addresses, the underlying hardware is not allowed to carry out the ordinary load at the beginning, and by the same token, a shared memory consistency protocol or an optimizing compiler cannot reorder the instructions in `P2`. A further investigation shows that the above result would be permitted if the `st.rel` in `P1` is changed to a `st`, or the `ld.acq` in `P2` is changed to a `ld`. Therefore, `st.rel` and `ld.acq` must both be used in pairs to achieve the "barrier" effect in this scenario.

A litmus test like this can reveal crucial information to help system designers make right decisions in code selection and optimizations. But as bigger tests are used and more intricate rules are involved, trace properties quickly become non-intuitive and hand-proving program compliance can be very difficult. How can one be assured that there does not exist an interacting rule that might introduce unexpected implications? Also, a large scale design is often composed from simpler components. To avoid being overwhelmed by the overall complexity, a useful technique is to isolate the rules related to a specific architectural feature so that one can analyze the model piece by piece. For example, if one can selectively enable/disable certain rules, he or she may quickly find out that the "program order" rules in [2] are critical to the scenario in Fig. 1 while many others are irrelevant.

These issues suggest that a series of useful features are needed from the specification framework to help people better understand the underlying model. Unfortunately, most non-operational specification methods leave these issues unresolved because they use notations that do not support analysis through execu-

[1] Briefly, a store-release instruction will, at its completion, ensure that all previous instructions are completed; a load-acquire instruction correspondingly ensures that all following instructions will complete only after it completes. These explanations are far from precise - what does "previous" and "completion" mean? A formal specification of a memory model is key to precisely capture these and all similar notions.

tion. Given that designers need lucid and reliable memory model specifications, and given that memory model specifications can live for decades, it is crucial that progress be made in this regard.

In this paper, we take a fresh look at the non-operational specification method and explore what verification techniques can be applied. We make the following contributions in this paper. First, we present a compositional method to axiomatically capture all aspects of the memory ordering requirements, resulting a comprehensive, constraint-based memory consistency model. Second, we propose a method to encode these specifications using FD-Prolog.[2] This enables one to perform interactive and incremental analysis. Third, we have harnessed a boolean satisfiability checker to solve the constraints. To the best of our knowledge, this is the first application of SAT methods for analyzing memory model compliance. As a case study in this approach, we have formalized a large subset of the Itanium memory model and used constraint programming and boolean satisfiability for program analysis.

Related work The area of memory model specification has been pursued under different approaches. Some researchers have employed *operational* style specifications [3] [4] [5] [6], in which the update of a global state is defined step-by-step with the execution of each instruction. For example, an operational model [4] for Sparc V9 [7] was developed in Murphi. With the model checking capability supported by Murphi, this executable model was used to examine many code sequences from the Sparc V9 architecture book. While the descriptions comprising an operational specification often mirror the decision process of an implementer and can be exploited by a model checker, they are not declarative. Hence they tend to emphasize the *how* aspects through their usage of specific data structures, not the *what* aspects that formal specifications are supposed to emphasize.

Other researchers have used *non-operational* (also known as *axiomatic*) specifications, in which the desired properties are directly defined. Non-operational styles have been widely used to describe conceptual memory models [8] [9]. One noticeable limitation of these specifications is the lack of a means for automatic execution. An axiomatic specification of the Alpha memory model was written by Yu [10] in Lisp in 1995. Litmus-tests were written in an S-expression syntax. Verification conditions were generated for the litmus tests and fed to the Simplify [11] verifier of Compaq/SRC. In contrast, we specify the modern Itanium memory model. Our specification is much closer to the actual industrial specification, thanks to the declarative nature of FD-Prolog. The FD constraint solver offers a more interactive and incremental environment. We have also applied SAT and demonstrated its effectiveness.

Lamport and colleagues have specified the Alpha and Itanium memory models in TLA+ [12] [13]. These specifications also support the execution of litmus tests. Their approach builds visibility order inductively. While this also precisely specifies the visibility order, the manner in which such inductive definitions

[2] FD-Prolog refers to Prolog with a finite domain (FD) constraint solver. For example, SICStus Prolog and GNU Prolog have this feature.

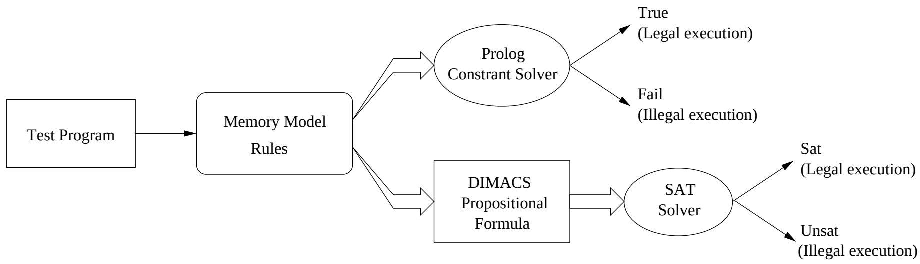

Fig. 2. The process of making an axiomatic memory model executable. Legality of a litmus test can be checked by either a constraint solver or a SAT solver.

are constructed will vary from memory model to memory model, making comparisons among them harder. Our method instead relies on primitive relations and directly describes the components to make up a full memory model. This makes our specifications easier to understand, and more importantly, to compare against other memory models using the same primitives. This also means we can disable some sub-rules quite reliably without affecting the other primitive ordering rules - a danger in a style which merges all the ordering concerns in a monolithic manner.

Roadmap In the next section, we introduce our methodology. Section 3 describes the Itanium memory ordering rules. Section 4 demonstrates the analysis of the Itanium memory model through execution. We conclude and propose future works in Section 5. The concise specification of the Itanium ordering constraints is provided in the Appendix.

2 Overview of the Framework

A pictorial representation of our methodology is shown in Fig. 2. We use a collection of primitive ordering rules, each serving a clear purpose, to specify even the most challenging commercial memory models. This approach mirrors the style adopted in modern declarative specifications written by the industry, such as [2]. Moreover, by using pure logic programs supported by certain modern flavors of Prolog that also include finite domain constraints, one can directly capture these higher order logic specifications and also interactively execute the specifications to obtain execution results for litmus tests. Alternatively, we can obtain SAT instances of the boolean constraints representing the memory model through symbolic execution, in which case boolean satisfiability tools can be employed to quickly answer whether certain litmus tests are legal or not.

2.1 Specification Method

To define a memory model, we use predicate calculus to specify all constraints imposed on an ordering relation *order*. The constraints are almost completely first-order; however, since *order* is a parameter to the specification, the constraints are most easily captured with higher order predicate calculus (we use the HOL logic [14]). Previous non-operational specifications often *implicitly* require general ordering properties, such as totality, transitivity, and circuit-freeness. This is the main reason why such specifications cannot readily be executed. In contrast, we are fully explicit about such properties, and so our constraints completely characterize the memory model.

2.2 Executing Axiomatic Specifications

A straightforward transcription of the formal predicate calculus specification into a Prolog-style logic program makes it possible for interactive and incremental execution of litmus tests. This encourages exploration and experiment in the validation and (we anticipate) the development of complex coherence protocols. To make a specification executable, we instantiate it over a finite execution and convert the verification problem to a satisfiability problem.

The Algorithm Given a finite execution *ops* with n operations, there are n^2 ordering pairs, constituting an ordering matrix $\mathcal{M}$, where the element $\mathcal{M}_{ij}$ indicates whether operations i and j should be ordered. We go through each ordering rule in the specification and impose the corresponding constraint regarding the elements of $\mathcal{M}$. Then we check the satisfiability of all the ordering requirements. If such a $\mathcal{M}$ exists, the trace *ops* is legal, and a valid interleaving can be derived from $\mathcal{M}$. Otherwise, *ops* is not a legal trace.

Applying Constraint Logic Programming Logic programming differs fundamentally from conventional programming in that it describes the logical structure of the problems rather than prescribing the detailed steps of solving them. This naturally reflects the philosophy of the axiomatic specification style. As a result, our formal specification can be easily encoded using Prolog. Memory ordering constraints can be solved through a conjunction of two mechanisms that FD-Prolog readily provides. One applies backtracking search for all constraints expressed by logical variables, and the other uses non-backtracking constraint solving based on *arc consistency* [15] for FD variables, which is potentially more efficient and certainly more complete (especially under the presence of negation) than with logical variables. This works by adding constraints in a monotonically increasing manner to a constraint store, with the in-built constraint propagation rules of FD-Prolog helping refine the variable ranges (or concluding that the constraints are not satisfiable) when constraints are discovered and asserted to the constraint store.

Applying Boolean Satisfiability Techniques The goal of a boolean satisfiability problem is to determine a satisfying variable assignment for a boolean formula or to conclude that no such assignment exists. A slight variant of the Prolog code can let us benefit from SAT solving techniques, which have advanced tremendously in recent years. Instead of solving constraints using a FD solver, we can let Prolog emit SAT instances through symbolic execution. The resultant formula is true if and only if the litmus test is legal under the memory model. It is then sent to a SAT solver to find out the result.

3 Specifying the Itanium Memory Consistency Model

The original Itanium memory ordering specification is informally given in various places in the Itanium architecture manual [1]. Intel later provided an application note [2] to guide system developers. This document uses a combination of English and informal mathematics to specify a core subset of memory operations in a non-operational style. We demonstrate how the specification of [2] can be adapted to our framework to enable computer aided analysis. Virtually the entire Intel application note has been captured.[3] We assume proper address alignment and common address size for all memory accesses, which would be the common case encountered by programmers (even these restrictions could be easily lifted). The detailed definition of the Itanium memory model is presented in the Appendix. This section explains each of the rules. The following definitions are used throughout this paper:

Instructions - Instructions with memory access or memory ordering semantics. Five instruction types are defined in this paper: load-acquire (`ld.acq`), store-release (`st.rel`), unordered load (`ld`), unordered store (`st`), and memory fence (`mf`). An instruction i may have *read semantics* ($\mathbf{isRd}\ i = true$) or *write semantics* ($\mathbf{isWr}\ i = true$). `Ld.acq` and `ld` have read semantics. `St.rel` and `st` have write semantics. `Mf` has neither read nor write semantics. Instructions are decomposed into *operations* to allow a finer specification of the ordering properties.

Execution - Also known as a *trace*, contains all memory operations generated by a program, with stores being annotated with the write data and loads being annotated with the return data. An execution is *legal* if there exists an order among the operations in the execution that satisfies all memory model constraints.

Address Attributes - Every memory location is associated with an address attribute, which can be write-back (WB), uncacheable (UC), or write-coalescing (WC). Memory ordering semantics may vary for different attributes. Predicate **attribute** is used to find the attribute of a location.

Operation Tuple - A tuple containing necessary attributes is used to mathematically describe memory operations. Memory operation `i` is represented by a tuple $\langle$`P,Pc,Op,Var,Data,WrId,WrType,WrProc,Reg,UseReg,Id` $\rangle$, where

[3] We have formally captured 21 out of 24 rules from [2]. Semaphore operations, which require 3 rules, have yet to be defined.

requireLinearOrder - requireIrreflexiveTotal - requireTransitive - requireAsymmetric *requireWriteOperationOrder* - local/remote case - remote/remote case *requireProgramOrder* - acquire case - release case - fence case	*requireMemoryDataDependence* - MD:RAW - MD:WAR - MD:WAW *requireDataFlowDependence* - DF:RAR - DF:RAW - DF:WAR *requireCoherence* - local/local case - remote/remote case *requireAtomicWBRelease*	*requireReadValue* - validWr - validLocalWr - validRemoteWr - validDefaultWr - validRd *requireNoUCBypasss* *requireSequentialUC* - RAR case - RAW case - WAR case - WAW case

Table 1. The specification hierarchy of the Itanium memory ordering rules.

p $i = P$:	issuing processor
pc $i = Pc$:	program counter
op $i = Op$:	instruction type
var $i = Var$:	shared memory location
data $i = Data$:	data value
wrID $i = WrId$:	identifier of a write operation
wrType $i = WrType$:	type of a write operation
wrProc $i = WrProc$:	target processor of a write operation
reg $i = Reg$:	register
useReg $i = UesReg$:	flag of a write indicating if it uses a register
id $i = Id$:	global identifier of the operation

A read instruction or a fence instruction is decomposed into a single operation. A write instruction is decomposed into multiple operations, comprising a local write operation (**wrType** i = *Local*) and a set of remote write operations (**wrType** i = *Remote*) for each target processor (**wrProc** i), which also includes the issuing processor. Every write operation i that originates from a single write instruction shares the same program counter (**pc** i) and write ID (**WrID** i).

3.1 The Itanium Memory Ordering Rules

As shown below, predicate `legal` is a top-level constraint that defines the legality of a trace *ops* by checking the existence of an *order* among *ops* that satisfies all requirements. Each requirement is formally defined in the Appendix.

legal *ops* $\equiv \exists$ *order*.
  requireLinearOrder *ops order* $\wedge$

requireWriteOperationOrder *ops order* $\wedge$
requireProgramOrder *ops order* $\wedge$
requireMemoryDataDependence *ops order* $\wedge$
requireDataFlowDependence *ops order* $\wedge$
requireCoherence *ops order* $\wedge$
requireReadValue *ops order* $\wedge$
requireAtomicWBRelease *ops order* $\wedge$
requireSequentialUC *ops order* $\wedge$
requireNoUCBypass *ops order*

Table 1 illustrates the hierarchy of the Itanium memory model definition. Most constraints strictly follow the rules from [2]. We also explicitly add a predicate `requireLinearOrder` to capture the general ordering requirement since [2] has only English to convey this important ordering property.

General Ordering Requirement (Appendix A.1) This requires `order` to be an irreflexive total order which is also circuit-free.

Write Operation Order (Appendix A.2) This specifies the ordering among write operations originate from a single write instruction. It guarantees that no write can become visible remotely before it becomes visible locally.

Program Order (Appendix A.3) This restricts reordering among instructions of the same processor with respect to the program order.

Memory-Data Dependence (Appendix A.4) This restricts reordering among instructions from the same processor when they access *common locations.*

Data-Flow Dependence (Appendix A.5) This is supposed to specify how local *data dependency* and *control dependency* should be treated. However, this is an area that is not fully specified in [2]. Instead of pointing to an informal document as done in [2], we provide a formal specification covering most cases of data dependency, namely establishing data dependency between two memory operations by checking the conflict usages of local registers.[4] Although [2] outlines four categories for data-flow dependency (RAR, RAW, WAR, and WAW), the WAW case (a *write* here is actually a *read* in terms of register usage, e.g., `st a,r`) does not establish any value-based data dependence relation. Therefore, data dependency as specified in `orderedByLocalDepencence` is only setup by the first three cases.

[4] We do not cover branch instructions or indirect-mode instructions that also induce data dependency. We provide enough data dependency specification to let designers experiment with straight-line code that uses registers - this is an important requirement to support execution.

Coherence (Appendix A.6) This constrains the order of writes to a *common location*. If two writes to the same location with the attribute of `WB` or `UC` become visible to a processor in some order, they must become visible to all processors in that order.

Read Value (Appendix A.7) This defines what data can be observed by a read operation. There are three scenarios: a read can get the data from a local write (`validLocalWr`), a remote write (`validRemoteWr`), or the default value (`validDefaultWr`). In `validRemoteWr` we require that "the read is not ordered with the candidate remote write". It is slightly different from [2], which requires that "the candidate write is ordered with the read". This results from the difference in the way the ordering path is constructed. Since we do no have an explicit rule that establishes the order when a read gets the value from a write, the ordering relation between them would not pre-exist. In [2], a total order is implicitly imposed. Similar to shared memory read value rules, predicate `validRd` guarantees consistent assignments of registers - the value of a register is obtained from the most recent previous assignment of the same register.

Total Ordering of WB Releases (Appendix A.8) This specifies that store-releases to `write-back` (WB) memory must obey remote write atomicity, i.e., they become remotely visible atomically.

Sequentiality of UC Operations (Appendix A.9) This specifies that operations to `uncacheable`(UC) memory locations must have the property of *sequentiality*, i.e., they must become visible in program order.

No UC Bypassing (Appendix A.10) This specifies that `uncacheable`(UC) memory is not cacheable and does not allow local bypassing from `UC` writes.

4 Making the Itanium Memory Model Executable

We have developed two methods to analyze the Itanium memory model. The first, as mentioned earlier, uses Prolog backtracking search, augmented with finite-domain constraint solving. The second approach targets the powerful SAT engines that have recently emerged.

The Constraint Logic Programming Approach
Our formal Itanium specification is implemented in SICStus Prolog [16]. Litmus tests are contained in a separate test file.[5] When a test number is selected, the FD constraint solver examines all constraints automatically and answers whether the selected execution is legal. By running the litmus tests we can learn

[5] We have verified most of the sample programs provided by [2]. The only 3 (out of 17) examples we cannot do at this point involve disjoint accesses to memory locations. Other litmus tests can also be easily added.

```
P1                                  P2

(1) st_local(a,1);                  (7) ld.acq(1,b);
(2) st_remote1(a,1);                (8) ld(0,a);
(3) st_remote2(a,1);
(4) st.rel_local(b,1);
(5) st.rel_remote1(b,1);
(6) st.rel_remote2(b,1);
```

Fig. 3. An execution resulted from the program in Fig. 1. Stores are decomposed into local stores and remote stores. Loads are associated with return values.

the degree to which executions are constrained, i.e., we can obtain a general view of the global ordering relation between pairs of instructions.

Consider, for example, the program discussed earlier in Fig. 1. Its instructions are decomposed into operations as shown in Fig. 3. After taking this trace as input, the Prolog tool attempts all possible orders until it can find an instantiation that satisfies all constraints. For this particular example, it returns "illegal trace" as the result. If one comments out the `requireProgramOrder` rule and examines the trace again, the tool quickly finds a legal ordering matrix and a corresponding interleaving as shown in Fig. 4. Many other experiments can be conveniently performed in a similar way. Therefore, not only does this approach give people the notation to write rigorous as well as readable specifications, it also allows users to play with the model, asking "what if" queries after selectively enabling/disabling the ordering rules that are crucial to their work.

Although translating the formal specification to Prolog is fairly straightforward, there does exist some "logic gap" between predicate calculus and Prolog. Most Prolog systems do not directly support quantifiers. Therefore, we need to implement the effect of a universal quantifier by enumerating the related finite domain. The existential quantifier is realized by the back tracking mechanism of

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
1	0	1	1	0	0	0	0	0
2	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	0
4	1	1	1	0	1	1	1	0
5	1	1	1	0	0	1	1	0
6	1	1	1	0	0	0	1	0
7	1	1	1	0	0	0	0	0
8	1	1	1	1	1	1	1	0

Fig. 4. A legal ordering matrix for the execution shown in Fig. 3 when requireProgramOrder is disabled. A value 1 indicates that the two operations are ordered. A possible interleaving *8 4 5 6 7 1 2 3* is also automatically derived from this matrix.

Prolog when proper predicate conditions are set.

The SAT Approach

As an alternative method, we use our Prolog program as a driver to emit propositional formulae asserting the solvability of the constraints. After being converted to a standard format called DIMACS, the final formula is sent to a SAT solver, such as `berkmin` [17] or `zChaff` [18]. Although the clause-generation phase can be detached from the logic programming approach, the ability to have it coexist with FD-Prolog might be advantageous since it allows the two methods to share the same specification base. The complexity of boolean satisfiability is NP-Complete. However, tremendous progress has been achieved in recent years in SAT tools, making SAT solving an effective technique for industrial applications. According to our initial results, this seems to offer an encouraging path to tackling larger problems.

Performance Results

Performance statistics from some litmus tests is shown below. These tests are chosen from [2] and represented by their original table numbers. Performance is measured on a Dell Inspiron 3800 machine with 512 MB memory 700 MHz CPU. SICStus Prolog is run under compiled mode. The SAT solver used is `berkmin`.

Test	Result	FD Solver(sec)	Vars	Clauses	SAT(sec)	CNF Gen Time
[2, Table 5]	illegal	0.38	64	679	0.03	negligible
[2, Table 10]	legal	2.36	100	1280	0.01	negligible
[2, Table 15]	illegal	17.7	576	15706	0.05	a minute
[2, Table 18]	illegal	1.9	144	2125	0.01	few secs
[2, Table 19]	legal	3.8	144	2044	0.02	few secs

5 Conclusions

The setting in which contemporary memory models are expressed and analyzed needs to be improved. Towards this, we present a framework based on axiomatic specifications (expressed in higher order logic) of memory ordering requirements. It is straightforward to encode these requirements as constraint logic programs or, by an extra level of translation, as boolean satisfiability problems. In the latter case, one can employ current SAT tools to quickly answer whether certain executions are permitted or not. Our techniques are demonstrated through the adaptation and analysis of the Itanium memory model. Being able to tackle such a complex design also attests to the scalability of our framework for cutting-edge commercial architectures.

Our methodology provides several benefits. First, the ability to execute the underlying model is a powerful feature that promotes understanding. Second, the compositional specification style provides modularity, reusability, and scalability. It also allows one to add constraints incrementally for investigation purposes. Third, the expressive power of the underlying logic allows one to define a wide range of requirements using the same notation, providing a rich taxonomy for memory consistency models. Finally, the method of converting axiomatic rules to a propositional formula allows one to perform property checking through boolean reasoning, thus opening up new means to conduct formal verification.

Future Work One enhancement that can be made is to develop the capability of exercising *symbolic* (*non-ground*) litmus tests. Such a tool may be used to automatically synthesize critical instructions of concurrent code fragments comprising compiler idioms or other synchronization primitives. For example, one could imagine using a symbolic store instruction in a program and asking a tool to solve whether it should be an "ordinary" or a "release" store to help generate aggressive code. Another area of improvement is in reducing the logic gap between the formal specification and the tools that execute the specification. One possibility is to apply a *quantified boolean formulae* (QBF) solver that directly accepts quantifiers. The research of QBF solvers is still at a preliminary stage compared to propositional SAT. We hope our work can help accelerate its development by providing industrially motivated benchmarks.

References

1. Intel Itanium Architecture Software Developer's Manual, http://developer.intel.com/design/itanium/manuals.htm
2. A Formal Specification of Intel Itanium Processor Family Memory Ordering. Application Note, Document Number: 251429-001 (October, 2002)
3. K. Gharachorloo: Memory consistency models for shared-memory multiprocessors. Technical Report CSL-TR-95-685, Stanford University, (December 1995)
4. D. Dill, S. Park, A. Nowatzyk: Formal Specification of Abstract Memory Models. Research on Integrated Systems: Proceedings of the 1993 Symposium, Ed. G. Borriello and C. Ebeling, MIT Press (1993)
5. Prosenjit Chatterjee, Ganesh Gopalakrishnan: Towards a Formal Model of Shared Memory Consistency for Intel Itanium. ICCD 2001, Austin,TX (Sept 2001)
6. Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom: Specifying Java Thread Semantics Using a Uniform Memory Model, Joint ACM Java Grande - ISCOPE Conference (2002)
7. The SPARC Architecture Manual, Version 9, Prentice Hall (1993)
8. Leslie Lamport: How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. IEEE Transactions on Computers, 28(9): 690–691 (1979)
9. M. Ahamad, G. Neiger, James Burns, Prince Kohli, Philip Hutto: Causal Memory: Definitions, Implementation and Programming. Technical Report: GIT_CC-93/95 (July 1994)
10. Yuan Yu, through personal communication
11. Simplify, http://research.compaq.com/SRC/esc/Simplify.html
12. TLA+, http://research.microsoft.com/users/lamport/tla/tla.html
13. Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, Yuan Yu: Checking Cache-Coherence Protocols with TLA+ Formal Methods in System Design. Formal Methods in System Design, 22(2): 125-131 (Mar 2003)
14. M. J. C. Gordon, T. F. Melham: Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press (1993)

15. J. Jaffar, J-L. Lassez: Constraint Logic Programming. Principles Of Programming Languages, Munich, Germany (January 1987)
16. SICStus Prolog, http://www.sics.se/sicstus
17. E. Goldberg, Y. Novikov: BerkMin: a Fast and Robust Sat-Solver. Design, Automation and Test in Europe Conference and Exhibition Paris, France (2002)
18. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik: Chaff: Engineering an Efficient SAT Solver. 39th Design Automation Conference, Las Vegas (June 2001)

Appendix: Formal Itanium Memory Ordering Specification

A.1 General Ordering Requirement

$\mathbf{requireLinearOrder}\ \mathit{ops}\ \mathit{order} \equiv$
$\quad \mathbf{requireIrreflexiveTotal}\ \mathit{ops}\ \mathit{order}\ \wedge\ \mathbf{requireTransitive}\ \mathit{ops}\ \mathit{order} \wedge$
$\quad \mathbf{requireAsymmetric}\ \mathit{ops}\ \mathit{order}$

$\mathbf{requireIrreflexiveTotal}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.$
$\quad \mathbf{id}\ i \neq \mathbf{id}\ j \Rightarrow (\mathbf{order}\ i\ j\ \vee\ \mathbf{order}\ j\ i)$

$\mathbf{requireTransitive}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j, k \in \mathit{ops}.\ (\mathbf{order}\ i\ j\ \wedge\ \mathbf{order}\ j\ k) \Rightarrow \mathbf{order}\ i\ k$

$\mathbf{requireAsymmetric}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.\ \mathbf{order}\ i\ j \Rightarrow \neg(\mathbf{order}\ j\ i)$

A.2 Write Operation Order

$\mathbf{requireWriteOperationOrder}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.$
$\quad \mathbf{orderedByWriteOperation}\ i\ j \Rightarrow \mathbf{order}\ i\ j$

$\mathbf{orderedByWriteOperation}\ i\ j \equiv \mathbf{isWr}\ i\ \wedge\ \mathbf{isWr}\ j \wedge \mathbf{wrID}\ i = \mathbf{wrID}\ j \wedge$
$\quad (\mathbf{wrType}\ i = \mathbf{Local}\ \wedge\ \mathbf{wrType}\ j = \mathbf{Remote}\ \wedge\ \mathbf{wrProc}\ j = \mathbf{p}\ i \vee$
$\quad \mathbf{wrType}\ i = \mathbf{Remote}\ \wedge\ \mathbf{wrType}\ j = \mathbf{Remote} \wedge$
$\quad \mathbf{wrProc}\ i = \mathbf{p}\ i\ \wedge\ \mathbf{wrProc}\ j \neq \mathbf{p}\ i)$

A.3 Program Order

$\mathbf{requireProgramOrder}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.$
$\quad (\mathbf{orderedByAcquire}\ i\ j\ \vee \mathbf{orderedByRelease}\ i\ j\ \vee \mathbf{orderedByFence}\ i\ j) \Rightarrow$
$\quad \mathbf{order}\ i\ j$

$\mathbf{orderedByProgram}\ i\ j \equiv \mathbf{p}\ i = \mathbf{p}\ j\ \wedge\ \mathbf{pc}\ i < \mathbf{pc}\ j$

$\mathbf{orderedByAcquire}\ i\ j \equiv \mathbf{orderedByProgram}\ i\ j\ \wedge\ \mathbf{op}\ i = \mathit{ld.acq}$

$\mathbf{orderedByRelease}\ i\ j \equiv \mathbf{orderedByProgram}\ i\ j\ \wedge\ \mathbf{op}\ j = \mathit{st.rel} \wedge$
$\quad (\mathbf{isWr}\ i\ \Rightarrow (\mathbf{wrType}\ i = \mathit{Local}\ \wedge\ \mathbf{wrType}\ j = \mathit{Local} \vee$
$\quad \mathbf{wrType}\ i = \mathit{Remote}\ \wedge\ \mathbf{wrType}\ j = \mathit{Remote}\ \wedge\ \mathbf{wrProc}\ i = \mathbf{wrProc}\ j))$

$\mathbf{orderedByFence}\ i\ j \equiv \mathbf{orderedByProgram}\ i\ j\ \wedge\ (\mathbf{op}\ i = \mathit{mf}\ \vee\ \mathbf{op}\ j = \mathit{mf})$

A.4 Memory-Data Dependence

$\mathbf{requireMemoryDataDependence}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.$
$\quad (\mathbf{orderedByRAW}\ i\ j \lor \mathbf{orderedByWAR}\ i\ j\ \lor\ \mathbf{orderedByWAW}\ i\ j) \Rightarrow$
$\quad \mathbf{order}\ i\ j$

$\mathbf{orderedByMemoryData}\ i\ j \equiv \mathbf{orderedByProgram}\ i\ j\ \land\ \mathbf{var}\ i = \mathbf{var}\ j$

$\mathbf{orderedByRAW}\ i\ j \equiv$
$\quad \mathbf{orderedByMemoryData}\ i\ j\ \land\ \mathbf{isWr}\ i\ \land\ \mathbf{wrType}\ i = \mathit{Local}\ \land\ \mathbf{isRd}\ j$

$\mathbf{orderedByWAR}\ i\ j \equiv$
$\quad \mathbf{orderedByMemoryData}\ i\ j\ \land\ \mathbf{isRd}\ i\ \land\ \mathbf{isWr}\ j\ \land\ \mathbf{wrType}\ j = \mathit{Local}$

$\mathbf{orderedByWAW}\ i\ j \equiv \mathbf{orderedByMemoryData}\ i\ j\ \land\ \mathbf{isWr}\ i\ \land\ \mathbf{isWr}\ j \land$
$\quad (\mathbf{wrType}\ i = \mathit{Local}\ \land\ \mathbf{wrType}\ j = \mathit{Local} \lor$
$\quad \mathbf{wrType}\ i = \mathit{Remote}\ \land\ \mathbf{wrType}\ j = \mathit{Remote} \land$
$\quad \mathbf{wrProc}\ i = \mathbf{p}\ i\ \land\ \mathbf{wrProc}\ j = \mathbf{p}\ i)$

A.5 Data-Flow Dependence

$\mathbf{requireDataFlowDependence}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.$
$\quad \mathbf{orderedByLocalDepencence}\ i\ j \Rightarrow \mathbf{order}\ i\ j$

$\mathbf{orderedByLocalDepencence}\ i\ j \equiv \mathbf{orderedByProgram}\ i\ j\ \land\ \mathbf{reg}\ i = \mathbf{reg}\ j \land$
$\quad (\mathbf{isRd}\ i\ \land\ \mathbf{isRd}\ j \lor$
$\quad \mathbf{isWr}\ i\ \land\ \mathbf{wrType}\ i = \mathit{Local}\ \land\ \mathbf{useReg}\ i\ \land\ \mathbf{isRd}\ j \lor$
$\quad \mathbf{isRd}\ i\ \land\ \mathbf{isWr}\ j\ \land\ \mathbf{wrType}\ j = \mathit{Local}\ \land\ \mathbf{useReg}\ j)$

A.6 Coherence

$\mathbf{requireCoherence}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.$
$\quad (\mathbf{isWr}\ i\ \land\ \mathbf{isWr}\ j\ \land\ \mathbf{var}\ i = \mathbf{var}\ j \land$
$\quad (\mathbf{attribute}\ (\mathbf{var}\ i) = WB\ \lor\ \mathbf{attribute}\ (\mathbf{var}\ i) = UC) \land$
$\quad (\mathbf{wrType}\ i = \mathit{Local}\ \land\ \mathbf{wrType}\ j = \mathit{Local}\ \land\ \mathbf{p}\ i = \mathbf{p}\ j \lor$
$\quad \mathbf{wrType}\ i = \mathit{Remote}\ \land\ \mathbf{wrType}\ j = \mathit{Remote}\ \land\ \mathbf{wrProc}\ i = \mathbf{wrProc}\ j) \land$
$\quad \mathbf{order}\ i\ j)$
$\quad \Rightarrow$
$\quad (\forall\ p, q \in \mathit{ops}.$
$\quad (\mathbf{isWr}\ p\ \land\ \mathbf{isWr}\ q\ \land\ \mathbf{wrID}\ p = \mathbf{wrID}\ i\ \land\ \mathbf{wrID}\ q = \mathbf{wrID}\ j \land$
$\quad \mathbf{wrType}\ p = \mathit{Remote}\ \land\ \mathbf{wrType}\ q = \mathit{Remote}\ \land\ \mathbf{wrProc}\ p = \mathbf{wrProc}\ q) \Rightarrow$
$\quad \mathbf{order}\ p\ q)$

A.7 Read Value

$\mathbf{requireReadValue}\ \mathit{ops}\ \mathit{order} \equiv \forall\ j \in \mathit{ops}.$
$\quad (\mathbf{isRd}\ j \Rightarrow (\mathbf{validLocalWr}\ \mathit{ops}\ \mathit{order}\ j\ \lor\ \mathbf{validRemoteWr}\ \mathit{ops}\ \mathit{order}\ j\ \lor$
$\quad \mathbf{validDefaultWr}\ \mathit{ops}\ \mathit{order}\ j)) \land ((\mathbf{isWr}\ j \land \mathbf{useReg}\ j) \Rightarrow \mathbf{validRd}\ \mathit{ops}\ \mathit{order}\ j)$

$\textbf{validLocalWr}\ \mathit{ops}\ \mathit{order}\ j \equiv \exists\ i \in \mathit{ops}.$
$(\textbf{isWr}\ i\ \wedge\ \textbf{wrType}\ i = \mathit{Local}\ \wedge\ \textbf{var}\ i = \textbf{var}\ j\ \wedge\ \textbf{p}\ i = \textbf{p}\ j\ \wedge$
$\textbf{data}\ i\ = \textbf{data}\ j\ \wedge\ \textbf{order}\ i\ j)\ \wedge$
$(\neg\ \exists\ k \in \mathit{ops}.\ \textbf{isWr}\ k\ \wedge\ \textbf{wrType}\ k = \mathit{Local}\ \wedge\ \textbf{var}\ k = \textbf{var}\ j\ \wedge\ \textbf{p}\ k = \textbf{p}\ j\ \wedge$
$\textbf{order}\ i\ k\ \wedge\ \textbf{order}\ k\ j)$

$\textbf{validRemoteWr}\ \mathit{ops}\ \mathit{order}\ j \equiv \exists\ i \in \mathit{ops}.$
$(\textbf{isWr}\ i\ \wedge\ \textbf{wrType}\ i = \mathit{Remote}\ \wedge\ \textbf{wrProc}\ i = \textbf{p}\ j\ \wedge\ \textbf{var}\ i = \textbf{var}\ j\ \wedge$
$\textbf{data}\ j\ = \textbf{data}\ i\ \wedge\ \neg(\textbf{order}\ j\ i))\ \wedge$
$(\neg\ \exists\ k \in \mathit{ops}.\ \textbf{isWr}\ k \wedge \textbf{wrType}\ k = \mathit{Remote} \wedge \textbf{var}\ k = \textbf{var}\ j \wedge \textbf{wrProc}\ k = \textbf{p}\ j \wedge$
$\textbf{order}\ i\ k\ \wedge\ \textbf{order}\ k\ j)$

$\textbf{validDefaultWr}\ \mathit{ops}\ \mathit{order}\ j \equiv$
$(\neg\ \exists\ i \in \mathit{ops}.\ \textbf{isWr}\ i\ \wedge\ \textbf{var}\ i = \textbf{var}\ j\ \wedge\ \textbf{order}\ i\ j\ \wedge$
$(\textbf{wrType}\ i = \mathit{Local}\ \wedge\ \textbf{p}\ i = \textbf{p}\ j\ \vee\ \textbf{wrType}\ i = \mathit{Remote}\ \wedge\ \textbf{wrProc}\ i = \textbf{p}\ j)) \Rightarrow$
$\textbf{data}\ j = \textbf{default}\ (\textbf{var}\ j)$

$\textbf{validRd}\ \mathit{ops}\ \mathit{order}\ j \equiv \exists\ i \in \mathit{ops}.$
$(\textbf{isRd}\ i\ \wedge\ \textbf{reg}\ i = \textbf{reg}\ j\ \wedge\ \textbf{orderedByProgram}\ i\ j\ \wedge\ \textbf{data}\ j = \textbf{data}\ i)\ \wedge$
$(\neg\ \exists\ k \in \mathit{ops}.\ \textbf{isRd}\ k\ \wedge\ \textbf{reg}\ k = \textbf{reg}\ j\ \wedge$
$\textbf{orderedByProgram}\ i\ k\ \wedge\ \textbf{orderedByProgram}\ k\ j)$

A.8 Total Ordering of WB Releases

$\textbf{requireAtomicWBRelease}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j, k \in \mathit{ops}.$
$(\textbf{op}\ i = \mathit{st.rel} \wedge \textbf{wrType}\ i = \mathit{Remote} \wedge \textbf{op}\ k = \mathit{st.rel} \wedge \textbf{wrType}\ k = \mathit{Remote} \wedge$
$\textbf{wrID}\ i = \textbf{wrID}\ k\ \wedge\ \textbf{attribute}\ (\textbf{var}\ i) = WB\ \wedge\ \textbf{order}\ i\ j\ \wedge\ \textbf{order}\ j\ k) \Rightarrow$
$(\textbf{op}\ j = \mathit{st.rel}\ \wedge\ \textbf{wrType}\ j = \mathit{Remote} \wedge\ \textbf{wrID}\ j = \textbf{wrID}\ i)$

A.9 Sequentiality of UC Operations

$\textbf{requireSequentialUC}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j \in \mathit{ops}.\ \textbf{orderedByUC}\ i\ j \Rightarrow \textbf{order}\ i\ j$

$\textbf{orderedByUC}\ i\ j\ \equiv$
$\textbf{orderedByProgram}\ i\ j \wedge \textbf{attribute}\ (\textbf{var}\ i) = UC \wedge \textbf{attribute}\ (\textbf{var}\ j) = UC \wedge$
$(\textbf{isRd}\ i\ \wedge\ \textbf{isRd}\ j\ \vee$
$\textbf{isRd}\ i\ \wedge\ \textbf{isWr}\ j\ \wedge\ \textbf{wrType}\ j = \mathit{Local} \vee$
$\textbf{isWr}\ i \wedge\ \textbf{wrType}\ i = \mathit{Local}\ \wedge\ \textbf{isRd}\ j \vee$
$\textbf{isWr}\ i\ \wedge\ \textbf{wrType}\ i = \mathit{Local}\ \wedge\ \textbf{isWr}\ j\ \wedge\ \textbf{wrType}\ j = \mathit{Local})$

A.10 No UC Bypassing

$\textbf{requireNoUCBypass}\ \mathit{ops}\ \mathit{order} \equiv \forall\ i, j, k \in \mathit{ops}.$
$(\textbf{isWr}\ i\ \wedge\ \textbf{wrType}\ i = \mathit{Local}\ \wedge\ \textbf{attribute}\ (\textbf{var}\ i) = UC\ \wedge\ \textbf{isRd}\ j \wedge$
$\textbf{isWr}\ k\ \wedge\ \textbf{wrType}\ k = \mathit{Remote}\ \wedge\ \textbf{wrProc}\ k = \textbf{p}\ k\ \wedge\ \textbf{wrID}\ k = \textbf{wrID}\ i \wedge$
$\textbf{order}\ i\ j\ \wedge\ \textbf{order}\ j\ k) \Rightarrow$
$(\textbf{wrProc}\ k \neq \textbf{p}\ j\ \vee\ \textbf{var}\ i \neq \textbf{var}\ j)$