Litmus Tests for Comparing Memory Consistency Models: How Long Do They Need to Be?[*]

Sela Mador-Haim
University of Pennsylvania

Rajeev Alur
University of Pennsylvania

Milo M. K. Martin
University of Pennsylvania

ABSTRACT

Memory consistency litmus tests are small parallel programs that are designed to illustrate subtle differences between memory consistency models by exhibiting different outcomes for different models. In this paper, we show that for a class of memory models that is restricted yet expressive enough to include all store-atomic hardware memory models, litmus tests of a bounded size are sufficient for illustrating differences between memory consistency models in this class. We establish a bound of two threads and no more than six memory access instructions for differentiating litmus tests in this class of models. Thus, we can prove equivalence of two specification of memory consistency models in this class by exploring a bounded number of litmus tests. We build a tool for comparing memory models based on this result, and we use the tool to explore and map the space of this class of models.

Categories and Subject Descriptors

C.1.2 [**Multiple Data Stream Architectures (Multiprocessors)**]: Parallel processors; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Design, Theory, Verification

Keywords

Memory Consistency Models, Concurrency, Litmus Tests

1. INTRODUCTION

Well-defined memory consistency models are required for reasoning about the correctness of multi-core hardware, parallel software, and compiler optimizations. Developing

Test A	
T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 2$
Fence	Read $Y \rightarrow r2$
Read $Y \rightarrow r1$	Read $X \rightarrow r3$
Outcome: $r1 = 0$; $r2 = 2$; $r3 = 0$	

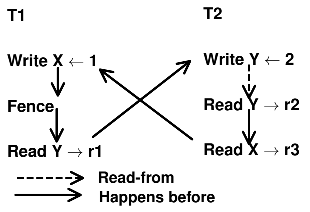

Figure 1: A litmus test for TSO

and understanding formal specifications of hardware memory models is a challenge due to the subtle differences in allowed reorderings. Architecture manuals include litmus tests, which are short parallel programs that can be used to identify the effect of subtle differences between memory consistency models on program execution, such as the program in Figure 1. They provide guidance to hardware developers for design validation and to software developers for writing multi-threaded programs.

Our recent work showed memory consistency models can be contrasted by systematically enumerating litmus tests of a bounded length [10]. We showed our technique can aid the process of developing specifications for hardware-level memory models, and we provided case studies where the technique detected subtle differences that might have gone undetected using a manually selected set of litmus tests. This technique, however, can not prove the equivalence of two models. Conventional wisdom is that contrasting litmus tests are typically short, but a bound for litmus tests has not been established.

Without restricting the class of memory models, there is no bound for the size of contrasting litmus tests. For example, for any bound k, we can define two models that behaves differently only for one specific test of size $k + 1$. Therefore, we need to consider a more restricted class of memory models.

This paper defines a class of memory models that is limited to show a bound, yet expressive enough to contain store-atomic hardware memory models, including Sun's SPARC [15], Intel's x86 [7] and Alpha [13]. This class is defined using a must-not-reorder function that specifies which instructions in the program cannot be reordered. We prove that short litmus tests consisting of two threads and up to three memory access instructions (reads and writes) in each thread are sufficient to illustrate differences between any two models in this class. Furthermore, we show that the number of non-memory-access operations is

[*]The authors acknowledge the support of NSF grants CCF-0905464 and CCF-0644197, and of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

also bounded and depends on the choice of predicates in the must-not-reorder function. In addition, we show that instantiations of seven different templates are sufficient for contrasting all memory models in this class. Consequently, we reduce the number of litmus tests necessary for equivalence checking of two memory models by several orders of magnitude over naive enumeration.

We explore this class of memory models by testing for equivalence between pairs of models in this class. We select a set of predicates that specifies the commonly used properties of memory models including reordering of different memory access instructions, fences, and data dependencies. We show that for this class of memory models, consisting of 90 different models, there are eight pairs of equivalent models, and identify nine litmus tests that are sufficient for differentiating all memory models in this class.

The main contributions of this paper are: (1) proving a theorem that bounds the size of contrasting litmus tests, which justifies the conventional wisdom regarding the size of litmus tests; (2) describing a tool for comparing memory consistency models, which works in a reasonable time (seconds) and can show the equivalence between memory models based on the theorem we proved; and (3) exploring the space of memory models in our class of models to show how different choices in the specification affect the models and which models are equivalent.

2. SPECIFYING MEMORY MODELS

This section defines the class of memory models studied in this paper. We provide a definition for an expressive but limited class of memory models, describe the predicates used for defining the models, and show how different known memory models are defined in our framework.

2.1 Program Executions

A *parallel program* P is a set of concurrently executed threads, where each *thread* is a sequence of instructions. In the context of memory models, we classify instructions into two groups: memory access instructions that read and write to memory, and non-memory access instructions, consisting of all other instruction including memory fences, arithmetic operations, and branches.

Memory consistency models define the possible behaviors of a parallel program and constrain the values each read may observe. A general way to define a memory model is as a set of allowed *program executions*. Informally, a program execution specifies the sequence of instructions that were executed in each thread, annotated with the actual values for all the involved registers. In each step, a program executes an instruction with concrete values for all of the involved registers. An instance of instruction i, annotated with the values for all the involved registers, is called an *instruction execution*.

A thread may exhibit many different executions, because an execution usually depends on the other threads. A *thread execution*, α_t is a sequence of instruction executions in the order they are executed in thread t. In case an instruction is executed more than once (due to a loop), there can be several instruction executions that correspond to the same instruction, so loops are unrolled. As an example for thread executions, suppose a thread t reads from memory to registers $r1$ and $r2$, computes $r3 = r1 + r2$, and then writes $r3$ to memory. An execution α_t of that thread could have any value for $r1$ and $r2$, but each write would always be $r1 + r2$.

The order of two instruction executions x and y with respect to a given thread execution α_t is called *program order*. We use the notation $x < y$ when x precedes y in α_t. A *program execution*, α_P, associates a thread execution with each thread in a program P. A memory model M is defined as a set of allowed program executions. For two memory models, M_1 and M_2, we say that $M_1 \subseteq M_2$ if and only if for every program execution α_P, $\alpha_P \in M_1$ implies $\alpha_P \in M_2$. If $M_1 \subseteq M_2$ and $M_2 \subseteq M_1$, the two models are equivalent.

2.2 The Class of Memory Models

Our approach is to consider a class of memory models that is limited enough to bound the size of contrasting litmus tests yet expressive enough to include most existing hardware memory models. The class of memory models we consider is a class of relaxed memory models that allow reorderings of local instructions. Memory access operations (read and write) may be performed out of program order. The types of instructions that can and cannot be reordered vary between different memory models. We allow a thread to read its own writes early, but do not allow it to read other threads' writes early [1]. Thus, this class of memory models is expressive enough to include most hardware memory models, including Sequential Consistency (SC) [9], Sun's SPARC [15] and Intel's x86 [7], but not non-store-atomic models like PowerPC [12].

These memory models are defined using a *must-not-reorder* function F that maps a pair of instructions to a boolean. Intuitively, if $F(x, y)$ is true for instructions x and y of the same thread, these instructions cannot be reordered and must be executed in program order. Based on the choice of F, we define which program executions are allowed, using two relations between instruction executions in a program execution α_P: A *read-from map* $\mapsto$, mapping reads to the writes they are reading from, and a *happens-before* order $\Rightarrow$, representing a global order of instruction executions in the program.

Given a program execution α_P and a must-not-reorder function F, a relation $\mapsto$ is a read-from relation between instruction executions in α_P, if: (1) if $x \mapsto y$, then x is a write, y is a read, and the value read by y is same as the value written by x; (2) if $x \mapsto y$ and $z \mapsto y$, then $x = z$ (only one write is mapped to each read); (3) if x is a read and there is no write y such that $y \mapsto x$, then x reads the initial value; and (4) if $x > y$, then $x \not\mapsto y$ (cannot read from a future write in the same thread).

Given a program execution α_P, a must-not-reorder function F and a read-from map $\mapsto$, a happens before relation $\Rightarrow$ is a partial order between instruction executions in α_P with the following properties:

1. **Program order**: If $F(x, y)$ and $y > x$ then $x \Rightarrow y$.
2. **Write-write**: If x and y are both writes to the same address, then either $x \Rightarrow y$ or $y \Rightarrow x$.
3. **Write-read**: If $x \mapsto y$ and x and y are from different threads, then $x \Rightarrow y$.
4. **Read-Write**: If x is a read and y is a write to the same address such that $y \not\mapsto x$, and there is no write z such that $z \mapsto x$ and $y \Rightarrow z$, then $x \Rightarrow y$.
5. **Ignore local**: If $x > y$ then $x \not\Rightarrow y$.

A program execution α_P is allowed in M_F, the model defined by F, if for some read-from relation $\mapsto$ for α_P there is a happens-before relation $\Rightarrow$ which is acyclic.

2.3 Must-not-reorder Function Predicates

The must-not-reorder function F we use in our class of models is a quantifier-free positive boolean formula, constructed from a set of predicates D on instruction executions. Predicates are either unary or binary, and are defined for instruction executions x, y in α_P.

For example, some commonly used predicates are: $Read(x)$, $Write(x)$, $Fence(x)$ — the instruction is a read, a write or a fence, $DataDep(x, y)$, $ControlDep(x, y)$ — data and control dependence, and $SameAddr(x, y)$ — x and y access the same address.

Our analysis is not restricted to a specific set of predicates. However, we require all predicates to preserve register, address, and value symmetries for read and write operations.

2.4 Memory Model Examples

Using the must-not-reorder function, we can define different hardware models. For example, Sequential Consistency (SC) does not allow any reordering and is therefore specified using $F_{SC} = True$.

IBM370's memory model allows reordering writes after reads, except reads to the same address. $F_{IBM370}(x, y) = (Write(x) \wedge Read(y) \wedge SameAddr(x, y)) \vee (Write(x) \wedge Write(y)) \vee Read(x) \vee Fence(x) \vee Fence(y)$.

SPARC's Total Store Order (TSO) allows reordering writes after reads, including reads to the same address. In case a write is ordered after a read to the same address, there is an effect of load forwarding, where a load observes local writes before they become visible to other threads. As seen in Figure 1, the $\Rightarrow$ relation does not include write-read edges between writes and reads in the same thread. There is no happens-before edge from Write $Y \leftarrow 2$ to Read $Y \rightarrow r2$ and thus $\Rightarrow$ is acyclic. $F_{TSO}(x, y) = (Write(x) \wedge Write(y)) \vee Read(x) \vee Fence(x) \vee Fence(y)$.

Finally, SPARC's Relaxed Memory Order (RMO) allows reordering everything except fences, dependent instructions and read/write instructions after a write to the same address. $F_{RMO}(x, y) = (Write(y) \wedge SameAddr(x, y)) \vee Fence(x) \vee Fence(y) \vee DataDep(x, y) \vee ControlDep(x, y)$.

3. SMALL LITMUS TEST THEOREM

Given two memory model specifications, we want to find whether they are equal or different. We show that for the family of memory models defined in Section 2.2, litmus tests with a bounded size are sufficient. We find the bound for these tests in terms of the number of memory accesses in the test and number of threads.

THEOREM 1. *For every two memory models, M_1 and M_2, that are defined via a must-not-reorder function, if $M_2 \not\subseteq M_1$, then there is a test P and an execution α_P with two threads and up to six memory access operations, such that $\alpha_P \notin M_1$, $\alpha_P \in M_2$.*

Section 3.1 defines conflict cycles, and proves that for any two observably different models, there is an execution consisting only of one conflict cycle that is feasible in one model but not in the other. Section 3.2 constructs a minimal test from a conflict cycle, and thus proves Theorem 1. Section 3.3 shows a bound for non-memory-access instructions, and Section 3.4 shows how to further reduce the number of litmus tests.

3.1 Conflict Cycle

Given two memory models M_1 and M_2, if $M_2 \not\subseteq M_1$, there is a test P and execution α_P such that $\alpha_P \in M_2$ and $\alpha_P \notin M_1$. In this case, there is a read-from map $\mapsto$ and a happens-before relation $\Rightarrow_2$ for α_P and M_2, such that $\Rightarrow_2$ is acyclic, and for every read-from map (including $\mapsto$) and $\Rightarrow_1$ for α_P and M_1, $\Rightarrow_1$ is cyclic.

Given $\Rightarrow_1$, a happens-before relation for $M_1, \alpha_P, \mapsto$, let C be the set of instruction executions in the smallest cycle in $\Rightarrow_1$. We construct the following execution, $\alpha_{P'}$, based on the instructions in C:

1. If $x \in C$ then $x \in \alpha_{P'}$.
2. If x is a write, y is a read and $x \Rightarrow_1 y$, the value read by y in $\alpha_{P'}$ is the value written by x
3. If x is a read, y is a write and $x \Rightarrow_1 y$, the value read by x in $\alpha_{P'}$ is the initial value.
4. If x, y are write instructions and $x \Rightarrow_1 y$, we add a new read instruction z at the end of the thread of x, reading the value written by y.

LEMMA 1. *$\alpha_{P'}$ is in M_2 but not in M_1*

Proof: $\Rightarrow_2$, the happens-before relation for M_2 is acyclic, and therefore the instructions in C do not form a cycle in $\Rightarrow_2$. There are two instructions $x, y \in C$, such that $x \Rightarrow_1 y$, $x \not\Rightarrow_2 y$ and there is no other instruction before x connected to any instruction after y in the graph of $\Rightarrow_2$ (there is no bypass edge). We call this edge between x and y a *critical edge*. The only source of difference between $\Rightarrow_1$ and $\Rightarrow_2$ is the difference in the must-not-reorder function, and therefore x and y belong to the same thread.

The only edges $\alpha_{P'}$ adds to C are due to the added read operations. These operations have incoming edges but no outgoing edges and therefore can not form a cycle. Hence, $\alpha_{P'}$ has an acyclic happens-before relation in M_2.

For any happens-before relation $\Rightarrow_{P'}$ for $\alpha_{P'}$ and M_1 and any x, y in C, if $x \Rightarrow_1 y$ is a program order edge, then $x \Rightarrow_{P'} y$ as well, because the must-not-reorder function stays the same. If x is a write and y is a read, then $x \Rightarrow_{P'} y$ because y still reads the value written by x. If x is a read and y is a write, then $x \Rightarrow_{P'} y$ because x reads the initial value and all writes precede it. If x and y are both writes, $x \Rightarrow_{P'} y$ as well, because the added read in x's thread sees the value of y, which is possible only if y precedes x. All the edges in the cycle of $\Rightarrow_1$ are preserved in $\Rightarrow_{P'}$, and therefore it is cyclic.

3.2 Constructing Minimal Test

A *segment* is a sequence of instructions, connected by program-order edges, that starts with a memory access operation (read or write), ends with a memory access, and has no other memory access in between them. We can classify the segments according to the type of memory accesses: read-read, read-write, write-read, and write-write.

A segment that contains a critical edge is a *critical segment*. We can now prove Theorem 1 by showing that for each type of critical segment, we can construct a litmus test with only two threads and up to six memory access operations, as illustrated in the diagrams in Figure 2.

Case 1. The critical segment is read-write. Use this segment at thread T_1. Add an identical segment for T_2, changing the address of the read to match the write in T_1, and the write to match the read in T_1. Total number of memory access operations: four.

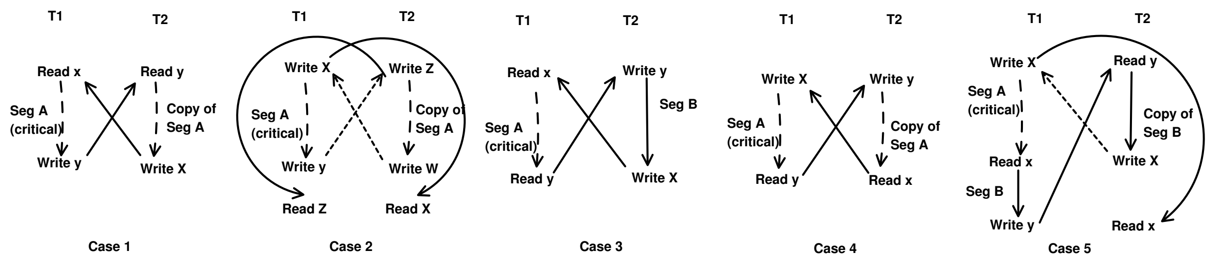

Figure 2: Litmus test templates by critical segment

Case 2. The critical segment is a write-write. Use this segment for thread T_1, duplicate it for thread T_2, switching addresses. Add a read at the end of T_1, reading the value of the first write in T_2, and a read at the end of T_2, reading the value of the first write in T_1. Total number of memory access operations: six.

Case 3. The critical segment is a read-read. Because there are no inter-thread read-read edges in $\Rightarrow$, there *must* be either a write-write segment or both a write-read segment $W1, ...R1$ and a read-write segment $R2, ...W2$ in the cycle. In the latter case, according to the symmetry requirement in Section 2.3, there is a symmetric segment $R2', ...W2'$ such that $R2' = R1$. Therefore, we merge both segments into a write-write segment: $W1, ...R1, ...W2'$. Use the read-read segment for thread T_1 and the write-write segment for thread T_2. Total number of memory access operations: four to five.

Case 4. The critical segment is a write-read to different addresses. Use this segment as T_1, duplicate it for T_2, change the read in T_2 to match the address of the write in T_1 and vice versa. Each read gets the initial value. Total number of memory access operations: four.

Case 5. Like Case 4, but write and read are both to the same address. If there is a segment with a write and then a read to the same address, there cannot be a memory-access read-write edge involving this read because according to our definition of a minimal cycle, the read should receive the initial value, which is impossible in this case. So we conclude there is another read-read or read-write segment in the same thread.

1. If there is a read-read segment to two different addresses, merge it after the critical segment (combined segment have three operations), and continue as in Case 4. Total number of memory access operations: six.
2. If there is a read-write segment, merge it after the critical segment, resulting in a write-write segment. Copy the read-write segment to T_2, connect the end of T_1 with T_2 using a write-read edge, and connect the end of T_2 with the beginning of T_1 (adding a read at the end of T_2 as previously discussed). Total number of memory access operations: six.

3.3 Local Segments

Theorem 1 bounds the number of threads and the number of memory access operations (reads and writes) required in a litmus test. However, additional instructions such as fences, arithmetic operations or branches affect the dependency relations between memory access instructions and therefore may be required. The bound for these additional instructions depends on the specific choice of predicates in D.

For example, consider a hypothetical model with n special fence instructions $f_1, ...f_n$ and the predicate $special(x, y)$ which is true if either: (1) x is a memory access instruction and $y = f_1$, (2) $x = f_n$ and y is a memory access, or (3) $x = f_i$ and $y = f_{i+1}$. Consider $F_1(x, y) = SameAddr(x, y) \vee special(x, y)$ and $F_2(x, y) = SameAddr(x, y)$. Any litmus test contrasting F_1 and F_2 should include a local segment of $n+2$ instructions such as $Read\ X, f_1, ...f_n, Write\ y$. Therefore, the minimal number of non-memory access instructions in a local segment depends on the choice of predicates and the instruction set.

The length of local segments is bounded by the number of equivalence classes of instructions according to our choice of predicates. Given a set of predicates D, two instruction x and y are equivalent with respect to D, $x \equiv_D y$ if for every predicate $d \in D$ and every instruction z, $d(x, z) = d(y, z)$ and $d(z, x) = d(z, y)$. Consider the memory model with a must-not-reorder function F_1, and a segment $i_1, ...i_n$ in a minimal conflict cycle for this model. Because it is a segment in a cycle, for every two adjacent instructions i_j, i_{j+1}, $F_1(i_j, i_{j+1})$ is true. Suppose two instructions in the segment are equivalent $i_j =_D i_k$ $(1 < j < k < n)$, then $F_1(i_j, i_{k+1})$ is also true and therefore we can reduce the segment to $i_1, ..i_j, i_{k+1}, ..i_n$, in contradiction to the minimality of the cycles. We conclude that a local segment cannot contain two equivalent non-memory-access instructions, and therefore its length is bounded by the number of equivalence classes for these instructions.

3.4 Reducing the Number of Litmus Tests

A consequence of the proof of Theorem 1 in Section 3.2 is that not only we can bound the size of the litmus tests, we can further reduce the number of litmus tests by exploring the cases described in the proof of Theorem 1. There are five different cases listed in the proof, and two of the cases (Case 3 and Case 5) are split into two sub-cases, which amounts to a total of seven templates. We can therefore compare memory models by instantiating these seven templates with all possible local segments.

Two segments s_1, s_2 are equivalent with respect to D if they are of the same length and for every pair of instructions in s_1, every predicate in D would have the same value as for a pair of instructions in the same position in s_2. Corollary 1 gives a bound for the number of tests as a function of the number of distinct segments of each type.

COROLLARY 1. *Suppose the number of distinct local segments of each type given by N_{WW}, N_{WR}, N_{RW}, and N_{RR}. The total number of required tests is given by $N_{RW} + N_{WW} + N_{RR}(N_{WW} + N_{WR} \times N_{RW}) + N_{WR}(1 + N_{RR} + N_{RW})$*

Test L1	
T1	T2
Write $X \leftarrow 1$ Write $Y \leftarrow 1$	Read $Y \rightarrow r1$ Fence Read $X \rightarrow r2$
Outcome: $r1 = 1$; $r2 = 0$	

Test L2	
T1	T2
Write $X \leftarrow 1$ Write $X \leftarrow 2$	Read $X \rightarrow r1$ Read $X \rightarrow r2$
Outcome: $r1 = 2$; $r2 = 0$	

Test L3	
T1	T2
Write $X \leftarrow 1$ Fence Write $Y \leftarrow 2$	Read $Y \rightarrow r1$ Read $X \rightarrow r2$
Outcome: $r1 = 2$; $r2 = 0$	

Test L4	
T1	T2
Write $X \leftarrow 1$ Fence Write $Y \leftarrow 2$	Read $Y \rightarrow r1$ t1 = r1-r1+X Read $[t1] \rightarrow r2$
Outcome: $r1 = 2$; $r2 = 0$	

Test L5	
T1	T2
Read $X \rightarrow r1$ Write $Y \leftarrow 1$	Read $Y \rightarrow r2$ Write $X \leftarrow 1$
Outcome: $r1 = 1$; $r2 = 1$	

Test L6	
T1	T2
Read $X \rightarrow r1$ t1 = r1-r1+1 Write $Y \leftarrow t1$	Read $Y \rightarrow r2$ t2 = r2-r2+1 Write $X \leftarrow t2$
Outcome: $r1 = 1$; $r2 = 1$	

Test L7	
T1	T2
Write $X \leftarrow 1$ Read $Y \rightarrow r1$	Write $Y \leftarrow 1$ Read $X \rightarrow r2$
Outcome: $r1 = 0$; $r2 = 0$	

Test L8	
T1	T2
Write $X \leftarrow 1$ Read $X \rightarrow r1$ t1 = r1-r1+Y Read $[t1] \rightarrow r2$	Write $Y \leftarrow 1$ Read $Y \rightarrow r3$ t2 = r3-r3+X Read $[t2] \rightarrow r4$
Outcome: $r1 = 1$; $r2 = 0$; $r3 = 1$; $r4 = 0$	

Test L9	
T1	T2
Write $X \leftarrow 1$ Read $X \rightarrow r1$ t1 = r1-r1+1 Write $Y \leftarrow [t1]$	Read $Y \rightarrow r2$ t2 = r2-r2+2 Write $X \rightarrow t2$ Read $X \rightarrow r3$
Outcome: $r1 = 1$; $r2 = 1$; $r3 = 1$	

Figure 3: Contrasting litmus tests

As discussed in Section 3.3, it is sufficient to know the set of predicates to bound the number of local segments. For example, suppose the predicates are: $Read(x)$, $Write(x)$, $Fence(x)$, $SameAddr(x, y)$ and $DataDep(x, y)$. For read-write segments, we need to consider segments with only independent read and write, with dependent read and write, and a segment with a fence between the read and the write. For each of these three cases, we need to consider read and write to the same address and to different addresses, so $N_{RW} = 6$ and similarly $N_{RR} = 6$. For write-read and write write segments we do not need to consider dependencies (writes do not generate dependencies), so $N_{WR} = N_{WW} = 4$. According to Corollary 1, we need a total of 230 tests to contrast memory models expressible with these predicates. Similarly, without data dependencies, we need 124 tests.

A naive enumeration of all tests within the bounds of Theorem 1 results in approximately million tests even without dependencies. Our earlier work describes optimizations that reduce the number of tests to several thousands [10]. This paper improves upon earlier work by more than an order of magnitude.

4. EXPERIMENTAL RESULTS

4.1 Implementation

We implemented a tool for contrasting memory model specifications via systematic exploration of litmus tests. This tool improves upon our previous work [10] by reducing the number of tests using the method described in Section 3.4. The memory models are specified using a must-not-reorder function and the axioms in Section 2.2. We use the SAT solver mini-sat [6] to test if a litmus test is admissible for a given memory model.

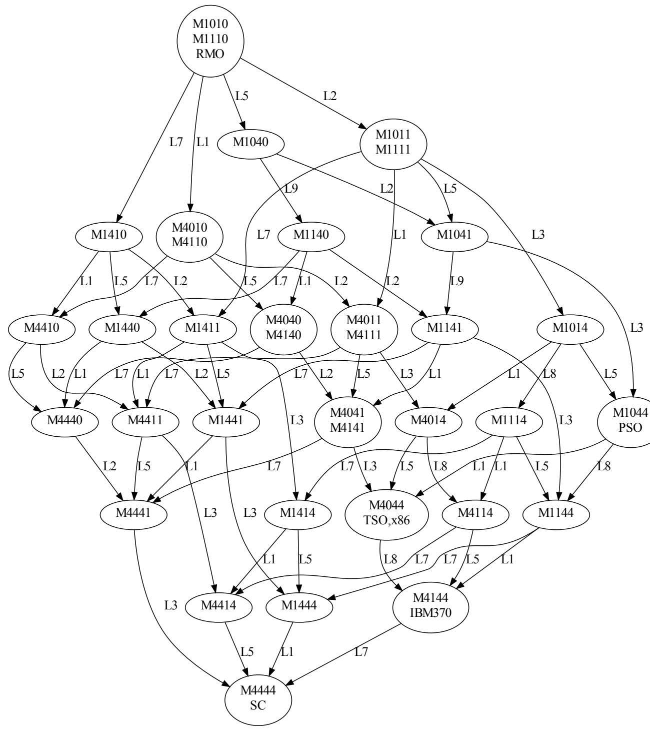

Figure 4: Relation between explored models (without data dependencies)

4.2 Exploring the Space of Memory Models

Using this tool, we performed an exhaustive exploration of all memory models that are expressible using the framework described in Section 2.2, with the predicates: $Read(x)$, $Write(x)$, $Fence(x)$, $SameAddr(x, y)$ and $DataDep(x, y)$. This set of predicates is sufficient to describe most common properties of memory models, including data dependencies. Models expressible in this framework include IBM370, Intel's x86, SPARC's TSO, PSO and variants of RMO and Alpha (for a complete specification of RMO and Alpha, we need to add control dependencies, which were not implemented in the tool but are supported by our framework). As discussed in Section 3.4, it is sufficient to check 230 litmus tests to contrast all models when using the above set of predicates.

Based in the selected predicates, there are five possible choices for each of the four pairs of memory operations (write-write, write-read, read-write and read-read). The options to allow reordering are: (0) always reorder, (1) reorder accesses to different addresses, (2) reorder if there are no data dependencies, (3) reorder if different addresses and no data dependencies, and (4) never reorder.

Some of the above options can be eliminated in certain cases. Reordering read-write and write-write with the same address violates single-thread consistency and therefore we do not consider them. Additionally, there is no need to consider dependencies for write-read and write-write. After eliminating these cases, there are two choices for write-write, three choices for write-read and read-write, and all five choices are available for read-read, which result in 90 possible memory models.

Using our tool, we compared these 90 models with each other. The tool tested whether each model is equivalent or strictly stronger than the other models, and which litmus tests can be used to contrast each pair of models. Comparing each pair of models took a few seconds, and a pairwise comparison of all 90 models completed in 20 minutes.

Out of the 90 different models, eight pairs of models are equivalent. All equivalent pairs of models differ only with the choice of whether to allow reordering of writes with later reads to the same address. Furthermore, a set of nine different litmus tests is sufficient to contrast any two non-equivalent memory models in this space. Figure 3 shows the set of nine litmus tests. The relationships between the explored models is shown in Figure 4. The direction of the arrows is from weaker to stronger models, and the labels on the edges are the litmus tests that distinguish between the models. Due to space considerations, Figure 4 does not include models with data dependencies.

A further analysis of this minimal set of litmus tests shows that tests L1 to L7 in Figure 3 correspond directly to the choices in model enumeration. For example, test L5 checks if a read can be reordered after an independent write to a different address, and test L6 checks if a read can be reordered after a dependent write to the same address (ignoring data dependencies). One exception is in the case of reordering writes after later reads to the same address. As shown in Case 5 of the proof of Theorem 1, when the critical segment is a write-read segment to the same address, either a read-read or a write-read consecutive segment is required to close a cycle. This leads to litmus tests L8 and L9 in Figure 3, where Test L8 detects write-read reordering in models that do not allow reordering reads to a different address (with or without dependencies) cannot be reordered, and Test L9 is relevant for models that do not allow reordering reads with later writes. In models that allow reordering both read-write and dependent read-read, no test can observe reordering writes after reads. These models are the eight pairs of equivalent models found by our experiments.

5. RELATED WORK

There has been a considerable amount of work on defining frameworks for specifying memory consistency models [1, 2, 3, 4, 11, 14, 16]. The concept of happens-before partial order between events was introduced by Lamport [8] and then by Adve and Hill [2] in the context of memory consistency models. Burckhardt and Musuvathi [5] provide a definition for SPARC's TSO using a happens-before relation. Alglave et al [3] define a framework for specifying hardware memory consistency models using a happens-before relation. Our class of memory models is similar to the one described by Alglave et al [3]. Our previous work describes a technique for contrasting memory models via automatic generation of litmus tests up to a certain bound, but we did not identify a bound for showing equivalence [10].

6. CONCLUSIONS

Even though litmus tests listed in architectural manuals are typically short, no bound was previously known for the size of litmus tests for differentiating memory consistency models. In this paper, we showed we can indeed bound the size of litmus tests sufficient for contrasting all models in a class that is expressive enough to include all existing store-atomic memory consistency models. Furthermore, we showed that for a selected set of predicates that represents the common features of hardware memory models, a set of nine litmus tests contrast all models in this class. This set of litmus tests may assist developing specifications of memory models, in this class, in validating hardware, and for presenting memory models in architectural manuals.

One shortcoming of this work is that it is limited to store-atomic memory models, in which all threads observe writes in the same order. Some memory models such as PowerPC [12] are non-store atomic and thus allow each thread to observe writes in a different order. These models are not included in the class of models defined in this paper, and they are known to require larger litmus tests with more than two threads. Extending this work to include non-store-atomic models is future work.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 1996.

[2] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *ISCA*, 1990.

[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In *CAV*, 2010.

[4] Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *ISCA*, 2006.

[5] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *CAV*, 2008.

[6] Niklas Een and Niklas Sorensson. Minisat - a SAT solver with conflict-clause minimization. In *SAT*, 2005.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, March 2010.

[8] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.

[9] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[10] Sela Mador-Haim, Rajeev Alur, and Milo Martin. Generating litmus tests for contrasting memory consistency models. In *CAV*, 2010.

[11] Vijay A. Saraswat, Radha Jagadeesan, Maged Michael, and Christoph von Praun. A theory of memory models. In *PPoPP*, 2007.

[12] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In *PLDI*, 2011.

[13] Richard L. Sites. *Alpha Architecture Reference Manual.* Prentice Hall PTR, 1992.

[14] Robert C. Steinke and Gary J. Nutt. A unified theory of shared memory consistency. *J. ACM*, 51(5), 2004.

[15] David L. Weaver and Tom Germond. *The SPARC Architecture Manual Version 9.* Prentice Hall PTR, 1994.

[16] Yue Yang, Ganesh Gopalakrishnan, Gary Lindstrom, and Konrad Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. *IPDPS*, 1, 2004.