Towards a Formal Model of Shared Memory Consistency for Intel Itanium^{TM}

Prosenjit Chatterjee and Ganesh Gopalakrishnan *
School of Computing, University of Utah
Technical Report UUCS-01-003
http://www.cs.utah.edu/formal_verification/

Abstract

We provide a simple formal model for Itanium^{TM} shared memory consistency [1, 2] covering a core set of instructions. Existing descriptions of Itanium shared memory consistency are based on an informal collection of ordering rules as well as several examples. Our operational model employs widely understood data structures such as buffers and memories, and expresses ordering constraints precisely using a collection of nondeterministic rules. This can enable the construction of reliable prototype implementations, formal verification against implementations, formal verification against other formal models, as well as verification of synchronization routines. Our model covers all published ordering constraints, and also sheds light on tricky concepts such as causality.

1 Introduction

The Itanium^{TM} shared memory consistency model [1, 2] is described in terms of a collection of ordering rules, constraints stated in English, and examples of legal and illegal executions. While good for initial understanding, such descriptions often leave many details unanswered. This can make it difficult for programmers to write reliable MP libraries. As far as we know, a formal specification (operational or otherwise) has not yet been published for Itanium.

In this paper, we provide a simple execution oriented (operational) model for the Itanium shared memory consistency reverse-engineered from [1, 2]. We believe that it accurately (and more completely) describes alternative descriptions publicly available. The availability of an operational model can help designers build executable prototypes to gain deeper understanding. In addition, they can use model-checkers to gain a deeper understanding with respect to synchronization routines as well as specific ordering issues [3, 4]. Like any formal specification, an operational model runs the risk of being over- or under-specified. In this paper we point out, as space permits[1]how we have strived to avoid these risks.

Our model deals with cacheable memory instructions consisting of *acquire load*s (written *ld.acq*), *ordinary load*s (*ld*), *release store*s (*st.rel*), and *ordinary store*s (*st*), as well as memory fences. It does not currently handle atomic read-modify-writes, noncacheable memory, or special rules pertaining to data dependencies involving registers [1, Section 13.2]. Despite its simplicity, our model captures all published ordering properties of the instructions we consider, and also sheds more light on corner cases pertaining to causality. While operational models have been proposed for commercial shared memory systems (notably for Sparc V9 [5]), a notable feature of our operational model is its use of a few explicit devices such as *vector timestamps* [6] to clearly describe the tricky notion of *causality.*

2 Overview of the Itanium Memory Model

The Itanium memory model can be understood in terms of *program* and *global visibility* ("visibility") orders. For memory operations of type 'store', visibility refers to when the effects of the store become apparent to all processors. For memory operations of type 'load,' visibility refers to when the execution of load appears to have been carried out for the processor carrying out the load. (All other processors do not directly observe the load happening.) As in [1], for two different memory operations X and Y, $X \rangle\rangle Y$ specifies that X is before Y in program order, $X \rightarrow Y$ indicates that Y must be visible only after X is visible. Further, if

*This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406

[1] Details appear in our webpage.

$X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$. Now, if X and Y are two memory operations in the same program, and $X \rangle\rangle Y$, Itanium requires the following:

1. If X is a load (store) and Y is a store (load) to the same location, WAR (RAW) hazards must be avoided.
2. If X and Y are stores to the same location, WAW hazards must be avoided. In addition, $X \rightarrow Y$.
3. If X and Y are memory operations to any location, and have a fence between them, then $X \rightarrow Y$.
4. If X is an Acquire load and Y is any other memory operation to any location, then $X \rightarrow Y$.
5. If X is any memory operation to any location and Y is a Release store, then $X \rightarrow Y$.

Under all other circumstances, X and Y can get executed in any order. [1] also asserts the following constraints on executions:

Coherence: There is a single visibility order (which is also a total order) of all stores per memory location observed by all the processors. Further, this total order is consistent with the program order for memory operations on that location in each processor,

RC_tso: Intuitively, *ld.acq* and *st.rel* are used to "bracket" instruction sequences, to permit more liberal execution orders for instructions in-between. To a crude approximation, *ld.acq* and *st.rel* are strongly ordered as in sequential consistency [7]. However, more precisely viewed, RC_tso [2] captures the orderings involving *ld.acq* and *st.rel* as per *Release Consistency* [7], with *ld.acq* and *st.rel* obeying TSO [5]. Under RC_tso, there is a single global visibility order of all Release Stores, with the exception that each processor may see (via ordinary or acquire loads) its own updates earlier than when other processors see it. Further, this global visibility order is a total order consistent with program order of all release stores in each processor.

Causality: When a *st.rel* instruction X in some processor P1 is read by an *ld.acq* instruction Y in another processor P2, then no store instruction following Y in program order must be visible to any processor before X is visible.

2.1 Examples

The following examples (some from [1]) illustrate the Itanium ordering rules (assume that each memory location has value 0 in the beginning).

• The following execution is *invalid* due to Rule 2 pertaining to WAW, Rule 4 pertaining to Acquire, and the requirement of Coherence,

```
    P              Q
st(A,1)        ld.acq(A,2)
st(A,2)        ld(A,1)
```

• Fence (Rule 3) is illustrated by the following *invalid* execution. Here, `ld(B,0)` and `ld(A,0)` are seen after a fence, while the stores that supply new values into `A` and `B` are not getting flushed as is required by *fence*s:

```
    P              Q
st(A,1)        st(B,1)
Fence          Fence
ld(B,0)        ld(A,0)
```

• Acquire and Release (Rules 4 and 5) are illustrated by the following *invalid* execution. Here, `st(A,1)` precedes `st.rel(B,1)`, while `ld.acq(B)` precedes `ld(A)`. However, we see `ld(A,0)` happening instead of `ld(A,1)`.

```
   P             Q
st(A,1)       ld.acq(B,1)
st.rel(B,1)   ld(A,0)
```

• Coherence is illustrated by the following *invalid* execution. This is because the Acquire semantics forces the `ld` instructions to occur after the `ld.acq` instructions. However, processors `R` and `S` are observing the updates to `A` in different orders:

```
    P            Q          R            S
st(A,1)       st(A,2)   ld.acq(A,1)  ld.acq(A,2)
                        ld(A,2)      ld(A,1)
```

• RC_tso is illustrated by the following *valid* execution. The store of P into A is locally visible to P (say, via a cache or store buffer) before it becomes visible to all other processors (and similarly for Q and variable B): that

```
    P             Q
st.rel(A,1)   st.rel(B,1)
ld.acq(A,1)   ld.acq(B,1)
ld(B,0)       ld(A,0)
```

• Another aspect of release stores is that all the `st.rel` of all the processors taken together forms *a single global visibility order* that is also a total order. Considering this, the following outcome is *not valid*, because `Q` and `R` are observing `st.rel(A,1)` and `st.rel(B,1)` in different orders.

```
     P            Q            R            S
st.rel(A,1)  ld.acq(A,1)  ld.acq(B,1)  st.rel(B,1)
             ld(B,0)      ld(A,0)
```

• The following example violates the causality rule. The `st.rel(A,1)` of `P` is observed by `Q` via a `ld.acq`. Further, `st(B,1)` of `Q` is observed by `ld.acq` of `R`. Causality now requires that `st(B,1)` must be visible to `R` only after `st.rel(A,1)`. However, in this example, `R` sees a different order.

```
     P            Q            R
st.rel(A,1)  ld.acq(A,1)  ld.acq(B,1)
             st(B,1)      ld(A,0)
```

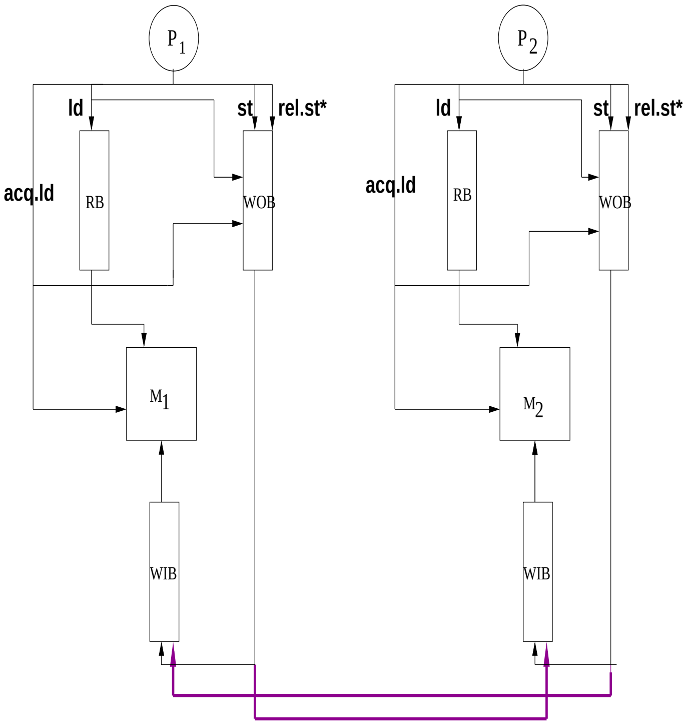

Figure 1. An Operational Model of Itanium

2.2 The Operational Model

Each instruction issued by a processor is modeled by a tuple $t = (p, l, o, a, d, v)$. The field p of tuple t is selected by $p(t)$, and so on. Here,

- $p(t)$ is the processor issuing the instruction.
- $l(t)$ is the ordinal position ("label") of the instruction in the sequential program running on $p(t)$.
- $o(t)$ is the operation type which can be ld, $ld.acq$, st, $st.rel$, or $Fence$.
- $a(t)$ is the memory location to be written into (if $o(t) \in \{st, st.rel\}$), or to be loaded from (if $o(t) \in \{ld, ld.acq\}$).
- $d(t)$ is the data value to be written into a (for stores), or to be loaded from a (for loads).
- $v(t)$ is a vector of labels, whose purpose is to model *causality*, as will be explained shortly.

Some of the fields of a tuple t may be undefined for certain instructions, as will be apparent from the transition system. The operational semantics is now described in terms of five data structures held by each processor p_i (see Figure 1), and how each instruction tuple t that is issued updates these data structures and/or returns the read value, as per Table 1. By 'buffer' we mean an unbounded structure in which the entries maintain their arrival order as in a FIFO, but entries may be removed from anywhere provided a removal condition is satisfied. The oldest entry is always at the head and the youngest at the tail. Initially, all buffers are empty. The data structure elements are:

Event	Guard	Actions
$ld.acq(t)$	$\exists t' \in WOB_{p(t)}$: $a(t') = a(t) \wedge d(t') = d(t)$ $\wedge\, t'$ youngest for address $a(t)$ **else** $M_{p(t)}[a(t)] = d(t)$ $\wedge$ $\neg\exists t' \in WIB_{p(t)}$: $a(t') = a(t) \wedge p(t') = p(t)$	none
$ld(t)$	$\exists t' \in WOB_{p(t)}$: $a(t') = a(t) \wedge d(t') = d(t)$ $\wedge\, t'$ youngest for address $a(t)$ else True	Issue($RB_{p(t)}$, t)
st.rel(t)	True	Issue($WOB_{p(t)}, t$)
$st(t)$	True	$t \leftarrow$ t[$L_{p(t)}/v$]; Issue($WOB_{p(t)}, t$)
$Fence(t)$	True	Flush(t)

$MW(t)$	$t \in WIB_{p(t)}$
	$\wedge$
	Allowed($WIB_{p(t)}, t$)
		$M_{p(t)}[a(t)] \leftarrow d(t)$;
		Delete($WIB_{p(t)}, t$);
		if($o(t) = st.rel$)
		then $L[p(t)] = l(t)$

$MR(t)$	t$\in RB_{p(t)}$ $M_{p(t)}[a(t)] = d(t)$ $\wedge \neg\exists t' \in WIB_{p(t)}$: $a(t') = a(t) \wedge p(t') = p(t)$	Delete($RB_{p(t)}, t$)
$Del(t)$	true	ProcWOB($WOB_{p(t)}, t$)

Table 1. Transition System

1. a memory M_i that spans the entire address-space of Itanium and holds word-sized data in each location. M_i is updated when the $MW(t)$ event of Table 1 fires, which removes an entry from WIB_i writes into M_i. Initially, each location of M_i carries data 0.
2. a write-out buffer WOB_i into which st and $st.rel$ are enqueued. When the $Del(t)$ event of Table 1 fires, an entry is removed from WOB_i and atomically copied into all WIB_j.
3. a load buffer RB_i into which ld instructions (but *not* $ld.acq$) are enqueued when event $ld(t)$ of Table 1 fires. Later, when an $MR(t)$ event fires, a

tuple t is removed from RB_i, and data $d(t)$ corresponding to this tuple gets returned.

4. a write-in buffer WIB_i, and

5. a label-vector L_i held by each processor p_i. This is a vector of natural numbers, with each entry initialized to 0. Specifically, $L_i[j]$ holds the label of the last $st.rel$ instruction of p_j that has already been written into M_i. In other words, $L_i[j]$ indicates the (release-store) instruction of p_j upto which M_i has "caught up." To maintain this invariant, whenever any memory location in M_i gets updated by a release store operation represented by the tuple t, $L_{p(t)}[p(t)]$ gets set to the value $l(t)$, which is the label of the release instruction represented by t. Before an st instruction is enqueued into WOB_i, the $v(t)$ field of this instruction is set to the current L_i value.

2.3 State Transition Rules

Table 1 defines the operational semantics of the Itanium shared memory model. The first column shows *Event*s that happen if the guard condition in the second column is true, performing the actions shown in the third column. At any time, any one of the eligible events may be picked in a *fair* manner. Each event happens when the next instruction t is issued by processor $p(t)$ (for events $ld.acq(t)$ through $Fence(t)$), or when an instruction is removed from one of the internal buffers and is carried out (for events $MW(t)$, $MR(t)$, and $Del(t)$). Notice that in case of events $ld.acq(t)$, $ld(t)$, as well as $MR(t)$, tuple t carries the data $d(t)$ being returned (following the convention used in [8]). When these events fire, a constraint expressed in the Guard field shows what this data is. We use = for equality testing, and ← for assignment.

$\underline{ld.acq(t)}$: If the next instruction tuple t of processor $p(t)$ is a $ld.acq$, we perform the $ld.acq(t)$ event. We seek an entry $t^{'}$ in $WOB_{p(t)}$ such that $a(t^{'}) = a(t)$, and $t^{'}$ is the youngest such entry, if multiple entries exist. If $t^{'}$ exists, the returned data $d(t)$ is the same as $d(t^{'})$. If no such entry exists ("else"), $ld.acq$ must get serviced from the memory $M_{p(t)}$, and that too, only when there is *no* tuple $t^{'}$ in the $WIB_{p(t)}$ buffer such that $p(t^{'}) = p(t)$ and $a(t^{'}) = a(t)$. The condition $p(t^{'}) = p(t)$ prevents a $ld.acq$ from bypassing an earlier issued st or $st.rel$ on the same address.

$\underline{ld(t)}$: As with $ld.acq(t)$, the $ld(t)$ event is serviced directly by $WOB_{p(t)}$ upon a 'hit'; otherwise, t is enqueued into $RB_{p(t)}$ via $Issue(RB_{p(t)}, t)$.

$\underline{st.rel(t)}$: results in t being enqueued into $WOB_{p(t)}$ via procedure Issue.

$\underline{st(t)}$ first updates the v field of tuple t with the label vector $L_{p(t)}$ (shown by t ← t[$L_{p(t)}/v$]), and then enqueues the resulting tuple t into $WOB_{p(t)}$ via procedure Issue.

$\underline{Fence(t)}$ is carried out by procedure Flush, which flushes every pending $RB_{p(t)}$ entry, every $WOB_{p(t)}$ entry, and every WIB_j entry for all j, where the entry comes from $p(t)$ and occurs earlier than t in program order.

$\underline{MW(t)}$ updates the memory array $M_{p(t)}$ from $WIB_{p(t)}$. Its guard 'Allowed' captures when tuple t, which is present in $WIB_{p(t)}$, can be processed ahead of all the other tuples within $WIB_{p(t)}$. This is precisely when there isn't an older $WIB_{p(t)}$ entry $t^{'}$ *and* **one** of the following four conditions hold: (i) $a(t) = a(t^{'})$, (ii) both t and $t^{'}$ are $st.rel$, (iii) both come from $p(t)$ with $o(t) = st.rel$, (iv) the label of $t^{'}$ matches $v(t)[p(t^{'})]$, which is the label of the last $st.rel$ from $p(t^{'})$ seen by $p(t)$, $o(t^{'}) = st.rel$, and $o(t) = st$. Condition (iv) blocks the st from happening until after $M_{p(t)}$ also has assimilated $t^{'}$, ensuring causality. When event $MW(t)$ fires, $M_{p(t)}$ is first updated, and tuple t is then deleted from $WIB_{p(t)}$ by procedure Delete. Also, if the operation of tuple t is $st.rel$, the label-vector $L_{p(t)}$ is updated to the label $v(t)$ carried by tuple t to record the release-store upto which $M_{p(t)}$ has caught up.

$\underline{MR(t)}$ represents when a tuple t buffered in $RB_{p(t)}$ (corresponding to an ld instruction) gets serviced. This event is allowed when memory array $M_{p(t)}$ holds $d(t)$ at address $a(t)$, and there is no $t^{'}$ in $WIB_{p(t)}$ with a matching address from the same processor.

$\underline{Del(t)}$ calls procedure ProcWOB which first checks if $o(t)$=st and there is an entry $t^{'}$ in $RB_{p(t)}$ with address $a(t)$, *or* if $o(t) = st.rel$ and there is an entry $t^{'}$ in either $RB_{p(t)}$ or $WOB_{p(t)}$ with a lower label. If neither, ProcWOB deletes t from WOB, copying it atomically into every WIB. The functions used in the transition system are now described.

Flush(t)**:**

```
WHILE ∨ $(len(WOB\_p(t)) \ > \ 0)$
      ∨ $(len(RB\_p(t)) \ > \ 0)$
      ∨ $(\exists \ i, \ \ t' \in WIB\_i : p(t) = p(t') \ \wedge \ l(t') < l(t))$
DO FOR $t'' \in RB\_p(t)$ DO an MR($t''$) event
   FOR $t'' \in WOB\_p(t)$ DO ProcWOB($WOB\_p(t), t$)
   FOR $t''$ ∈some $WIB\_i$ where $p(t) = p(t'') \wedge \ l(t'') < l(t)$
                                   DO $MW(t'')$
END WHILE
```

ProcWOB($WOB\_p(t), t$)**:**

```
IF $\vee(o(t) = st \wedge \neg\exists \ t' \in \{RB\_p(t), WOB\_p(t)\}:$
                 $a(t) = a(t') \wedge \ l(t') < l(t))$
   $\vee(o(t) = st.rel \wedge \neg\exists \ t' \in \{RB\_p(t), WOB\_p(t)\}:$
                $l(t')<l(t))$
THEN
 Delete $t$ from $WOB\_p(t)$;
 FOR all $i$ DO $Issue(WIB\_i, t)$
END IF
```

Allowed($WIB\_p(t)$,t):

```
$\neg\exists\ t' \in\ WIB\_p(t):$
$t'$ older than $t$
$\wedge$
 $(\vee(a(t) = a(t'))$
  $\vee(o(t) = o(t') = st.rel)$
  $\vee(p(t) = p(t') \wedge o(t) = st.rel)$
  $\vee(l(t') = v(t)[p(t')])$
              $\wedge$
           $(o(t') = st.rel \wedge o(t) = st)$
 )
```

Issue($Buffer, t$): Add t to the tail of Buffer as in a FIFO queue.
Delete($Buffer, t$): Here, Buffer is either RB or WIB. This procedure deletes t wherever it may be in Buffer.

3 Analysis of our Operational Model

We now show how our operational model meets the requirements laid out in Section 2.

1. RAW for load operation r and store operation w earlier in program order: (i) If r is satisfied when it hits a w in WOB (Table 1, event $ld.acq(t)$ or $ld(t)$), RAW is satisfied. (ii) If r is satisfied from memory, in case w has already been written into memory, the RAW hazard is avoided. If however w is in WIB hence blocks r (which is in RB) from issuing, we freeze r till w is written into the memory (Table 1, event $MR(t)$). Thus, here also the RAW hazard is avoided.

2. WAR for load r and store w from the same processor: Note that w cannot move from WOB to WIB until the load is drained from WIB (see ProcWOB). Hence, WAR hazards are avoided.

3. WAW, as well as visibility order for stores to the same location are guaranteed as follows. If there are two stores to the same address in WOB, event $Del(t)$ removes them in the oldest-first order. If the second store comes while the first has gone into WIB, then the "$t^{'}$ issued before t" check in function Allowed prevents a younger write from overtaking an older one.

4. Fence: Procedure Flush carries out all "preceding" instructions before allowing instruction issuing to resume. Hence Rule 3 is obeyed.

5. Acq: Hazard aspects of Acq have already been covered. Since Acq blocks further instruction issuing till it gets carried out (see event $ld.acq(t)$), Rule 4 pertaining to visibility is satisfied.

6. Rel: Loads that come before $st.rel$ are handled by ProcWOB that checks for loads with lower labels. Stores before $st.rel$ are also checked in a similar manner. A $st.rel$ that enters WIB when there is another store in WIB from the same processor is prevented from reordering by function Allowed. This meets Rule 5.

7. Coherence: The rules for handling WOB and WIB ensure Coherence.

8. RC_tso: Handling of WOB and WIB ensure a total global visibility order of release stores. The "TSO" aspect of RC_tso comes naturally because each processor may see its own update early via the WOB, exactly as in classical TSO [5].

All rules except for causality have been discussed. We now discuss causality in some detail. Causality can be summarized at a high level as follows: "Before any st operation o is posted into any M_i, ensure that every $st.rel$ operation r that o is "causally dependent upon" has already been updated into M_i. "Causally dependent on" means o was issued by some p_j after it had updated its own store M_j with the value provided by r.

Causality is obeyed to a certain extent. Specifically, if a $st.rel$ satisfies a $ld.acq$ instruction then all subsequent store operations following that $ld.acq$ instruction in program order will be visible to all processors after that $st.rel$ operation. It suffices to prove this condition by proving that if X is a $st.rel$ from any processor $p(X)$ satisfying Y which is a $ld.acq$ in processor $p(Y)$, Z is a st to any memory address in $p(Z)$ where $Y \rangle\rangle Z$ (hence $p(Y) = p(Z)$), and $X \rightarrow Y$ in $p(Y)$, then $X \rightarrow Z$ for *any* processor p_k. Since $X \rightarrow Y$,

- X must have been updated in M_Y by the time Y is carried out,

- the label vector $v(Z)$ must reflect the update of X, *i.e.*, $v(Z)[p(X)] \geq l(X)$, and

- for any other processor p_k, either X is updated in M_k or else it resides in WIB_k. When Z gets issued to all WIB buffers, and in particular WIB_k, it cannot participate in the $MW(t)$ event before X can do so, due to the behavior of function Allowed.

As an example, consider the earlier discussed example, now with labels:

```
      P               Q               R
1:st.rel(A,1)  1:ld.acq(A,1)  1:ld.acq(B,1)
               2:st(B,1)      2:ld(A,0)
```

The label vector carried by instruction `st(B,1)` would be `[1,0,0]` because `Q` would have seen the `st.rel(A,1)` instruction of `P` situated at label `1` when it issues `st(B,1)`. If `st.rel(A,1)` still resides in the WIB_R buffer when `st(B,1)` also enters WIB_R, function Allowed ensures that the former is posted into M_R before the latter. Thus, `ld(A,0)` is impossible in `R`.

3.1 Ordering Relaxations

We now discuss a few examples of *ordering relaxations* correctly supported by our model.

Releases can be bypassed by subsequent operations. Moreover, these operations may bypass operations preceding release. In the following program,

```
st(a,1)
st.rel(b,1)
st(c,1)
```

`st(c,1)` can bypass both `st.rel(b,1)` and `st(a,1)`. This is supported by our operational model as follows. Suppose these instructions are in WOB. ProcWOB will consider `st(c,1)` as well as `st(a,1)` eligible for movement into WIB, because, for `st` instructions, the label comparisons are done address-wise. However, ProcWOB will *not* be able to move `st.rel(b,1)` into WIB before it moves `st(a,1)`, because for release stores, label comparisons are across all addresses.

Itanium is not required to provide any global total order for `st` instructions. In this example,

```
   P1       P2        P3          P4
st(a,1)  st(b,2)  ld.acq(a,1)  ld.acq(b,2)
                  ld(b,0)      ld(a,0)
```

it allows `P3` to see `st(a,1)` before `st(b,2)` and vice versa in `P4`. This relaxation is supported by function Allowed. Suppose `st(a,1)` and `st(b,2)` are both in WIB_{P3} and WIB_{P4} in some order. Function Allowed can pick `st(a,1)` to post first in M_{P3}, while it can pick `st(b,2)` to post first in M_{P4}.

3.2 How $R\rangle\rangle R$ may impact causality

It is unclear by reading [1] whether the following execution is legal or not:

```
   P1              P2          P3        P4
ld(A,1)       st.rel(A,1)  st(A,2) ld.acq(B,1)
ld.acq(A,2)                        ld(A,0)
st(B,1)
```

If the instructions `ld(A,1)` and `ld.acq(A,2)` are ordered because they are loads on the same location, **then** the following consequences of causality emerge. We have `st.rel(A,1)` being ordered before `ld.acq(A,2)` in the visibility order of `P1`. Due to the acquire semantics, `st(B,1)` is performed after `ld.acq(A,2)`. The situation is quite analogous to the Causality example on Page 2, except the causal chain forms through a load-to-load order. Now, since `st(B,1)` is observed by `ld.acq(B,1)`, we cannot have `ld(A,0)` in `P4` due to causality. It is unknown to us whether load-to-load orderings such as between `ld(A,1)` and `ld.acq(A,2)` are to be obeyed, and if so must cause causal chains in this fashion.

4 Concluding Remarks

In this paper, we provided a simple operational model for Itanium^{TM} shared memory consistency. Our operational model is based on three buffers, a memory array, a label-array, and a collection of non-deterministic rules to process loads, stores, and fences with respect to these data structures. We point out aspects of this memory model, including causality rules. We believe that our model can form a concrete point of discussion for understanding the Itanium^{TM} processor. We also anticipate usage in formal verification, as well as easy adaptation through changes to the rules to other memory models.

References

[1] Intel, *The IA-64 Architecture Software Developer's Manual Vol. 2 rev. 1.1: Itanium (TM); System Architecture*, Intel, 2000, Volume 2, Chapter 13, "Coherence and MP Ordering." http://developer.intel.com/design/ia-64/downloads/24531802.htm.

[2] Gil Neiger, 2001, http://www.cs.utah.edu/mpv/papers/neiger/fmcad2001.pdf.

[3] David L. Dill, Seungjoon Park and Andreas Nowatzyk, "Formal Specification of Abstract Memory Models", in Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pp. 38–52. MIT Press, 1993.

[4] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem and Ganesh Gopalakrishnan, "The 'Test Model-Checking' Approach to the Verification of Formal Memory Models of Multiprocessors", in Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pp. 464–476, Vancouver, BC, Canada, June 1998, Springer-Verlag.

[5] David L. Weaver and Tom Germond, *The SPARC Architecture Manual – Version 9*, P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.

[6] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli and Phillip W. Hutto, "Causal Memory: Definitions, Implementation and Programming", *Distributed Computing*, vol. 9, n. 1, pp. 37–49, 1995.

[7] Sarita V. Adve and Kourosh Gharachorloo, "Shared memory consistency models: A tutorial", *Computer*, vol. 29, n. 12, pp. 66–76, December 1996.

[8] Rob Gerth, "Sequential Consistency and the Lazy Caching Algorithm", *Distributed Computing*, vol. ?, n. 12, pp. 57–59, 1999.

A Details of function Allowed

Why $l(t^{'}) = v(t)[p(t^{'})]$ and not $l(t^{'}) \leq v(t)[p(t^{'})]$ is used in function Allowed: Suppose $l(t^{'}) < v(t)[p(t^{'})]$. Then the value L $= v(t)[p(t^{'})]$ corresponds to some instruction, say $t^{''}$. There are two cases: (i) $t^{''}$ is in $WIB_p(t)$. In this case, function Allowed ensures that $t^{'}$ gets posted into $M_{p(t)}$ before $t^{''}$. Then, we will be back to the = test. (ii) $t^{''}$ has already posted into M, in which case it isn't in $WIB_{p(t)}$. This is a contradiction because $t^{'}$ is still in WIB, violating Allowed.