Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 174 (2007) 63–84
www.elsevier.com/locate/entcs

What is Itanium Memory Consistency from the Programmer's Point of View?

Lisa Higham[1]

Department of Computer Science
The University of Calgary
Calgary, Canada

LillAnne Jackson[2]

Department of Computer Science
The University of Victoria
Victoria, Canada

Jalal Kawash[3]

Department of Computer Science
American Univ. of Sharjah
Sharjah, UAE

Abstract

A programmer-centric model describes the memory consistency rules of amultiprocessor as a collection, one for each processor, of 'views' of instructions and some agreements between these views. It also requires the natural notion of validity: the value read from a shared memory location is the one that was most recently stored, according to a given view. This allows reasoning about programs at a non-operational level in the natural way, not obscured by the implementation details of the underlying architecture. In this paper, we formulate a programmer-centric description of the memory consistency model provided by the Itanium architecture. However, our definition is not tight. We provide two very similar definitions and show that the specification of the Itanium memory model lies between the two. These two definitions are motivated by slightly different implementations of load-acquire instructions. A further entertainment of a handful of other load-acquire rules leads us to question whether the specification of the Itanium memory order [9] is indeed faithful to the Itanium architecture intentions.

Keywords: Programmer-centric memory consistency, itanium multiprocessor.

[1] Email: higham@cpsc.ucalgary.ca
[2] Email: lillanne@cs.uvic.ca
[3] Email: jkawash@aus.edu

doi:10.1016/j.entcs.2007.04.007

1 Introduction

Modern multiprocessor systems include a variety of hardware components such as write-buffers, caches, distributed memory and multiple buses all intricately interconnected. As a consequence, the way that information flows in these various architectures differs widely. To program such systems correctly while exploiting the potential efficiencies available because of these components, it is crucial to have a thorough understanding of this information flow. The rules that describe this flow of data for a particular architecture is called the *memory consistency model* of that architecture. These rules are presented in the system architecture manuals (and other sources) using many different descriptive (in)formalisms. This motivates us to respecify memory consistency models using a common framework. Such a framework helps us compare different systems, port code between systems, and transfer our expertise in one system to facility in another. One framework often used to specify a memory consistency model for an architecture $\mathcal{A}$ is to describe each *instruction* by a program of lower level *operations* acting on the components of $\mathcal{A}$. An *execution* is a sequence of all the operations of all the instructions executed by each processor. Then,

- rules are added that restrict the order in which these lower level operations can occur in any execution, and
- a notion of *validity* is defined, which constrains what values can legitimately be associated with each operation.

An execution must satisfy all the ordering rules and the validity condition in order to be a possible execution on $\mathcal{A}$.

For example, this style has been used to describe the memory semantics of Sparc machines [12]. To illustrate, in this style, a $\mathrm{ld}_p(x)$ (load x by p instruction) could be specified to be the program of operations that first checks the local write buffer of p for a value for x; if it is there it returns the most recently stored such value, otherwise it returns the value of x from main memory. Rules would include one that required that this program of operations must be executed in order, but that the read operation of main memory could be delayed while other operations intervened. The validity condition would require that $\mathrm{ld}_p(x)$ could return without consulting main memory, if and only if there was a value for x in p's write-buffer just before this load was invoked. The Intel Itanium architecture is similarly specified [9,10] but in this case the specification does not prescribe any particular implementing architecture.

We contend, however, that for programming purposes, a memory consistency model should be specified as a set of (ordering) rules on the *instructions* used by the programmer, rather than on a lower level collection of *operations*. Furthermore, the validity condition should be the natural notion of validity of sequences of these instructions acting on the objects of the system. For example, in a valid sequence of loads and stores, the value returned by each load instruction should be the value written by the most recent preceding instruction in the sequence that stored a value to the same memory location. Such a description is useful to a programmer

of the system since she can reason about her code directly, and therefore we call it *programmer-centric*. Descriptions in terms of lower level operations specify an implementation (in hardware or on a virtual platform) and are useful for an architect who is building the system, but should not be confused with its specification. In this case these lower level implementations should be *proved* equivalent to the specification. A further advantage of our approach is that constructions can be composed. A high level specification of an object oriented system can be implemented by a succession of constructions, such that an implementation at one level is the specification for a still lower level, and each level of implementation is proved to correctly implement its specification. This, of course, is the familiar notion of abstraction; we simply extend it to weak models of memory consistency.

In previous work [7,6] we have established a framework for specifying programmer-centric memory consistency models and for proving such equivalences between specifications and implementations. We have applied this framework and the proof techniques to an extensive example involving write buffer architecture [6]. This paper applies these ideas to the Intel Itanium architecture. That is, we aim for a programmer-centric specification of the memory consistency of the Itanium multiprocessor. As will be seen, we failed to realize this goal. Instead, we define two very similar programmer-centric memory consistency models, $Itanium_w$ and $Itanium_s$, and show that "official" Itanium memory consistency [9], henceforth referred to as Itanium (with no subscript), lies strictly between these two (Section 4). $Itanium_w$ and $Itanium_s$ differ only slightly in the ordering constraints involving Itanium load-acquire instructions, and each is motivated by a plausible hardware implementation. We show that several other plausible definitions also fail to exactly capture the Itanium memory consistency specification (Section 5). Furthermore, we know of no lower level description that has a plausible hardware implementation and is equivalent to Itanium. For example, we have been able to use our techniques to show that the machine proposed by Chatterjee and Gopalakrishnan [3] does not exactly implement Itanium (Section 6). We are even led to speculate whether the specification of the Itanium memory consistency [9] is really what the Itanium architects intended!

Section 2 highlights the architectural features of Itanium. Before we present our main results, we briefly describe the model in Subsection 3.1. Subsection 3.2 defines $Itanium_w$ and $Itanium_s$. In Section 4 we argue that the Itanium specification [9] is weaker than $Itanium_s$ and stronger than $Itanium_w$. The formal proofs are elsewhere [8]. In Section 5, we consider several other potential programmer-centric definitions, and show each of them also fails to exactly capture Itanium consistency. The rest of this paper requires familiarity with the Itanium memory consistency model [9]. Those definitions that are essential for this work are reproduced in Subsection 3.3.

Several other frameworks for describing memory consistency have been proposed but are not central to this paper. The framework of Adir, Attiya and Shurek [1] is very similar to ours and precedes ours. Arvind and Maessem [2] provide a framework for serializable memory models. We are unaware, however, of how to use these

frameworks to prove equivalence between systems. Yang *et. al.* [13,14,4] present a non-operational approach to specifying and analyzing shared memory consistency models and use it to provide a translation of the rules of Itanium specification. The TLA work of Joshi *et. al.* [11] is a precise specification of Itanium and is the basis of the official specification [9].

2 Itanium Architecture Highlights

The Itanium specifications [9] are independent of specific machine implementations. While we do not know of a concrete machine implementation that exactly captures the Itanium specifications, this section gives an overview of the architectural features of such a machine.

Itanium provides a distributed-shared memory (DSM) architecture where each processor maintains a replicated copy of the shared address space. For this paper the shared memory consists of only shared variables. That is, instructions either load or store a shared variable. Loads are satisfied by returning the value in the local replica, without any communication with the other processors. However, stores are performed by updating the local replica and broadcasting the update to every other processor to apply it in their replicas. Stores by a processor are visible to that processor before they can be visible to other processors. The channels between processors are not necessarily FIFO when two instructions are applied to different variables. The rules that govern processor execution and inter-processor interaction are complicated and give rise to complex behaviors.

Itanium also supports write-buffers with read by-passing. A store is buffered before it is committed to the replicas in order to hide store latencies. A load by a processor checks if the local write-buffer contains a store that can satisfy the load (both the load and the store are on the same variable). If this is the case, the value of the most recent such store is returned, without the need to check the local replica. In the case of a buffer miss, the load bypasses the pending stores in the buffer and loads the value from the local replica. Write-buffering further complicates the behavior of Itanium. Bypassing loads can complete before earlier buffered stores and give rise to an out-of-order execution. From the programmer's point of view, a *foreign* load returns a value from the local replica, rather than from the local buffer. This is guaranteed, for instance, when a processor loads a variable that it never stores, such as a a single-writer variable owned by a different processor. The write buffers in Itanium are guaranteed to be FIFO only per variable. Hence, two store instructions to different variables can be applied to a replica in the opposite order in which they occur in a processor's program.

In order to restrict out-of-order execution, Itanium supports the extensions of "acquire" and "release" to load and store instructions, respectively. A load-acquire instruction is required to be always performed before any subsequent instruction in the program. A store-release is required to be always performed after every preceding instruction in the program. Typically a program with a critical section performs an acquire before entering the critical section, ensuring that the critical

section is delayed until the proper lock is obtained. Also, it performs a release after exiting the critical section ensuring that the lock is not released earlier than the actual exit from the critical section. Itanium also provides fence instructions and semaphore read-modify-write instructions which combine both acquire and release requirements.

Acquires and releases restrict the write buffer's behavior. For instance, when a release is buffered it forces all previously buffered stores to be removed from the buffer and applied to the local replicas before the release itself. Depending on the implementation, a load-acquire can force the buffer to be flushed, but this is not necessary in general. They also restrict the inter-processor interaction behavior. Incoming store-releases restrict how processors apply the other incoming stores. Typically, just as a release forces a preceeding store to be removed from the buffer earlier, these stores have to be applied remotely earlier than the release.

From the programmer's point of view, an Itanium execution is a collection of "views", one for each processor. Due to DSM nature, a processor's view consists of its own loads and all processors' stores. The view allows out-of-order execution and there has to be a minimum level of agreement between these views. The Itanium specifications [9] are summarized at the end next of section, in which we formulate these Itanium views.

3 Multiprocesses, Computations and Memory Consistency

3.1 Instructions, multiprocessors and computations

As each process in a multiprocess system executes, it issues a sequence of instruction invocations on shared memory objects. [4] For this paper the shared memory consists of only shared variables, and each *instruction invocation* is *Itanium-based*. That is, each instruction invocation is of the form $\mathrm{st}_p(x, v)$ or $\mathrm{st.rel}_p(x, v)$ meaning that process p writes a value v to the shared variable x, or $\mathrm{ld}_p(x)$ or $\mathrm{ld.acq}_p(x)$ meaning that process p reads a value from shared variable x or of fence_p meaning that process p invoked a memory fence instruction. Instruction invocations st and st.rel are referred to collectively as *store* instructions and have *store semantics*; ld and ld.acq are called *load* instruction invocations and have *load semantics*. It suffices (for this paper) to model each individual process p as a sequence of these instruction invocations and call such a sequence an *individual (Itanium-based) program.* [5] An *(Itanium-based) multiprogram* is a finite set of these individual programs.

An *instruction* is an instruction invocation completed with a response. In our setting the response of a store instruction invocation or a fence instruction invocation is an acknowledgment and is ignored. The response of a load invocation is the value returned by the invocation. A *(multiprocess) computation of an Itanium-based*

[4] Parts of this section were first used in previous work (Section 2.2 of [5]); they are re-used in this work in a modified form.

[5] We have made common some simplifying assumptions such as memory locations do not overlap, memory is cacheable (i.e., WB) and semaphores are omitted.

multiprogram, P is created from P by changing each load instruction invocation, $\mathrm{ld}_p(x)$ (respectively, $\mathrm{ld.acq}_p(x)$) to $\nu \leftarrow \mathrm{ld}_p(x)$ (respectively, $\nu \leftarrow \mathrm{ld.acq}_p(x)$) where ν is either the initial value of x or some value stored to x by some store to x in the multiprogram.

Notice that the definition of a computation permits the value returned by each $\mathrm{ld}(x)$ or $\mathrm{ld.acq}(x)$ instruction invocation to be arbitrarily chosen from the set of values stored to x by the multiprogram. In an Itanium machine (or any other multiprocessor), the values that might actually be returned are substantially further constrained by its architecture, which determines the way in which the processes communicate and that shared memory is implemented. A *memory consistency model* captures these constraints by specifying a set of additional requirements that computations must satisfy. Typically, these require the existence of a set of sequences of instructions that satisfy certain properties. A collection of such sequences that meet all the requirements is called a set of *verifying sequences.* We use $\mathcal{C}(P, \mathcal{MC})$ to denote the set of all computations of multiprogram P that satisfy the memory consistency model MC. Memory consistency model MC is *stronger than* MC' if, for every Itanium-based Multiprogram P, $\mathcal{C}(P, \mathcal{MC}) \subseteq \mathcal{C}(P, \mathcal{MC}')$. MC is *strictly stronger than* MC' if for every Itanium-based Multiprogram P, $\mathcal{C}(P, \mathcal{MC}) \subsetneq \mathcal{C}(P, \mathcal{MC}')$ The terms *weaker* and *strictly weaker* are defined similarly.

The description of a memory consistency model is simplified by assuming that each store instruction invocation has a distinct value. Although it is technically straightforward to remove this assumption, without it, the description of the memory model is messy and its properties are consequently obscured.

For an Itanium-based computation C, $I(C)$ denotes all the instructions in C. $I(C)|p$ is the subset of $I(C)$ in processor p's program sequence; $I(C)|x$ is the subset of $I(C)$ applied to variable x; $I(C)|r$ is the subset containing only the load instructions; $I(C)|w$ is the subset containing only the store instructions; Let $I(C)|acq$ denote the subset containing all ld.acq instructions plus the memory fence instructions; let $I(C)|rel$ denote the subset containing all st.rel instructions plus the memory fence instructions. The relation $(I(C), \xrightarrow{prog})$, called *program order*, is the set of all pairs (i, j) of instructions that are in the same individual computation of C and such that i precedes j in that sequence. For any partial order relation $(I(C), \xrightarrow{y})$, the notation $i \xrightarrow{y} j$ is used to mean of $(i, j) \in (I(C), \xrightarrow{y})$.

A load instruction is *domestic* if the value it returns was stored into shared memory location x by a store instruction by the same processor; memory fence instructions and load instructions that are not domestic are *foreign.* If an instruction, i, with load semantics returns the value stored by an instruction, j, with store semantics then i and j are *causally related.*

3.2 Weak and strong Itanium memory consistency

This section formulates two programmer-centric definitions of Itanium consistency. They differ only in the way a ld.acq is implemented.

Define the following partial orders:

Weak Orderable Order: $(I(C)|p \cup I(C)|w, \stackrel{word_p}{\longrightarrow})$ for each $p \in P$: $i \stackrel{word_p}{\longrightarrow} j$ if $i, j \in I(C)|p \cup I(C)|w$ and $i \stackrel{prog}{\longrightarrow} j$ and one of the following:
Weak Acquire: $i \in I(C)|acq$ and is foreign, or
Release: $j \in I(C)|rel$, or
Same Memory: $i, j \in I(C)|x$ and $[(i \in I(C)|w$ or $j \in I(C)|w)$ or $(i \in I(C)|acq)]$

Strong Orderable Order: $(I(C)|p \cup I(C)|w, \stackrel{sord_p}{\longrightarrow})$ for each $p \in P$: $i \stackrel{sord_p}{\longrightarrow} j$ if $i, j \in I(C)|p \cup I(C)|w$ and $i \stackrel{prog}{\longrightarrow} j$ and one of the following:
Strong Acquire: $i \in I(C)|acq$, or
Release: $j \in I(C)|rel$, or
Same Memory (Simpler): $i, j \in I(C)|x$ and $[i \in I(C)|w$ or $j \in I(C)|w\ ]$

The Strong Orderable Order requires a "text-book" or conservative implementation of ld.acq instructions. That is, it requires the ld.acq to precede any instruction that follows it in the program. In the presence of buffers, certain architectural decisions can sacrifice this "text-book" behavior. For instance Weak Orderable Order captures the situation when a ld.acq can be satisfied from the buffer (a domestic ld.acq). A following (in program order) ld can by-pass the buffer. Or, a following st to a different variable can be committed to the local replica earlier than the buffered st that is used to satisfy the ld.acq. In this case, the order between ld.acq and the subsequent ld or st can no longer be guaranteed. There is one occurrence of each st in a processor's view, and these views are constructed based on the order in which stores occur in the local replicas. To maintain the intuitive notion of validity, the ld.acq must be delayed in the view until its causally-related st occurs in the local replica. Hence, a domestic ld.acq may occur in a view after a ld or a st that follows it in program order. Weak Orderable Order allows this behavior, but prohibits it when the ld.acq is foreign (necessarily satisfied from the local replica rather than the buffer). The Same Memory condition prohibits this behavior when ld.acq and the ld are applied to the same variable: if the ld.acq is satisfied from the buffer, then either the ld is also satisfied from the buffer or if not the st under consideration must have been applied to the local replica.

One mechanism to prohibit a domestic ld.acq to occur in a processor's view later than it should be is to flush the buffer before the ld.acq is completed, ensuring that the ld.acq is always satisfied from the local replica. Such an architecture could achieve views satisfying Strong Orderable Order.

The Release condition is simply what a programmer expects: any instruction preceding a st.rel must maintain this order in the processors' views. The following defines these views.

Definition 3.1 A computation C satisfies *Weak Itanium consistency*, denoted $Itanium_w$, if for each $p \in P$ there is a sequence S_p of the instructions $I(C)|p \cup I(C)|w$ that is valid for p, such that:

(i) If $i, j \in I(C)|p \cup I(C)|w$ and $i \stackrel{word_p}{\longrightarrow} j$ then $i \stackrel{S_p}{\longrightarrow} j$, (Orderable requirement) and

(ii) If $i, j \in I(C)|x|w$ and $i \xrightarrow{S_p} j$ then $i \xrightarrow{S_q} j$, $\forall q \in P$, (Same Memory agreement) and

(iii) If $i, j \in I(C)|rel$ and $i \xrightarrow{S_p} j$ then $i \xrightarrow{S_q} j$, $\forall q \in P$, (Release agreement) and

(iv) If $i \in I(C)|rel$ and $j \in I(C)|st|p$ and $i \xrightarrow{S_p} j$ then $i \xrightarrow{S_q} j$, $\forall q \in P$, (Release to Store agreement) and

(v) There does not exist a cycle of $i_1, i_2 \ldots i_k \in I(C)|w$ where $i_j \in I(C)|p_j, \forall j \in \{1, 2, \ldots k\}$ and $k \leq n$ such that: $i_k \xrightarrow{S_1} i_1$, and $i_1 \xrightarrow{S_2} i_2$, and $i_2 \xrightarrow{S_3} i_3 \ldots$ and $i_{k-1} \xrightarrow{S_k} i_k$ (Cycle Free agreement)

Definition 3.2 A computation satisfies *Strong Itanium consistency*, denoted $Itanium_s$, if it satisfies all the conditions of $Itanium_w$, but with Weak Orderable order replaced by Strong Orderable order(in item 1. of Definition 3.1 above).

Hence, a view of a processor consists of its own instructions in addition to the store instructions of all other processors. Each view maintains the required Orderable Order (item 1). The remaining items are "agreement" requirements, establishing required relationships between the different views. Since channels between processors are FIFO for each variable, the communicated store instructions to the same variable must appear in every view in the same order (item 2). A st.rel instruction occurs in all replicas atomically. Hence item 3 requires the st.rel instructions to be seen in the same order by all processors and item 4 enforces that for any st seen by its processor after a st.rel, that st must be seen in the same way by all processors. Item 5 is a technical condition arising from timing considerations. Consider a store s_p by p and a store s_q by q. Since a store is visible to the storing processor before it is visible to others, it is not possible for p see s_q before s_p, and yet for q see s_p before s_q. Item 5 generalizes this to any number of processors.

3.3 Itanium memory consistency according to the Itanium manual

The proofs that $Itanium_w$ and $Itanium_s$ bound the definition of Itanium are elsewhere [8] and they make extensive reference to the Intel manual [9]. We still reference the specifications of the Intel manual in this paper, particularly when we argue if a given computation satisfies Itanium or otherwise. Recall that Itanium (without a subscript) refers to the system specified in this manual [9]. For completeness, the definitions that we require are paraphrased from this manual next. When the same things are named differently in the manual [9] and in our framework (Section 3), we maintain our terminology and notation. For example, what we call a computation is exactly what the manual calls an *execution*, and we denote program order by $\xrightarrow{prog}$ whereas the manual uses $\gg$. We also define a few additional terms to simplify notation. The symbol st[.rel] represents a store instruction (i.e. either st or st.rel), ld[.acq] represents a load instruction (i.e. either ld or ld.acq), and i represents any Itanium-based instruction.

Each Itanium-based instruction is decomposed into *operations* that either read values from or write values to memory locations. An instruction's operations corre-

spond to different aspects of the visibility of the instruction for different processors. Specifically, ld[.acq] is "decomposed" into a single read operation R(ld[.acq]). st[.rel] by processor p is decomposed into $n+1$ write operations for an n-processor multiprocessor: a local write operation visible only to p denoted LV(st[.rel]) and a remote write operation for each processor q in the system denoted RV_q(st[.rel]). fence is "decomposed" into just one operation, F(fence). The operations of an instruction and the instruction itself *correspond*. For example, each of the operations $\mathrm{LV}(\mathrm{st}_p(x,v))$, $\mathrm{RV}_p(\mathrm{st}_p(x,v))$ and $\mathrm{RV}_q(\mathrm{st}_p(x,v))$ for every processor $q \neq p$ corresponds to the store instruction $\mathrm{st}_p(x,v)$. The operation `O` is a *read operation* (respectively, *write operation*) if `O` corresponds to load (respectively, store) instruction.

We assume that memory locations with distinct names do not overlap. Let `WR` be a (write) operation corresponding to instruction st[.rel] and `RD` be a (read) operation corresponding to instruction ld[.acq]. The value stored by st[.rel] (respectively, written by `WR`) is denoted WrVal(st[.rel]) (respectively, WrVal(`WR`)). Similarly, the value loaded by ld[.acq] (respectively, read by `RD`) is denoted RdVal(ld[.acq]) (respectively, RdVal(`RD`)). Every location b in memory has an *initial value*, denoted by InitVal(b), that will be returned to read operations when they occur before there are any write operations to that location.

Any computation of the basic Itanium processor family memory ordering model must have an associated *visibility order* which linearly orders all the operations that correspond to all the instructions of the computation and satisfies the *Itanium rules* below. (If there is no visibility order for a computation that satisfies all of these rules, the computation is not permitted by the architecture.)

If an instruction i is by a processor p, we write $p = \mathrm{Proc}(i)$. For any two operations `O` and `U`, $\mathtt{O} \xrightarrow{V} \mathtt{U}$ means that `O` precedes `U` in the visibility order V. If there is a store instruction $\mathrm{st}_p(x,\cdot)$ and a load instruction $\mathrm{ld}_p(x)$ such that $\mathrm{LV}(\mathrm{st}_p(x,\cdot)) \xrightarrow{V} \mathrm{R}(\mathrm{ld}_p(x)) \xrightarrow{V} \mathrm{RV}_p(\mathrm{st}_p(x,\cdot))$ then the operation $\mathrm{R}(\mathrm{ld}_p(x))$ is a *local read in V* and $\mathrm{ld}_p(x)$ is a *local load in V* (or simply a local load or local read when V is clear).

Itanium rules

Write Operation Order

(WO): No store can become visible remotely before it becomes visible locally.
For every store st[.rel] where p=proc(st[.rel]),
$\mathrm{LV}(\mathrm{st}[.\mathrm{rel}]) \xrightarrow{V} \mathrm{RV}_p(\mathrm{st}[.\mathrm{rel}])$ and
$\mathrm{RV}_p(\mathrm{st}[.\mathrm{rel}]) \xrightarrow{V} \mathrm{RV}_q(\mathrm{st}[.\mathrm{rel}])$ for $q \neq$Proc(st[.rel]).

Program Order

(ACQ): No instruction can become visible before a preceding ld.acq.
If ld.acq $\xrightarrow{prog} i$, `A` is a read operation corresponding to ld.acq, and `O` is an operation corresponding to i, then $\mathtt{A} \xrightarrow{V} \mathtt{O}$.

(REL): No st.rel can become visible before a preceding instruction.

- If $i \xrightarrow{prog}$ st.rel, and i is not a store instruction, and `O` is an operation correspond-

ing to i, then
$\mathtt{O} \xrightarrow{V} \mathrm{LV}(\mathrm{st.rel})$.

- If $\mathrm{st[.rel]} \xrightarrow{prog} \mathrm{st.rel}$ where st[.rel] is a store instruction, then $\mathrm{LV}(\mathrm{st[.rel]}) \xrightarrow{V} \mathrm{LV}(\mathrm{st.rel})$ and
$\mathrm{RV}_p(\mathrm{st[.rel]}) \xrightarrow{V} \mathrm{RV}_p(\mathrm{st.rel})$ for each processor p.

(FEN):[6] Instructions become visible in order with respect to fence instructions.

- If $\mathrm{fence} \xrightarrow{prog} i$ and O is an operation corresponding to i, then $\mathrm{F(fence)} \xrightarrow{V} \mathtt{O}$.
- If $i \xrightarrow{prog} \mathrm{fence}$ and O is an operation corresponding to i, then $\mathtt{O} \xrightarrow{V} \mathrm{F(fence)}$.

Memory-Data Dependence

(MD:RAW): No load may become visible before an earlier store to a common location.

- If st[.rel] and ld[.acq] access the same memory location and $\mathrm{st[.rel]} \xrightarrow{prog} \mathrm{ld[.acq]}$, then
$\mathrm{LV}(\mathrm{st[.rel]}) \xrightarrow{V} \mathrm{R}(\mathrm{ld[.acq]})$.

(MD:WAR): No store may become visible locally before an earlier load to a common location.

- If ld[.acq] and st[.rel] access the same memory location and $\mathrm{ld[.acq]} \xrightarrow{prog} \mathrm{st[.rel]}$, then
$\mathrm{R}(\mathrm{ld[.acq]}) \xrightarrow{V} \mathrm{LV}(\mathrm{st[.rel]})$.

(MD:WAW): Stores by a processor to a common location become visible to that processor in program order.

- If $\mathrm{st[.rel]}_1$ and $\mathrm{st[.rel]}_2$ access the same memory location and $\mathrm{st[.rel]}_1 \xrightarrow{prog} \mathrm{st[.rel]}_2$, then
$\mathrm{LV}(\mathrm{st[.rel]}_1) \xrightarrow{V} \mathrm{LV}(\mathrm{st[.rel]}_2)$.

Coherence

(COH): Stores to the same location become remotely visible in the same order for every processor.

- If $\mathrm{st[.rel]}_1$ and $\mathrm{st[.rel]}_2$ are stores to the same location and $\mathrm{Proc}(\mathrm{st[.rel]}_1)=\mathrm{Proc}(\mathrm{st[.rel]}_2)$ and
$\mathrm{LV}(\mathrm{st[.rel]}_1) \xrightarrow{V} \mathrm{LV}(\mathrm{st[.rel]}_2)$ then
$\mathrm{RV}_p(\mathrm{st[.rel]}_1) \xrightarrow{V} \mathrm{RV}_p(\mathrm{st[.rel]}_2)$.
- If $\mathrm{st[.rel]}_1$ and $\mathrm{st[.rel]}_2$ are stores to the same location and $\mathrm{RV}_p(\mathrm{st[.rel]}_1) \xrightarrow{V} \mathrm{RV}_p(\mathrm{st[.rel]}_2)$ for any processor p, then $\mathrm{RV}_q(\mathrm{st[.rel]}_1) \xrightarrow{V} \mathrm{RV}_q(\mathrm{st[.rel]}_2)$ for all processors q.

Store-release

(WBR): Store-release instructions become remotely visible atomically.

- If $\mathrm{RV}_p(\mathrm{st.rel}) \xrightarrow{V} \mathtt{O} \xrightarrow{V} \mathrm{RV}_q(\mathrm{st.rel})$ then
$\mathtt{O}=\mathrm{RV}_r(\mathrm{st.rel})$ for some processor r.

Read Value

(RV1): Let ld[.acq] be a local load of location x and st[.rel] be a store to x, such that Proc(st[.rel]) = Proc(ld[.acq]). Suppose that LV(st[.rel]) $\xrightarrow{V}$ R(ld[.acq]) and there is no other store,st[.rel]' to x with
Proc(st[.rel]')= Proc(ld[.acq]) where
LV(st[.rel]) $\xrightarrow{V}$ LV(st[.rel]') $\xrightarrow{V}$ R(ld[.acq]). Then
RdVal(ld[.acq]) = WrVal(st[.rel]).

(RV2): Let ld[.acq] be a non-local load of location x and p = Proc(ld[.acq]). Suppose there is a store st[.rel] to x such that RV_p(st[.rel]) $\xrightarrow{V}$ R(ld[.acq]), and there is no other store st[.rel]' to x with
RV_p(st[.rel]) $\xrightarrow{V}$ RV_p(st[.rel]') $\xrightarrow{V}$ R(ld[.acq]). Then RdVal(ld[.acq]) = WrVal(st[.rel]).

(RV3): Let ld[.acq] be a non-local load instruction of location x and p = Proc(ld[.acq]). Suppose there is no store st[.rel] to x such that RV_p(st[.rel]) $\xrightarrow{V}$ R(ld[.acq]). Then RdVal(ld[.acq]) = InitVal(x).

4 Itanium is strictly between Itanium$_w$ and Itanium$_s$

The proofs of our two major theorems, Theorems 4.1 and 4.2 are elsewhere [8].

Theorem 4.1 *Itanium memory consistency is strictly stronger than Itanium$_w$ memory consistency.*

Theorem 4.2 *Itanium$_s$ memory consistency is strictly stronger than Itanium memory consistency.*

Here we focus on a compuation that captures the essential difference between *Itanium$_w$* and *Itanium*.

Comp 1 $\begin{cases} p: 3 \leftarrow \mathrm{ld}(x)\ \ \mathrm{st}(x,2)\ \ 2 \leftarrow \mathrm{ld.acq}(x)\ \ \mathrm{st}(y,4) \\ q: 4 \leftarrow \mathrm{ld.acq}(y)\ \ \mathrm{st}(x,3) \end{cases}$

In Computation 1, the $2 \leftarrow \mathrm{ld.acq}_p(x)$ instruction is domestic while the $4 \leftarrow \mathrm{ld.acq}_q(y)$ instruction is foreign. $4 \leftarrow \mathrm{ld.acq}_q(y)$ must be satisfied from the local replica and not the write-buffer, but it is possible for $2 \leftarrow \mathrm{ld.acq}_p(x)$ to be satisfied from p's write-buffer while $\mathrm{st}_p(x,2)$ is pending, waiting to be applied to p's local replica. Since the write-buffers are only FIFO per variable, it is possible for $\mathrm{st}_p(y,4)$ to be applied to p's replica before $\mathrm{st}_p(x,2)$. Hence, in p's view it is possible for $2 \leftarrow \mathrm{ld.acq}_p(x)$ to occur after $\mathrm{st}_p(y,4)$, a violation of the "text-book" implementation of ld.acq. *Itanium$_w$* allows this behavior, which is captured by the following verifying sequences:

$\begin{cases} S_p: \mathrm{st}_p(y,4)\ \ \mathrm{st}_q(x,3)\ \ 3 \leftarrow \mathrm{ld}_p(x)\ \ \mathrm{st}_p(x,2)\ \ 2 \leftarrow \mathrm{ld.acq}_p(x) \\ S_q: \mathrm{st}_p(y,4)\ \ 4 \leftarrow \mathrm{ld.acq}_q(y)\ \ \mathrm{st}_q(x,3)\ \ \mathrm{st}_p(x,2) \end{cases}$

Computation 1 does not satisfy *Itanium* because to the following cycle of operations:

$\mathrm{R}(3 \leftarrow \mathrm{ld}_p(x)) \overset{MD:WAR}{\longrightarrow} \mathrm{LV}(\mathrm{st}_p(x,2)) \overset{(MD:RAW)}{\longrightarrow}$
$\mathrm{R}(2 \leftarrow \mathrm{ld.acq}_p(x)) \overset{(ACQ)}{\longrightarrow} \mathrm{LV}(\mathrm{st}_p(y,4)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_p(\mathrm{st}_p(y,4)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_q(\mathrm{st}_p(y,4))$
$\overset{(RV2)}{\longrightarrow} \mathrm{R}(4 \leftarrow \mathrm{ld.acq}_q(y)) \overset{(ACQ)}{\longrightarrow}$
$\mathrm{LV}(\mathrm{st}_q(x,3)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_q(\mathrm{st}_q(x,3)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_p(\mathrm{st}_q(x,3)) \overset{(RV2)}{\longrightarrow} \mathrm{R}(3 \leftarrow \mathrm{ld}_p(x))$.
Any verifying visibility sequence is a total order, so no such sequence could extend the orders of this cycle.

Also, Computation 1 does not satisfy $Itanium_s$, which requires $2 \leftarrow \mathrm{ld.acq}_p(x)$ to precede $\mathrm{st}_p(y,4)$ in p's view. However, this is not possible because S_p must extend: $\mathrm{st}_q(x,3) \overset{valid}{\longrightarrow} 3 \leftarrow \mathrm{ld}_p(x) \overset{same\ memory}{\longrightarrow} \mathrm{st}_p(x,2) \overset{same\ memory}{\longrightarrow} 2 \leftarrow \mathrm{ld.acq}_p(x) \overset{strong\ acquire}{\longrightarrow} \mathrm{st}_p(y,4)$.

The Cycle Free agreement requirement needs $\mathrm{st}_q(x,3) \overset{S_q}{\longrightarrow} \mathrm{st}_p(y,4)$ because otherwise $\mathrm{st}_q(x,3) \overset{S_p}{\longrightarrow} \mathrm{st}_p(y,4) \overset{S_q}{\longrightarrow} \mathrm{st}_q(x,3)$ which is not allowed. Thus, S_q contains the following cycle: $\mathrm{st}_q(x,3) \overset{cycle\ free}{\longrightarrow} \mathrm{st}_p(y,4) \overset{valid}{\longrightarrow} 4 \leftarrow \mathrm{ld.acq}_q(y) \overset{strong\ acquire}{\longrightarrow} \mathrm{st}_q(x,3)$.

While the "liberal" behavior of the ld.acq instructions in $Itanium_w$ allows computations that are otherwise prohibited under *Itanium*, the conservative "textbook" behavior of the ld.acq instructions in $Itanium_s$ is too prohibitive.

Computation 2 satisfies *Itanium* consistency but not $Itanium_s$ consistency.

Comp 2
$$\begin{cases} p: 4 \leftarrow \mathrm{ld.acq}(y)\ \ \mathrm{st}(x,5)\ \ \mathrm{st.rel}(z,2) \\ q: \mathrm{st}(x,3)\ \ 3 \leftarrow \mathrm{ld.acq}(x)\ \ \mathrm{st}(y,4) \\ \quad 2 \leftarrow \mathrm{ld.acq}(z)\ \ 3 \leftarrow \mathrm{ld}(x) \end{cases}$$

Processor q can place $\mathrm{st}_q(x,3)$ in its write-buffer, satisfy $3 \leftarrow \mathrm{ld.acq}(x)$ from the buffer, and then buffer $\mathrm{st}_q(y,4)$. Since the write-buffers are only FIFO per variable it is possible for $\mathrm{st}_q(y,4)$ to be applied to both replicas while $\mathrm{st}_q(x,3)$ is still pending in the buffer. Processor p can perform $4 \leftarrow \mathrm{ld.acq}_p(y)$ and then apply $\mathrm{st}_p(x,5)$ to q'a replica while $\mathrm{st}_q(x,3)$ is still in q's buffer.

Formally, a sequence V that satisfies *Itanium* is:
$\mathrm{LV}(\mathrm{st}_q(x,3))$, $\mathrm{R}_q(3 \leftarrow \mathrm{ld.acq}_q(x))$, $\mathrm{LV}(\mathrm{st}_q(y,4))$,
$\mathrm{RV}_q(\mathrm{st}_q(y,4))$, $\mathrm{RV}_p(\mathrm{st}_q(y,4))$, $\mathrm{R}_p(4 \leftarrow \mathrm{ld.acq}_p(y))$,
$\mathrm{LV}(\mathrm{st}_p(x,5))$, $\mathrm{LV}(\mathrm{st.rel}_p(z,2))$, $\mathrm{RV}_p(\mathrm{st}_p(x,5))$, $\mathrm{RV}_q(\mathrm{st}_p(x,5))$, $\mathrm{RV}_p(\mathrm{st.rel}_p(z,2))$,
$\mathrm{RV}_q(\mathrm{st.rel}_p(z,2))$, $\mathrm{R}_q(2 \leftarrow \mathrm{ld.acq}_q(z))$,
$\mathrm{R}_q(3 \leftarrow \mathrm{ld}_q(x))$, $\mathrm{RV}_q(\mathrm{st}_q(x,3))$, $\mathrm{RV}_p(\mathrm{st}_q(x,3))$.

$Itanium_s$ does not allow Computation 2 since $Itanium_s$ requires all ld.acq instructions to be satisfied from the local replica rather than the buffer. Hence, $\mathrm{st}_q(x,3)$ is guaranteed to be applied to q's replica before even $\mathrm{st}_q(y,4)$ is buffered. p must see $\mathrm{st}_q(y,4)$ before it buffers $\mathrm{st}_p(x,5)$ because it sees the value in y through a ld.acq instruction. When p sees $\mathrm{st}_q(y,4)$, the value of x in q's replica must be 3. p's $\mathrm{st.rel}_p(z,2)$ forces $\mathrm{st}_p(x,5)$ to be applied everywhere before the st.rel itself. When q sees $\mathrm{st.rel}_p(z,2)$, it must also have seen $\mathrm{st}_p(x,5)$. So the value of x in q's replica must be 5, overwriting the earlier value of 3. $3 \leftarrow \mathrm{ld}_q(x)$ must take place

after $2 \leftarrow \mathrm{ld.acq}_q(z)$, since $Itanium_s$ requires the ld.acq to precede any following instruction. However, we have already argued that the value of x according to q cannot be 3.

Formally, the $Itanium_s$ sequence, S_p, must extend: $\mathrm{st}_q(y,4) \xrightarrow{valid} 4 \leftarrow \mathrm{ld.acq}_p(y) \xrightarrow{strong\ acquire} \mathrm{st}_p(x,5) \xrightarrow{release} \mathrm{st.rel}_p(z,2)$. The Cycle Free agreement requirement needs $\mathrm{st}_q(y,4) \xrightarrow{S_q} \mathrm{st}_p(x,5)$ because otherwise $\mathrm{st}_q(y,4) \xrightarrow{S_p} \mathrm{st}_p(x,5) \xrightarrow{S_q} \mathrm{st}_q(y,4)$ which is not allowed. Thus, S_q must extend: $\mathrm{st}_q(x,3) \xrightarrow{same\ memory} 3 \leftarrow \mathrm{ld.acq}_q(x) \xrightarrow{strong\ acquire} \mathrm{st}_q(y,4) \xrightarrow{cycle\ free} \mathrm{st}_p(x,5) \xrightarrow{release} \mathrm{st.rel}_p(z,2) \xrightarrow{valid} 2 \leftarrow \mathrm{ld.acq}_q(z) \xrightarrow{strong\ acquire} 3 \leftarrow \mathrm{ld}_q(x)$. This makes the final $3 \leftarrow \mathrm{ld}_q(x)$ invalid.

5 Comparing Alternative Acquire Orders

$Itanium_w$ and $Itanium_s$ bound $Itanium$ and the only difference between them is slight changes in the Acquire Order. So a natural question is: "Is there a definition of an Acquire Order that yields a programmer-centric memory consistency specification that is equivalent to $Itanium$?" This section examines several plausible Acquire Order definitions and compares their relative strengths. One interesting result is another memory consistency model that is weaker that $Itanium_s$ yet still strictly stronger than $Itanium$.

5.1 Acquire order definitions

Define the following Acquire orders: $i_1 \xrightarrow{Acq} i_2$ if $i_1, i_2 \in I(C)|p \cup I(C)|w$ and $i_1 \xrightarrow{prog} i_2$ and:

Acquire A or Strong Acquire $i_1 \in I(C)|acq$

Acquire B or Weak Acquire $i_1 \in I(C)|acq$ and is foreign

Acquire C $i_1 \in I(C)|acq$ and i_2 is not a domestic load.

Acquire D $i_1 \in I(C)|w$ and $\exists i_3$ such that $i_1 \xrightarrow{prog} i_3 \xrightarrow{prog} i_2$ and $i_3 \in I(C)|acq$ and i_1 is causally related to i_3.

Acquire A and B were defined and motivated in Subsection 3.2. Acquire C models a possible implementation where two load instructions, $i_1 =$ ld.acq program ordered before $i_2 =$ ld or ld.acq, i_1 checks the write-buffer and misses it, bypasses any pending stores, and returns its value from the local replica. Meanwhile i_2 hits the buffer and returns. The effect is that i_2 bypasses i_1 because when constructing the processor's view i_2 will be delayed until its causally-related buffered write is committed to the local replica. Acquire D restricts this behavior in which any instruction can similarly bypass an earlier (in program order) domestic ld.acq. The bypassing instruction cannot be moved too early in the processor's view. It must follow the st that is causally related to the bypassed ld.acq.

If Acquire A is used in the Itanium memory consistency defined in Subsection 3.2 the $Itanium_s$ memory consistency model is defined. If, instead, Acquire B is used

in the Itanium system defined in Subsection 3.2 the $Itanium_w$ memory consistency model is defined. $Itanium_C$ is defined similarly by using the Acquire C order and $Itanium_D$ is defined by using the Acquire D order.

Other Itanium consistency models are defined by the intersection between pairs of these consistency models. For example: Computations satisfying $Itanium_{C\cap D}$ must satisfy both $Itanium_C$ and $Itanium_D$. The verifying sequences that show that a computation satisfies both consistency models may differ (the verifying sequences for $Itanium_C$ and those for $Itanium_D$ may be different sequences).

The final type of Itanium-based memory consistency models that we consider are defined by the conjunction of pairs of consistency models. It requires that there is one set of verifying sequences, which satisfy both models simultaneously. For example an $Itanium_{C\wedge D}$ computation requires one set of sequences that simultaneously meets the properties of both models $Itanium_C$ and $Itanium_D$. Clearly, $Itanium_{C\wedge D}$ is strictly stronger than $Itanium_{C\cap D}$, $Itanium_{C\wedge B}$ is strictly stronger than $Itanium_{C\cap B}$ and $Itanium_{D\wedge B}$ is strictly stronger than $Itanium_{D\cap B}$. Also, observe that $Itanium_s$ is stronger than $Itanium_w$ and $Itanium_C$ and $Itanium_D$. Any of the systems $Itanium_{C\wedge B}$, $Itanium_{C\wedge D}$ and $Itanium_{D\wedge B}$ is also weaker than $Itanium_s$.

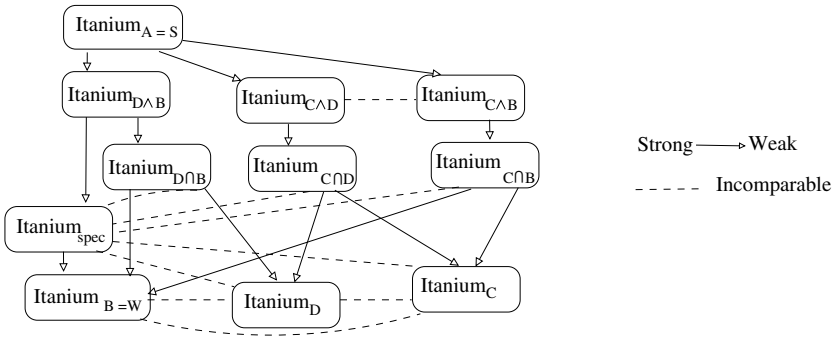

Fig. 1. Relative Strength of Various Systems

5.2 Incomparable consistency models

Figure 1 shows the relative strength of each system. The next two subsections present computations and proofs that establish the relationships in Figure 1. We abbreviate our notation as follows: $a \xrightarrow{valid} b$ means that validity requires that a precedes b in the sequence being discussed; $a \xrightarrow{Same\ Mem} b$ means that the Same Memory Order requires that a precedes b in the sequence being discussed; $a \xrightarrow{Same\ Mem\ Agree} b$ means that the Same Memory Agreement requires that a precedes b in the sequence being discussed; and hence forth.

Computation 1 of Section 4 was shown to satisfy $Itanium_w$ but not $Itanium$ (consequently, it does not satisfy $Itanium_s$). Computation 1 does not satisfy $Itanium_C$ (consequently, it does not satisfy any of $Itanium_{C\cap B}$, $Itanium_{C\wedge B}$, $Itanium_{C\cap D}$, or $Itanium_{C\wedge D}$); Acquire C prohibits a ld.acq to be bypassed by

any following instruction in program order, unless it is a domestic ld. In Computation 1, both ld.acq instructions are followed by a single st instruction. Acquire C requires both ld.acq instructions to maintain their program order with the following st instructions. Hence in Computation 1 Acquire A and Acquire C are equivalent.

Formally, the sequence S_q must extend:
$\mathrm{st}_p(y,4) \xrightarrow{valid} 4 \leftarrow \mathrm{ld.acq}_q(4) \xrightarrow{Acquire\ C} \mathrm{st}_q(x,3)$.
Since $\mathrm{st}_p(y,4) \xrightarrow{S_q} \mathrm{st}_q(x,3)$ the Cycle Free Agreement requires that $\mathrm{st}_p(y,4) \xrightarrow{S_p} \mathrm{st}_q(x,3)$. Thus the sequence S_p does not exist because of the cycle: $\mathrm{st}_p(y,4) \xrightarrow{Cycle\ Free} \mathrm{st}_q(x,3) \xrightarrow{valid} 3 \leftarrow \mathrm{ld}_p(x) \xrightarrow{Same\ Mem} \mathrm{st}_p(x,2) \xrightarrow{Same\ Mem} 2 \leftarrow \mathrm{ld.acq}_p(x) \xrightarrow{Acquire\ C} \mathrm{st}_p(y,4)$.

Computation 1 satisfies $Itanium_D$. Since $4 \leftarrow \mathrm{ld.acq}_q(y)$ is foreign, Acquire D does not require program order to be maintained between $4 \leftarrow \mathrm{ld.acq}_q(y)$ and the following $\mathrm{st}_q(x,3)$. The following verifying sequences show that Computation 1 satisfies $Itanium_D$.

$$\begin{cases} S_p : \mathrm{st}_q(x,3)\ \ 3 \leftarrow \mathrm{ld}_p(x)\ \ \mathrm{st}_p(x,2)\ \ 2 \leftarrow \mathrm{ld.acq}_p(x) \\ \qquad \mathrm{st}_p(y,4) \\ S_q : \mathrm{st}_q(x,3)\ \ \mathrm{st}_p(x,2)\ \ \mathrm{st}_p(y,4)\ \ 4 \leftarrow \mathrm{ld.acq}_q(y) \end{cases}$$

Since Computation 1 satisfies both $Itanium_w$ and $Itanium_D$, it also satisfies $Itanium_{D \cap B}$. However, it does not satisfy $Itanium_{D \wedge B}$. The sequence S_q must extend:
$\mathrm{st}_p(y,4) \xrightarrow{valid} 4 \leftarrow \mathrm{ld.acq}_q(y) \xrightarrow{Acquire\ B} \mathrm{st}_q(x,3)$.
Since $\mathrm{st}_p(y,4) \xrightarrow{S_q} \mathrm{st}_q(x,3)$ the Cycle Free Agreement requires that $\mathrm{st}_p(y,4) \xrightarrow{S_p} \mathrm{st}_q(x,3)$. Thus the sequence S_p does not exist because of the cycle: $\mathrm{st}_p(y,4) \xrightarrow{Cycle\ Free} \mathrm{st}_q(x,3) \xrightarrow{valid} 3 \leftarrow \mathrm{ld}_p(x) \xrightarrow{Same\ Mem} \mathrm{st}_p(x,2) \xrightarrow{Acquire\ D} \mathrm{st}_p(y,4)$.

Computation 3 is an example of a foreign ld.acq followed by a domestic ld.

Comp 3 $\begin{cases} p : \mathrm{st}(x,1)\ \ 2 \leftarrow \mathrm{ld.acq}(y)\ \ 1 \leftarrow \mathrm{ld}(x) \\ q : 1 \leftarrow \mathrm{ld}(x)\ \ \mathrm{st}(x,3)\ \ \mathrm{st.rel}(y,2) \end{cases}$

Computation 3 is not possible under *Itanium*. For q to see 1 in x, $\mathrm{st}_p(x,1)$ must have been applied to both replicas. When p sees 2 in y, $\mathrm{st}_q(x,3)$ must have been also propagated to p's replica, overwriting $\mathrm{st}_p(x,1)$, because the 2 is propagated via a following st.rel. There is no way by which p can see 1 in x, when performing $1 \leftarrow \mathrm{ld}_p(x)$ unless it is performed before $2 \leftarrow \mathrm{ld.acq}_p(y)$; however, *Itanium* requires $2 \leftarrow \mathrm{ld.acq}_p(y)$ to precede $1 \leftarrow \mathrm{ld}_p(x)$.

Formally, Computation 3 does not satisfy *Itanium* (consequently, it does not satisfy $Itanium_s$) because the visibility order V must extend:
$\mathrm{LV}(\mathrm{st}_p(x,1)) \xrightarrow{(WO)} \mathrm{RV}_p(\mathrm{st}_p(x,1)) \xrightarrow{(WO)} \mathrm{RV}_q(\mathrm{st}_p(x,1)) \xrightarrow{(RV2)} \mathrm{R}(1 \leftarrow \mathrm{ld}_q(x)) \xrightarrow{(WAR)} \mathrm{LV}(\mathrm{st}_q(x,3)) \xrightarrow{(WO)} \mathrm{RV}_q(\mathrm{st}_q(x,3)) \xrightarrow{(WO)} \mathrm{RV}_p(\mathrm{st}_q(x,3)) \xrightarrow{(REL)} \mathrm{RV}_p(\mathrm{st.rel}_q(y,2)) \xrightarrow{(RV2)}$

$\mathrm{R}(2 \leftarrow \mathrm{ld.acq}_p(y)) \overset{(ACQ)}{\longrightarrow} \mathrm{R}(1 \leftarrow \mathrm{ld}_p(x))$.
This, however, ensures that the $\mathrm{R}(1 \leftarrow \mathrm{ld}_p(x))$ does not satisfy any of (RV1), (RV2), or (RV3).

Computation 3 is not possible with $Itanium_w$. Note that $2 \leftarrow \mathrm{ld.acq}_p(y)$ is foreign and Acquire A and Acquire B are equivalent in this computation. In other words, $Itanium_w$ requires the buffer to be flushed before the foreign $2 \leftarrow \mathrm{ld.acq}_p(y)$ is performed. Hence, $1 \leftarrow \mathrm{ld}_p(x)$ is not possible using an argument similar to the argument given for $Itanium$.

Formally, Computation 3 does not satisfy $Itanium_w$ (consequently, it does not satisfy any of $Itanium_{C \wedge B}$, $Itanium_{D \wedge B}$, $Itanium_{C \cap B}$, or $Itanium_{D \cap B}$) : the sequence S_q must maintain $\mathrm{st}_p(x,1) \overset{valid}{\longrightarrow} 1 \leftarrow \mathrm{ld}_q(x) \overset{Same\ Mem}{\longrightarrow} \mathrm{st}_q(x,3) \overset{Release}{\longrightarrow} \mathrm{st.rel}_q(y,2)$. Since $\mathrm{st}_p(x,1) \overset{S_q}{\longrightarrow} \mathrm{st}_q(x,3)$, by the Same Memory Agreement we must have $\mathrm{st}_p(x,1) \overset{S_p}{\longrightarrow} \mathrm{st}_q(x,3)$. So, sequence S_p must maintain $\mathrm{st}_p(x,1) \overset{Same\ Mem\ Agree}{\longrightarrow} \mathrm{st}_q(x,3) \overset{Release}{\longrightarrow} \mathrm{st.rel}_q(y,2) \overset{valid}{\longrightarrow} 2 \leftarrow \mathrm{ld.acq}_q(y) \overset{Acquire\ B}{\longrightarrow} 1 \leftarrow \mathrm{ld}_p(x)$, yielding an invalid $1 \leftarrow \mathrm{ld}_p(x)$.

Both Acquire C and Acquire D allow $1 \leftarrow \mathrm{ld}_p(x)$ to appear as if it completed earlier than $2 \leftarrow \mathrm{ld.acq}_p(y)$. Hence Computation 3 is $Itanium_C$ and $Itanium_D$ as formally confirmed by the following sequences:

$$\begin{cases} S_p : \mathrm{st}_p(x,1)\ \ 1 \leftarrow \mathrm{ld}_p(x)\ \ \mathrm{st}_q(x,3)\ \ \mathrm{st.rel}_q(y,2) \\ \qquad 2 \leftarrow \mathrm{ld.acq}_p(y) \\ S_q : \mathrm{st}_p(x,1)\ \ 1 \leftarrow \mathrm{ld}_q(x)\ \ \mathrm{st}_q(x,3)\ \ \mathrm{st.rel}_q(y,2) \end{cases}$$

Note that these sequences simultaneously satisfy $Itanium_D$ and $Itanium_C$. So, Computation 3 also satisfies $Itanium_{C \wedge D}$ and $Itanium_{C \cap D}$.

Computation 4 has a domestic ld.acq followed in program order by a domestic ld. $Itanium_C$ does not constrain the ordering of this ld, while $Itanium_D$ requires that $2 \leftarrow \mathrm{ld}_p(y)$ follows $\mathrm{st.rel}_p(x,1)$ because it is causally related to $1 \leftarrow \mathrm{ld.acq}_p(x)$, which precedes $2 \leftarrow \mathrm{ld}_p(y)$.

Comp 4
$$\begin{cases} p : \mathrm{st}(y,2)\ \ 5 \leftarrow \mathrm{ld}(x)\ \ \mathrm{st.rel}(x,1) \\ \qquad 1 \leftarrow \mathrm{ld.acq}(x)\ \ 2 \leftarrow \mathrm{ld}(y) \\ q : 2 \leftarrow \mathrm{ld}(y)\ \ \mathrm{st}(y,4)\ \ \mathrm{st.rel}(x,5) \end{cases}$$

Computation 4 is not $Itanium$. For q to see 2 in y, it must have been the case that $\mathrm{st}_p(y,2)$ removed from p's buffer and applied to both replicas. So, when p sees 5 in x, it must also have received $\mathrm{st}_q(y,4)$ because the 5 is being propagate by a st.rel. $Itanium$ requires $5 \leftarrow \mathrm{ld}_p(x)$ to precede $\mathrm{st.rel}_p(x,1)$ and $2 \leftarrow \mathrm{ld}_p(y)$ to follow $1 \leftarrow \mathrm{ld.acq}_q(x)$. Also $\mathrm{st.rel}_p(x,1)$ must precede $1 \leftarrow \mathrm{ld.acq}_p(x)$ because both are on x. Hence, $2 \leftarrow \mathrm{ld}_p(y)$ must be performed after $5 \leftarrow \mathrm{ld}_p(x)$. However, we argued that on and after $5 \leftarrow \mathrm{ld}_p(x)$, the value of y in p's replica must be 4, not 2.

Formally, Computation 4 does not satisfy $Itanium$ nor $Itanium_s$. Had it been the case, V must extend: $\mathrm{LV}(\mathrm{st}_p(y,2)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_p(\mathrm{st}_p(y,2)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_q(\mathrm{st}_p(y,2)) \overset{(RV2)}{\longrightarrow} \mathrm{R}(2 \leftarrow \mathrm{ld}_q(y)) \overset{(RAW)}{\longrightarrow} \mathrm{LV}(\mathrm{st}_q(y,4)) \overset{(WO)}{\longrightarrow} \mathrm{RV}_q(\mathrm{st}_q(y,4)) \overset{(WO)}{\longrightarrow}$

$\mathrm{RV}_p(\mathrm{st}_q(y,4)) \overset{(REL)}{\longrightarrow} \mathrm{RV}_p(\mathrm{st.rel}_q(x,5)) \overset{(RV2)}{\longrightarrow} \mathrm{R}(5 \leftarrow \mathrm{ld}_p(x)) \overset{(REL)}{\longrightarrow} \mathrm{LV}(\mathrm{st.rel}_q(x,1)) \overset{(RV1\ or\ 2)}{\longrightarrow} \mathrm{R}(1 \leftarrow \mathrm{ld.acq}_p(x)) \overset{(ACQ)}{\longrightarrow} \mathrm{R}(2 \leftarrow \mathrm{ld}_p(y))$. However this means that the final $\mathrm{R}(2 \leftarrow \mathrm{ld}_p(y))$ does not satisfy any of (RV1), (RV2), or (RV3).

Computation 4 satisfies $Itanium_{C \wedge B}$ (consequently, $Itanium_C$, $Itanium_w$, and $Itanium_{C \cap B}$). $1 \leftarrow \mathrm{ld.acq}_p(x)$ is domestic and under these models, it can be satisfied from the buffer. These models allow $2 \leftarrow \mathrm{ld}_p(y)$ to complete (or appear to complete) before all of the preceding operations: $1 \leftarrow \mathrm{ld.acq}_p(x)$ because it is domestic, $\mathrm{st.rel}_p(x,1)$ because it is on x, and $5 \leftarrow \mathrm{ld}_p(x)$ also because it is on a different variable. This is shown formally by the following sequences:

$$\begin{cases} S_p : \mathrm{st}_p(y,2)\ \ 2 \leftarrow \mathrm{ld}_p(y)\ \ \mathrm{st}_q(y,4)\ \ \mathrm{st.rel}_q(x,5)\ \ 5 \leftarrow \mathrm{ld}_p(x) \\ \qquad \mathrm{st.rel}_p(x,1)\ \ 1 \leftarrow \mathrm{ld.acq}_p(x) \\ S_q : \mathrm{st}_p(y,2)\ \ 2 \leftarrow \mathrm{ld}_q(y)\ \ \mathrm{st}_q(y,4)\ \ \mathrm{st.rel}_q(x,5)\ \ \mathrm{st.rel}_p(x,1) \end{cases}$$

Computation 4 does not satisfy $Itanium_D$ (and hence it does not satisfy any of $Itanium_{C \wedge D}$, $Itanium_{D \wedge B}$, $Itanium_{D \cap B}$, or $Itanium_{C \cap D}$). Acquire D prohibits $2 \leftarrow \mathrm{ld}_p(y)$ to be moved forward in p's view past $\mathrm{st.rel}_p(x,1)$ because it is causally related to $1 \leftarrow \mathrm{ld.acq}_p(x)$ which precedes $2 \leftarrow \mathrm{ld}_p(y)$. So, a similar argument given for *Itanium* applies here.

Formally, the sequence S_q must extend:
$\mathrm{st}_p(y,2) \overset{valid}{\longrightarrow} 2 \leftarrow \mathrm{ld}_q(y) \overset{Same\ Mem}{\longrightarrow} \mathrm{st}_q(y,4)$. Thus, the Same Memory Agreement requires that $\mathrm{st}_p(y,2) \overset{S_p}{\longrightarrow} \mathrm{st}_q(y,4)$. Therefore, S_p must extend:
$\mathrm{st}_p(y,2) \overset{Same\ Mem\ Agree}{\longrightarrow} \mathrm{st}_q(y,4) \overset{Release}{\longrightarrow} \mathrm{st.rel}_q(x,5) \overset{valid}{\longrightarrow} 5 \leftarrow \mathrm{ld}_p(x) \overset{Release}{\longrightarrow} \mathrm{st.rel}_p(x,1) \overset{Acquire\ D}{\longrightarrow} 2 \leftarrow \mathrm{ld}_p(y)$. This ensures that the final $2 \leftarrow \mathrm{ld}_p(y)$ is invalid.

In Computation 5, $1 \leftarrow \mathrm{ld.acq}_p(x)$ is domestic and is followed by a st. only Acquire B among all other acquire orders allows $\mathrm{st}_p(y,2)$ to occur earlier than $1 \leftarrow \mathrm{ld}_p(x)$ in p's view.

Comp 5 $\begin{cases} p : \mathrm{st}(x,1)\ \ 1 \leftarrow \mathrm{ld.acq}(x)\ \ \mathrm{st}(y,2) \\ q : 2 \leftarrow \mathrm{ld}(y)\ \ \mathrm{st}(y,3)\ \ \mathrm{st.rel}(x,4) \\ t : 4 \leftarrow \mathrm{ld.acq}(x)\ \ 1 \leftarrow \mathrm{ld}(x) \end{cases}$

Computation 5 satisfies *Itanium* (Consequently, $Itanium_w$) as shown by the following visibility order V:
$\mathrm{LV}(\mathrm{st}_p(x,1))$ $\mathrm{R}(1 \leftarrow \mathrm{ld.acq}_p(x))$ $\mathrm{LV}(\mathrm{st}_p(y,2))$ $\mathrm{RV}_p(\mathrm{st}_p(y,2))$ $\mathrm{RV}_q(\mathrm{st}_p(y,2))$ $\mathrm{RV}_t(\mathrm{st}_p(y,2))$ $\mathrm{R}(2 \leftarrow \mathrm{ld}_q(y))$ $\mathrm{LV}(\mathrm{st}_q(y,3))$
$\mathrm{RV}_q(\mathrm{st}_q(y,3))$ $\mathrm{RV}_p(\mathrm{st}_q(y,3))$ $\mathrm{RV}_t(\mathrm{st}_q(y,3))$ $\mathrm{LV}(\mathrm{st.rel}_q(x,4))$ $\mathrm{RV}_q(\mathrm{st.rel}_q(x,4))$ $\mathrm{RV}_p(\mathrm{st.rel}_q(x,4))$ $\mathrm{RV}_t(\mathrm{st.rel}_q(x,4))$
$\mathrm{R}(4 \leftarrow \mathrm{ld.acq}_t(x))$ $\mathrm{RV}_p(\mathrm{st}_p(x,1))$ $\mathrm{RV}_t(\mathrm{st}_p(x,1))$ $\mathrm{R}(1 \leftarrow \mathrm{ld}_t(x))$ $\mathrm{RV}_q(\mathrm{st}_p(x,1))$.

Computation 5 is $Itanium_w$ as shown by the following sequences:

$$\begin{cases} S_p : \mathrm{st}_p(y,2)\ \ \mathrm{st}_q(y,3)\ \ \mathrm{st.rel}_q(x,4)\ \ \mathrm{st}_p(x,1)\ \ 1 \leftarrow \mathrm{ld.acq}_p(x) \\ S_q : \mathrm{st}_p(y,2)\ \ 2 \leftarrow \mathrm{ld}_q(y)\ \ \mathrm{st}_q(y,3)\ \ \mathrm{st.rel}_q(x,4)\ \ \mathrm{st}_p(x,1) \\ S_t : \mathrm{st}_p(y,2)\ \ \mathrm{st}_q(y,3)\ \ \mathrm{st.rel}_q(x,4) \\ \qquad 4 \leftarrow \mathrm{ld.acq}_t(x)\ \ \mathrm{st}_p(x,1)\ \ 1 \leftarrow \mathrm{ld}_t(x) \end{cases}$$

However, Computation 5 does not satisfy $Itanium_C$ nor $Itanium_D$. The se-

quence S_q must extend $\mathrm{st}_p(y,2) \overset{valid}{\longrightarrow} 2 \leftarrow \mathrm{ld}_q(y) \overset{Same\ Mem}{\longrightarrow} \mathrm{st}_q(y,3)$. Thus, the Same Memory Agreement requires that $\mathrm{st}_p(y,2) \longrightarrow \mathrm{st}_q(y,3)$ in all sequences. So, the sequence S_p must extend $\mathrm{st}_p(x,1) \overset{Same\ Mem}{\longrightarrow} 1 \leftarrow \mathrm{ld.acq}_p(x) \overset{Acquire\ C}{\longrightarrow} \mathrm{st}_p(y,2) \overset{Same\ Mem\ Agree}{\longrightarrow} \mathrm{st}_q(y,3) \overset{Release}{\longrightarrow} \mathrm{st.rel}_q(x,4)$ or $\mathrm{st}_p(x,1) \overset{Acquire\ D}{\longrightarrow} \mathrm{st}_p(y,2) \overset{Same\ Mem\ Agree}{\longrightarrow} \mathrm{st}_q(y,3) \overset{Release}{\longrightarrow} \mathrm{st.rel}_q(x,4)$. Thus, the Same Memory Agreement requires that $\mathrm{st}_p(x,1) \longrightarrow \mathrm{st.rel}_q(x,4)$ in all sequences. Observe that the final part of Same Memory Order requires that sequence S_t maintains $4 \leftarrow \mathrm{ld.acq}(x)$ before $1 \leftarrow \mathrm{ld}(x)$ and thus it cannot be valid.

Consequently, Computation 5 does not satisfy any of the models: $Itanium_{C \wedge D}$, $Itanium_{C \wedge B}$, $Itanium_{D \wedge B}$, $Itanium_{D \cap B}$, $Itanium_{C \cap D}$, or $Itanium_{C \cap B}$.

Theorem 5.1 *Itanium is incomparable to*

(i) $Itanium_C$,

(ii) $Itanium_D$,

(iii) $Itanium_{C \wedge D}$,

(iv) $Itanium_{C \wedge B}$,

(v) $Itanium_{D \cap B}$,

(vi) $Itanium_{C \cap B}$,

(vii) $Itanium_{C \cap D}$

Proof.

(i) Computation 3 is not *Itanium* but is $Itanium_C$. Computation 5 is not $Itanium_C$ but is *Itanium*.

(ii) Computation 1 is not *Itanium* but is $Itanium_D$. Computation 5 is not $Itanium_D$ but is *Itanium*.

(iii) Computation 3 is not *Itanium* but is $Itanium_{C \wedge D}$. Computation 5 is not $Itanium_{C \wedge D}$ but is *Itanium*.

(iv) Computation 4 is not *Itanium* but is $Itanium_{C \wedge B}$. Computation 5 is not $Itanium_{C \wedge B}$ but is *Itanium*.

(v) Computation 1 is not *Itanium* but is $Itanium_{D \cap B}$. Computation 5 is not $Itanium_{D \cap B}$ but is *Itanium*.

(vi) Computation 4 is not *Itanium* but is $Itanium_{C \cap B}$. Computation 5 is not $Itanium_{C \cap B}$ but is *Itanium*.

(vii) Computation 3 is not *Itanium* but is $Itanium_{C \cap D}$. Computation 5 is not $Itanium_{C \cap D}$ but is *Itanium*.

□

Theorem 5.2 *$Itanium_w$ is incomparable to*

(i) $Itanium_C$, *and*

(ii) $Itanium_D$.

Proof.

(i) Computation 3 is not $Itanium_w$ but is $Itanium_C$. Computation 1 is not $Itanium_C$ but is $Itanium_w$.

(ii) Computation 3 is not $Itanium_w$ but is $Itanium_D$. Computation 5 is not $Itanium_D$ but is $Itanium_w$.

□

Theorem 5.3 *$Itanium_C$ is incomparable to $Itanium_D$.*

Proof. Computation 4 is not $Itanium_D$ but is $Itanium_C$. Computation 1 is not $Itanium_C$ but is $Itanium_D$. □

Theorem 5.4 *$Itanium_{C\wedge D}$ is incomparable to $Itanium_{C\wedge B}$.*

Proof. Computation 3 is not $Itanium_{C\wedge B}$ but is $Itanium_{C\wedge D}$. Computation 4 is not $Itanium_{C\wedge D}$ but is $Itanium_{C\wedge B}$. □

5.3 A consistency model strictly weaker than Itanium$_s$ and stronger than Itanium

It can be established that $Itanium_{D\wedge B}$ is strictly stronger than *Itanium* [8].

Theorem 5.5 *$Itanium_{D\wedge B}$ memory consistency is strictly stronger than Itanium memory consistency.*

So, $Itanium_{D\wedge B}$ is weaker than $Itanium_s$ but still stronger than *Itanium*. At present a programmer-centric consistency model that is equivalent to *Itanium* has not been identified. However, there is promise in this technique of strengthening the Acquire B order.

6 An Itanium Operational Model

One possible *Itanium* machine has been defined by Chatterjee and Gopalkrishnan [3]. They provide an operational model that is defined in terms of buffers and memories and uses non-deterministic ordering rules. Figure 2 is a drawing of their machine. The local replica of processor p is denoted M_p. Each processor has three buffers: Write Output Buffer (WOB), Write Input Buffer (WIB), and a Read Buffer (RD). These abbreviations are subscripted by the owner's id when required. For example, WOB_p is the WOB of p.

We provide an informal description of the associated transition system. A ld.acq by p atomically checks p's WOB for pending stores to the same memory location and either returns that value or, if none exist, returns the value of that location from M_p. A ld by p checks p's WOB for pending stores to the same memory location and either returns that value or issues the ld to p's RB, denoted Issue(ld to RB_p). A st.rel by p issues the st.rel to p's WOB, Issue(st.rel to WOB_p). A st by p issues the st to p's WOB, Issue(st to WOB_p). A fence by p flushes the buffers, Flush(p). A memory write event at p checks to see if it is allowed, then updates memory p from the WIB for p and deletes the instruction, i from the WIB for p. We will denote

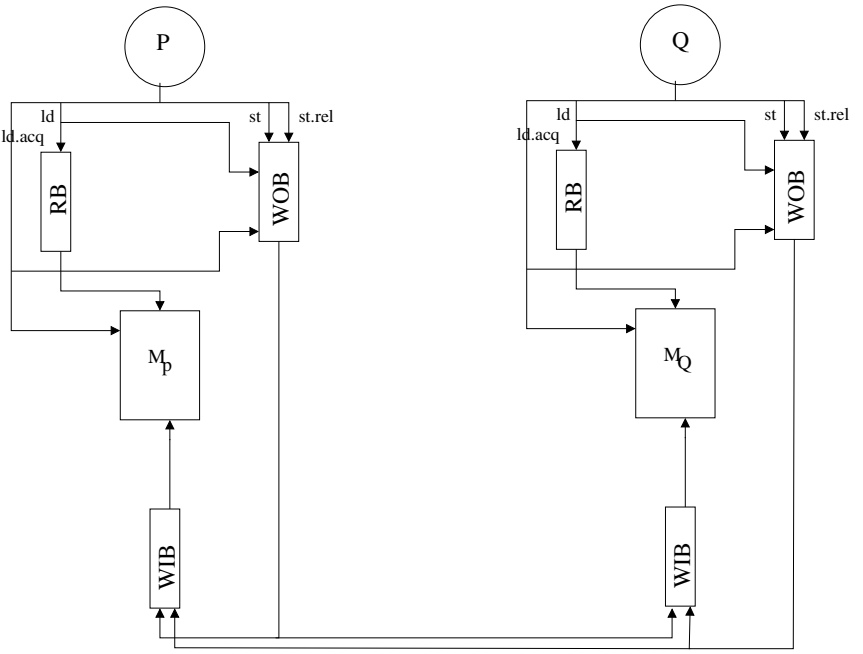

Fig. 2. Chatterjee and Gopalkrishnan Operational Model

these operations as Update(i into M_p) and Delete(i from WIB_p) respectively. A memory read event at p returns the specified value from memory and deletes the load from the RB for p. We use the completed ld instruction to indicate the return of the read value and denote the delete as Delete(i from RB_p). This short summary is missing many details about when transitions can occur. See [3] for that information.

Computation 4, which was previously shown to be not possible on Itanium as specified by the architectural definition [9], is possible on this machine. The ld.acq in p of that computation is domestic it completes before the $5 \leftarrow \mathrm{ld}_p(x)$ instruction which was issued to RB_p. The following sequence shows the transitions:

$\mathrm{Issue}(\mathrm{st}_p(y,2)$ to $\mathrm{WOB}_p) \longrightarrow$
$\mathrm{Issue}(\mathrm{ld}_p(x)$ to $\mathrm{RB}_p) \longrightarrow$
$\mathrm{Issue}(\mathrm{st.rel}_p(x,1)$ to $\mathrm{WOB}_p) \longrightarrow$
$1 \leftarrow \mathrm{ld.acq}_p(x)$ [7] $\longrightarrow$
$2 \leftarrow \mathrm{ld}_p(y)$ [8] $\longrightarrow$
[$\mathrm{Delete}(\mathrm{st}_p(y,2)$ from $\mathrm{WOB}_p)$ $\mathrm{Issue}(\mathrm{st}_p(y,2)$ to $\mathrm{WIB}_p)$
$\mathrm{Issue}(\mathrm{st}_p(y,2)$ to $\mathrm{WIB}_q)$] $\longrightarrow$
[$\mathrm{Delete}(\mathrm{st}_p(y,2)$ from $\mathrm{WIB}_q)$
Update(st(y)2 into M_q)] $\longrightarrow$
[$\mathrm{Delete}(\mathrm{st}_p(y,2)$ from $\mathrm{WIB}_p)$
Update(st(y)2 into M_p)] $\longrightarrow$
Issue($\mathrm{ld}_q(y)$ to $\mathrm{RB}_q) \longrightarrow$

[7] Completes using a value in WOB_p
[8] Completes using a value in WOB_p

$[\ \text{Delete}(\text{ld}_q(y) \text{ from } \text{RB}_q)\ 2 \leftarrow \text{ld}_q(y)\ ] \longrightarrow$
$\text{Issue}(\text{st}_q(y,4) \text{ to } \text{WOB}_q) \longrightarrow$
$[\ \text{Delete}(\ \text{st}_q(y,4) \text{ from } \text{WOB}_q)\ \text{Issue}(\text{st}_q(y,4) \text{ to } \text{WIB}_p)$
$\quad \text{Issue}(\text{st}_q(y,4) \text{ to } \text{WIB}_q)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st}_q(y,4) \text{ from } \text{WIB}_p)$
$\quad \text{Update}(\text{st}_q(y,4) \text{ into } \text{M}_p)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st}_q(y,4) \text{ from } \text{WIB}_q)$
$\quad \text{Update}(\text{st}_q(y,4) \text{ into } \text{M}_q)\ ] \longrightarrow$
$\text{Issue}(\text{st.rel}_p(x,5) \text{ to } \text{WOB}_q) \longrightarrow$
$[\ \text{Delete}(\text{st.rel}_p(x,5) \text{ from } \text{WOB}_q)\ \text{Issue}(\text{st.rel}_p(x,5) \text{ to } \text{WIB}_p)$
$\quad \text{Issue}(\text{st.rel}_p(x,5) \text{ to } \text{WIB}_q)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st.rel}_p(x,5) \text{ from } \text{WIB}_p)$
$\quad \text{Update}(\text{st.rel}_p(x,5) \text{ into } \text{M}_p)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st.rel}_p(x,5) \text{ from } \text{WIB}_q)$
$\quad \text{Update}(\text{st.rel}_p(x,5) \text{ into } \text{M}_q)\ ] \longrightarrow$
$[\ \text{Delete}(\text{ld}_p(x) \text{ from } \text{RB}_p)\ 5 \leftarrow \text{ld}_p(x)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st.rel}_p(x,1) \text{ from } \text{WOB}_p)\ \text{Issue}(\text{st.rel}_p(x,1) \text{ to } \text{WIB}_p)$
$\quad \text{Issue}(\text{st.rel}_p(x,1) \text{ to } \text{WIB}_q)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st.rel}_p(x,1) \text{ from } \text{WIB}_p)$
$\quad \text{Update}(\text{st.rel}_p(x,1) \text{ into } \text{M}_p)\ ] \longrightarrow$
$[\ \text{Delete}(\text{st.rel}_p(x,1) \text{ from } \text{WIB}_q)$
$\quad \text{Update}(\text{st.rel}_p(x,1) \text{ into } \text{M}_q)\ ]\ .$

It remains to determine the relationship between this machine and $Itanium_w$. The problem of finding an operational model that exactly captures *Itanium* also remains open.

7 Concluding Remarks

The Itanium memory consistency model is specified at the architectural level, without a reference architecture implementation. Such a low-level specification can be very useful to chip verification. However, they are not convenient to programmers and algorithm designers, who normally reason about their programs at a higher level. This work attempts but fails to formulate a programmer-centric description of the Itanium memory consistency model. Instead, it provides two very similar definitions (stronger and weaker than Itanium) that bound the official lower-level specifications. These two definitions differ in the way the load-acquires are implemented in the presence of write-buffers, such as if and when a load-acquire causes the buffer to be flushed.

This lead us to investigate different possible acquire orders and consequently different possible implementations for load-acquires. The result is an array of different programmer-centric models largely incomparable to each other, but none of them tightly captures the official Itanium memory model.

We have also looked at an earlier attempt to provide an operational model for Itanium. We showed that this implementation admits behaviors that are prohibited

under Itanium.

Though these definitions do not tightly capture Itanium, they are still very useful to programmers. For instance to prove that a problem cannot be solved with Itanium, it suffices to show it is not solvable using Strong Itanium. To prove the correctness of an algorithm for Itanium, it suffices to show correctness under Weak Itanium. This however does not replace a single programmer-centric definition that is tightly captures the Itanium behavior. This work shows that this goal is a real challenge. We hope that the techniques demonstrated in this paper allow us to achieve this goal.

References

[1] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the PowerPC architecture. *IEEE Trans. on Parallel and Distributed Systems*, 14(5):502–515, 2003.

[2] Arvind and J. W. Maessem. Memory model = instruction reordering + store atomicity. In *Proceedings of ISCA 2006*.

[3] P. Chatterjee and G. Gopalakrishnan. Towards a formal model of shared memory consistency for intel itaniumtm. In *Proc. 2001 IEEE International Conference on Computer Design (ICCD)*, pages 515–518, Sept 2001.

[4] G. Gopalakrishnan, Y. Yang and H. Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *Proc. 16th International Conference on Computer Aided Verification (CAV04)*, 2004.

[5] L. Higham and L. Jackson. Porting between itanium and sparc multiprocessing systems. In *Accepted to: 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '06)*, to appear 2006.

[6] L. Higham, L. Jackson, and J. Kawash. Specifying memory consistency of write buffer multiprocessors. *ACM Trans. on Computer Systems*. To appear.

[7] L. Higham, L. Jackson, and J. Kawash. Capturing register and control dependence in memory consistency models with applications to the itanium architecture, May 2006. Submitted to: DISC 2006.

[8] L. Higham, L. Jackson, and J. Kawash. Programmer-centric conditions for itanium memory consistency. Technical Report Technical Report 2006-838-31, Department of Computer Science, The University of Calgary, July 2006.

[9] Intel Corporation. A formal specification of the intel itanium processor family memory ordering. http://www.intel.com/, Oct 2002.

[10] Intel Corporation. Intel itanium architecture software developer's manual, volume 2: System architecture. http://www.intel.com/, Oct 2002.

[11] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla, 2003.

[12] SPARC Int'l, Inc. *The SPARC Architecture Manual version 8*. Prentice-Hall, 1992.

[13] Y. Yang, G. Gopalakrishnan and K. Slind. Analyzing the intel itanium memory ordering rules using logic programming and sat. In *Proc. 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME03)*, pages 81–95, 2003.

[14] Y. Yang, G. Gopalakrishnan and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *Proc. 18th International Parallel and Distributed Processing Symposium (IPDPS04)*, 2004.