# Programmer-Centric Conditions for Itanium Memory Consistency

Lisa Higham, LillAnne Jackson, and Jalal Kawash
The University of Calgary, Calgary, Canada

**Abstract**

A *programmer-centric model* of memory consistency provides a sequence of instructions for each processor, and requires that these sequences satisfy a collection of rules. It also requires that the notion of validity of a sequence is the natural one: the value read from a shared memory location must be one that was written by the most recent preceding instruction that stored to the same location. A programmer-centric model supports reasoning about programs at a non-operational level. It is 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, each motivated by slightly different implementations of load-acquire instructions, and prove that the specification of the Itanium memory model lies strictly between the two. We also entertain a handful of other natural notions of load-acquire rules and show that none exactly captures the Itanium specification. This leads us to question whether the specification of the Itanium memory order [5] is actually faithful to the Itanium architects' intentions.

**Keywords:** Programmer-centric memory consistency, Itanium multiprocessor

## 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 $\mathcal{A}$.

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 [4, 3] we established a framework for specifying programmer-centric memory consistency models and for proving such equivalences between specifications and implementations. We applied this framework and the proof techniques to an extensive example involving write buffer architecture [3]. 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, called Itanium$_A$ and Itanium$_B$, and prove that the "official" Itanium memory consistency [5], henceforth referred to as Itanium (with no subscript), lies strictly between these two (Section 3). We even show that eight other plausible programmer-centric definitions also fail to exactly capture the Itanium memory consistency specification (Section 4). The resulting ten Itanium definitions differ only (often, slightly) in their ordering constraints involving Itanium load-acquire instructions, and each is motivated by a plausible hardware implementation. Yet, none of these tightly captures the specification of the Itanium memory consistency [5]. The operational model proposed by Chatterjee and Gopalakrishnan [1] is another reasonable and plausible Itanium implementation. Yet, it does not exactly satisfy the Itanium [6]. We know of no description that has a plausible hardware implementation and is equivalent to Itanium. We are led to speculate whether the specification of the Itanium memory consistency [5] is really what the Itanium architects intended!

Before we present our main results, we briefly describe the model in Subsection 2.1. Subsection 2.2 defines Itanium$_A$ and Itanium$_B$. The paper requires familiarity with the Itanium memory consistency model [5]. Those definitions that are essential for this work and are summarized in Section 2.3.

Several other frameworks for describing memory consistency have been proposed but are not central to this paper. Yang *et. al.* [8] 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.* [7] is a precise specification and is the basis of the official specification [5].

# 2 Multiprocessors, Computations and Memory Consistency

## 2.1 Instructions, multiprocessors and computations

As each processor in a multiprocessor system executes, it issues a sequence of instruction invocations on shared memory objects.[1] For this paper the shared memory consists of only shared variables. Each *instruction invocation* is of the form $\mathrm{st}_p(x, v)$ or $\mathrm{st.rel}_p(x, v)$ meaning that processor $p$ writes a value $v$ to the shared variable $x$, or $\mathrm{ld}_p(x)$ or $\mathrm{ld.acq}_p(x)$ meaning that $p$ reads a value from shared variable $x$, or $\mathrm{fence}_p$ meaning that $p$ invokes a memory fence instruction. Instructions st and st.rel are referred to collectively as *store* instructions; ld and ld.acq are called *load* instructions. These five forms of instruction invocations are called *Itanium-based*. It suffices to model each individual processor as a sequence of these instruction invocations and call such a sequence an *individual Itanium-based program*.[2] An *(Itanium-*

[1] Some of the definitions in this subsection were used in previous work (Section 2.2 of [2]); they are re-stated here in modified form.

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

*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 or fence instruction invocation is an acknowledgment and is ignored. The response of a load invocation is the value returned by the invocation. Let $P$ be an Itanium-based multiprogram. An *(Itanium-based multiprocessor) computation of* $P$ 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 $\nu$ is either the initial value of $x$ or some value stored to $x$ by a store instruction in $P$.

Notice that the definition of a computation permits the value returned by each load instruction invocation of variable $x$ 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 processors 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. We use $\mathcal{C}(P, \mathrm{MC})$ to denote the set of all computations of multiprogram $P$ that satisfy the memory consistency model MC. A collection of such sequences that meet all the requirements of MC is called a set of *MC-verifying sequences*. If $C \in \mathcal{C}(P, \mathrm{MC})$, then there exist MC-verifying sequences *for* $C$. Memory consistency model MC is *stronger than* MC' if, for every Itanium-based Multiprogram $P$, $\mathcal{C}(P, \mathrm{MC}) \subseteq \mathcal{C}(P, \mathrm{MC}')$. Similarly, The term *strictly stronger* requires that $\mathcal{C}(P, \mathrm{MC}) \subset \mathcal{C}(P, \mathrm{MC}')$.

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; $I(C)|acq$ is the subset containing all ld.acq instructions plus the memory fence instructions; $I(C)|rel$ is 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 of $C$ by the same processor $p$ such that the invocation of $i$ precedes the invocation of $j$ in $p$'s program. For any partial order relation $(I(C), \xrightarrow{y})$, the notation $i \xrightarrow{y} j$ is used interchangeably with $(i, j) \in (I(C), \xrightarrow{y})$. Also, if $(I(C), \xrightarrow{y})$ is a total order on a finite set, then $y$ denotes the unique sequence such that $i$ precedes $j$ in $y$ if and only if $(i, j) \in (I(C), \xrightarrow{y})$.

A load instruction is *domestic* if the value it returns was stored into a shared variable by the same processor. a *foreign* instruction is either a non-domestic load or a fence instruction. If a load instruction, $i$, returns the value stored by a store instruction, $j$, to the same variable then $i$ and $j$ are *causally related*. A sequence of Itanium-based instructions is *valid* if, for every load instruction, its causally related store instruction is the most recent preceding store to the same variable in the sequence.

## 2.2 Weak and strong Itanium memory consistency

The following definition is parameterized by an arbitrary partial order on $I(C)$, denoted $R$, which will be replaced by various partial orders to construct differing versions of Itanium consistency.

**Definition 2.1** *A computation $C$ satisfies* $\mathrm{Itanium}_R$ *if for each $p \in P$ there is a total order $(I(C)|p \cup I(C)|w, \xrightarrow{S_p})$ such that $S_p$ is valid and for every $i, j \in I(C)|p \cup I(C)|w$:*

1. *If $i \xrightarrow{R} j$ then $i \xrightarrow{S_p} j$ (R Order), and*
2. *If $i \xrightarrow{prog} j$ and $j \in I(C)|rel$ then $i \xrightarrow{S_p} j$ (Release Order), and*
3. *If $i \xrightarrow{prog} j$ and $i, j \in I(C)|x$ and [($i \in I(C)|w$ or $j \in I(C)|w$) or ($i \in I(C)|acq$)] then $i \xrightarrow{S_p} j$ (Same Memory Order), and*

4. *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*

5. *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*

6. *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*

7. *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* $i_2 \xrightarrow{S_3} i_3$ *... and* $i_{k-1} \xrightarrow{S_k} i_k$ *(Cycle Free Agreement).*

Define the following partial orders. Let $i, j \in I(C)$ such that $i \xrightarrow{prog} j$.

**Acquire A:** $i \xrightarrow{Acquire\ A} j$ if and only if $i \in I(C)|acq$.

**Acquire B:** $i \xrightarrow{Acquire\ B} j$ if and only if $i \in I(C)|acq$ and $i$ is foreign.

Itanium$_A$ abbreviates Itanium$_R$ when $R$ = Acquire A. Itanium$_B$ is defined similarly ($R$ = Acquire B). Section 4 defines additional Itanium models based on further variants of Acquire orders.

## 2.3 Memory consistency specification of the basic Itanium processor family

In order to prove that Itanium$_B$ and Itanium$_A$ bound Intel's definition of the *basic Itanium processor family memory ordering model*, we make extensive reference to that model as defined by the Intel manual [5]. This section is quoted and paraphrased from [5].

### 2.3.1 The framework of the Itanium manual

In [5] an execution is considered to be a set of sequences of instructions, one sequence per processor. Each sequence contains the instructions performed by the associated processor listed in program order.[3] Program order is a total order when restricted to the instructions of a single processor.

Each instruction can read values from memory, write values to memory, or neither read nor write memory.[4] For any two instructions $\mathtt{I}_1$ and $\mathtt{I}_2$, if $\mathtt{I}_1$ and $\mathtt{I}_2$ are by the same process (denoted p=Proc($\mathtt{I}_1$)=Proc($\mathtt{I}_2$) and $\mathtt{I}_1$ precedes $\mathtt{I}_2$ in program order, we write $\mathtt{I}_1 \xrightarrow{prog} \mathtt{I}_2$.[5]

Instructions are decomposed into *operations*.[6] As an example, a load instruction may be thought of as having a read operation, a store instruction as having a write operation and a semaphore as having both read and write operations. In general, an instruction's operations correspond to different aspects of the visibility of the instruction at different processors. A load instruction R is *local for byte b* if b∈Rng(R) and there is a store W to b with p=Proc(W)=Proc(R) such that LV(W) $\longrightarrow$ R(R) $\longrightarrow$ RV$_p$(W).

- If an instruction or operation X reads from memory, we say that X has *read semantics* or is a *read*. RdVal(X) is the value read by X from Rng(X).[7] If b∈Rng(X), RdVal(X;b) is the value read by X for byte b and we say that X is read from b.
- If an instruction or operation X writes from memory, we say that X has *write semantics* or is a *write*. WrVal(X) is the value written by X from Rng(X). If b∈Rng(X), WrVal(X;b) is the value written by X to byte b and we say that X is write to b. The expression Rng($\mathtt{I}_1$)∩Rng($\mathtt{R}_2$)$\neq \phi$ instructions $\mathtt{I}_1$ and $\mathtt{I}_2$ access the same memory locations. The notation Rng(I) expresses the range of memory locations on which an instruction operates.
- Every byte in memory has an *initial value* that it will return to reads that occur before there are any writes to the byte. For byte b, this value is denoted by InitVal(b).

[3] In our framework this is called a computation.

[4] In [5] they also allow an instruction to do both read and write, but this is skipped here because we are not considering semaphore instructions in this work.

[5] In [5] the notation $\gg$ is used. We use $\xrightarrow{prog}$ to maintain consistency with our notation.

[6] In [5], each instruction is first decomposed into and access then into operations. We omit can omit accesses in this work.

[7] In this work we are assuming that memory locations do not overlap, that is Rng(X) is distinct for every X.

Every execution of the model being defined has a single associated *visibility order* which we call V. This order must totally (i.e., linearly) order all the operations the execution generates. Each specification has a set of rules that constrain the order in which the operations can appear and how the operations affect memory. For any two operations $\mathtt{O}_1$ or $\mathtt{O}_2$, if $\mathtt{O}_1$ precedes $\mathtt{O}_2$ in visibility order, we write $\mathtt{I}_1 \longrightarrow \mathtt{I}_2$.

### 2.3.2 Visibility Order Rules

In this section the symbol W is used to represent an instruction that has write semantics, the symbol R represents an instruction with read semantics, the symbol FEN represents a fence instruction, the symbol ACQ represents an instruction with acquire semantics, the symbol REL represents an instruction with acquire semantics and the symbol SR represents a instruction store release instruction. Also in V, the symbol R( ) represents a read operation, LV( ) represents a local visibility operation, $\mathrm{RV}_q$( ) a remote visibility operation at processor q.

The visibility order V of the executions of the basic Itanium processor family memory ordering model must satisfy the following rules. If there is no visibility order for an execution that satisfies all of these rules, the execution is not permitted by the architecture.

#### Write Operation Order

**(WO):** No write can become visible remotely before it becomes visible locally. For every write instruction W, $\mathrm{LV}(\mathtt{W}) \longrightarrow \mathrm{RV}_p(\mathtt{W})$ for p=proc(W), and $\mathrm{RV}_p(\mathtt{W}) \longrightarrow \mathrm{RV}_q(\mathtt{W})$ for p=proc(W) and q≠proc(W).

#### Program Order:

**(ACQ):** No operation can become visible before a preceding acquire. If $\mathtt{ACQ} \xrightarrow{prog} \mathtt{I}$, A is an operation of instruction ACQ, and O is an operation of instruction I, then $\mathtt{A} \longrightarrow \mathtt{O}$.

**(REL)** : No release can become visible before a preceding instruction's operations.

- If $\mathtt{I} \xrightarrow{prog} \mathtt{REL}$, instruction I does not have write semantics, and operation O is an operation of I, then $\mathtt{O} \longrightarrow \mathrm{LV}(\mathtt{REL})$.
- If $\mathtt{I} \xrightarrow{prog} \mathtt{REL}$ and instruction I has write semantics, then $\mathrm{LV}(\mathtt{I}) \longrightarrow \mathrm{LV}(\mathtt{REL})$ and $\mathrm{RV}_p(\mathtt{I}) \longrightarrow \mathrm{RV}_p(\mathtt{REL})$ for each processor p.

Recall that all instructions with release semantics also have write semantics and thus have LV and RV operations.

**(FEN)** [8]: Operations become visible in-order with respect to memory fences.

- If $\mathtt{FEN} \xrightarrow{prog} \mathtt{I}$ and O is an operation of instruction I, then $\mathrm{F}(\mathtt{FEN}) \longrightarrow \mathtt{O}$.
- If $\mathtt{I} \longrightarrow \mathtt{FEN}$ and O is an operation of instruction I, then $\mathtt{O} \longrightarrow \mathrm{F}(\mathtt{FEN})$.

Notice that either case implies that any two memory fences become visible in program order: if $\mathtt{FEN}_1 \xrightarrow{prog} \mathtt{FEN}_2$, then $\mathrm{F}(\mathtt{FEN}_1) \longrightarrow \mathrm{F}(\mathtt{FEN}_2)$.

#### Memory-Data Dependence:

**(MD:RAW)** : No read may become visible locally before an earlier write to a common location.

- If $\mathrm{Rng}(\mathtt{W}) \cap \mathrm{Rng}(\mathtt{R}) \neq \phi$ and $\mathtt{W} \xrightarrow{prog} \mathtt{R}$, then $\mathrm{LV}(\mathtt{W}) \longrightarrow \mathrm{R}(\mathtt{R})$.

[8] In [5] this is called (REL). We prefer to call it (FEN) to distinguish from the previous item.

**(MD:WAR)** : No write may become visible locally before an earlier read to a common location.

- If Rng(R)∩Rng(W)≠ $\phi$ and R $\xrightarrow{prog}$ W, then R(R) $\longrightarrow$ LV(W).

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

- If Rng($W_1$)∩Rng($W_2$)≠ $\phi$ and $W_1 \xrightarrow{prog} W_2$, then LV($W_1$) $\longrightarrow$ LV($W_2$) for processor p=Proc($W_1$)=Proc($W_2$).

**Coherence:**

**(COH)** : Suppose that $W_1$ and $W_2$ are write instructions to the same non-WC memory attribute and that Rng($W_1$)∩Rng($W_2 \neq \phi$. The following must hold:

- If LV($W_1$) $\longrightarrow$ LV($W_2$) and processor p=Proc($W_1$)=Proc($W_2$), then $RV_p(W_1) \longrightarrow RV_p(W_2)$ for all processors p.
- If $RV_p(W_1) \longrightarrow RV_p(W_2)$ for processor p, then $RV_q(W_1) \longrightarrow RV_q(W_2)$ for all processors q.

**Read Value:**

**(RV1)** : Suppose that R is local for b. Let W be a write to b such that Proc(W)=p, LV(W) $\longrightarrow$ R(R), and there is no other write W' to b with Proc(W')=p and LV(W) $\longrightarrow$ LV(W') $\longrightarrow$ R(R). Then RdVal(R; b) = WrVal(W: b).

**(RV2)** : Suppose that R (where Proc(R)= $p$) is not local for b, there is a W to b such that $RV_p(W) \longrightarrow$ R(R), and there is no other write W' to b with $RV_p(W) \longrightarrow RV_p(W') \longrightarrow$ R(R). Then RdVal(R; b) = WrVal(W: b).

**(RV3)** : Suppose that R (where Proc(R)= $p$) is not local for b, and there is no W to b such that $RV_p(W) \longrightarrow$ R(R). Then RdVal(R; b) = InitVal(b).

Total Ordering of WB Releases:

**(WBR)** : Store-releases that write to WB memory become remotely visible atomically.

- If SR writes to WB memory and $RV_p(SR) \longrightarrow o \longrightarrow RV_q(SR)$ for processors then o=$RV_r$(SR) for some processor r.

# 3 Itanium **is strictly between** Itanium$_B$ **and** Itanium$_A$

## 3.1 Itanium **computations satisfy** Itanium$_B$

Let $P$ be any Itanium-based multiprogram. The proof that $\mathcal{C}(P, \text{Itanium})$ is a proper subset of $\mathcal{C}(P, \text{Itanium}_\text{B})$ is constructive. Given an Itanium computation for $P$ we construct "views" for each processor in $P$ and prove that these views constitute a collection of Itanium$_B$-verifying sequences. Then we provide a computation that satisfies Itanium$_B$ but not Itanium to establish strict inequality.

Most of the Itanium consistency rules are applicable to a subset of the instructions in the computation or operations in the visibility order. For example, each load instruction ld[.acq]in a visibility order is either local or non-local and if non-local R(ld[.acq]) is either preceded in the visibility order by a store to the same location or not. Hence, only one of the three Read Value Rules (RV1), (RV2), or (RV3) is applicable to ld[.acq], and if the visibility order is valid the others hold vacuously for ld[.acq]. In our proofs, we note which rule *applies* in this sense, and prove whatever is required for that case, without repeatedly noting that the other rules hold vacuously.

Let $C$ be any computation of $P$ that satisfies Itanium, and let V be a visibility order for $C$ that satisfies all the requirements of Itanium. Create an altered visibility order $\mathcal{S}$(V) for $C$ as follows:

$\mathcal{S}$(V)**:** For every processor $p$, for every store instruction st[.rel] by processor $p$, let ld[.acq]$_1$, ld[.acq]$_2$, ..., ld[.acq]$_k$ be the local loads that are causally related to st[.rel] listed in the order in which their corresponding read operations occur in V, if $k \geq 1$, move R(ld[.acq]$_1$), (ld[.acq]$_2$), ..., (ld[.acq]$_k$) to immediately follow RV$_p$(st[.rel]) in V.

**Lemma 3.1** *$\mathcal{S}$(V) satisfies the Read Value Rules and only rules (RV2) and (RV3) apply to read operations in $\mathcal{S}$(V).*

**Proof:** V satisfies (RV1), (RV2) and (RV3) by definition. Since only local read operations are moved to form $\mathcal{S}$(V) from V, in $\mathcal{S}$(V) any non-local load still satisfies (RV2) or (RV3). For a local load ld[.acq], rule (RV1) applies to ld[.acq] in V. But then operation R(ld[.acq]) is moved so that ld[.acq] is no longer local and instead satisfies (RV2) in $\mathcal{S}$(V). ■

Given a visibility order V for a computation $C$, create a sequence $S_p(V)$ for each $p \in P$ as follows.

- $S_p(V) \leftarrow$V.
- Delete all LV(st[.rel]) from $S_p(V)$.
- Delete all R(ld[.acq]) where Proc(ld[.acq])$\neq p$ from $S_p(V)$.
- Delete each RV$_q$(st[.rel]) where $q \neq p$ from $S_p(V)$.
- Replace each remaining operation with its corresponding instruction.

**Lemma 3.2** *For any computation $C$ with visibility order V, which satisfies Itanium, the sequences $S_p(\mathcal{S}(\text{V}))$ $\forall p \in P$ comprise a set of* Itanium$_B$*-verifying sequences for $C$.*

**Proof:**

Validity: By Lemma 3.1 $\mathcal{S}$(V) satisfies the Read Value rule and only (RV2) and (RV3) apply to load instructions. Thus for any R(ld$_p(x)$) (or R(ld.acq$_p(x)$)) in $\mathcal{S}$(V), its most recent preceding RV$_p$(st$_q(x,v)$) or RV$_p$(st.rel$_q(x,v)$) must satisfy $v$ = RdVal(R(ld$_p(x)$)) (or $v$ =RdVal(R(ld.acq$_p(x)$))). This property is maintained in the construction of $S_p$ for the corresponding load and store instructions. Thus, for each sequence $S_p$, for each load instruction ld[.acq] in $S_p$, the store instruction that is causally related to ld[.acq] is the most recent preceding store to the same location.

Acquire-B Order: V satisfies (ACQ). Only local read operations are moved to form $\mathcal{S}$(V), so (ACQ) is satisfied in $\mathcal{S}$(V) for non-local instructions. Since all the ld.acq instructions by $p$ are in the sequence $S_p(\mathcal{S}(\text{V}))$ in the same position relative to $p$'s other instructions as the corresponding operations are in $\mathcal{S}$(V), and since all foreign loads correspond to non-local read operations, these sequences extend Acquire-B order.

Release Order: V maintains (REL) order. Let i $\xrightarrow{prog}$ *st.rel* in the program for processor $p$. If $i$ is a st[.rel] then RV$_q(i)$ $\xrightarrow{V}$ RV$_q$(st.rel) by (REL), so RV$_q(i)$ $\xrightarrow{\mathcal{S}(\text{V})}$ RV$_q$(st.rel) since neither operation moves in the construction of $\mathcal{S}$(V). Therefore, $i$ $\xrightarrow{S_q(\mathcal{S}(\text{V}))}$ st.rel for each processor $q$. If $i$ is a ld[.acq] then $i$ appears only in $S_p(\mathcal{S}(\text{V}))$ (not in any $S_q(\mathcal{S}(\text{V}))$ for $q \neq p$) and again by (REL) R($i$) $\xrightarrow{V}$ RV$_p$(st.rel). Provided R($i$) is not a local read, again neither operation moves in the construction of $\mathcal{S}$(V) and thus i $\xrightarrow{S_p(\mathcal{S}(\text{V}))}$ st.rel.

The only remaining case is if $i$ is a local ld[.acq], and there is a st.rel operation following R($i$) in V but preceding it in $\mathcal{S}$(V). Let $j$ be the store (necessarily by $p$) that is causally related to $i$. Suppose there is an st.rel also by $p$ satisfying LV($j$) $\xrightarrow{V}$ R($i$) $\xrightarrow{V}$ RV$_p$(st.rel) $\xrightarrow{V}$ RV$_p(j)$. By (REL) st.rel $\xrightarrow{prog}$ $j$, and by memory data dependence, $j$ $\xrightarrow{prog}$ $i$. Hence st.rel $\xrightarrow{prog}$ $i$, so in this case there is no release order to maintain between st.rel and $i$.

Same Memory Order: Let $i \xrightarrow{prog} j$ and $i, j \in I(C)|x$. Observe that $\mathcal{S}$(V) satisfies (MD:WAW) and (WO) because the conversion to $\mathcal{S}$(V) does not move any LV or RV operations. $\mathcal{S}$(V) maintains (MD:RAW) order because moving R(ld[.acq]) later in the sequence does not alter the relative order of a read operation and a write operation that precedes it. Since the creation of $S_p(\mathcal{S}(\text{V}))$ retains only the instructions corresponding to RV and R operations by $p$, the Same Memory Order holds when $j \in I(C)|w$. Although $\mathcal{S}$(V) does not maintain (MD:WAR), the (COH) rule of V ensures that R(ld[.acq]) $\xrightarrow{\mathcal{S}(\text{V})}$ RV$_p$(st[.rel]) when

ld[.acq] $\xrightarrow{prog}$ st[.rel]. Since R(ld[.acq]) and $\text{RV}_p$(st[.rel]) are the operations that are converted to ld[.acq] and st[.rel] instructions respectively. This gives the Same Memory Order when $i \in I(C)|w$.

Now consider the final case of Same Memory Order when $i \in I(C)|acq$. Consider the operations to the same location when a local R(ld.acq) operation is moved in the construction of $\mathcal{S}$(V). By construction, any operations to the same location that correspond to load instructions are also moved to after the R(ld.acq) operation; any LV operations will be deleted in the final conversion; and any $\text{RV}_p$ operations must correspond to a write operation st[.rel]' that is st[.rel]' $\xrightarrow{prog}$ st[.rel] $\xrightarrow{prog}$ ld[.acq], where st[.rel] is causally related to ld[.acq]. This is because (COH) ensures that LV(st[.rel]') $\xrightarrow{V}$ LV(st[.rel]) $\xrightarrow{V}$ R(ld[.acq]) $\xrightarrow{V}$ $\text{RV}_p$(st[.rel]') $\xrightarrow{V}$ $\text{RV}_p$(st[.rel]). Thus, Same Memory order is assured when $i \in I(C)|acq$. Hence, $S_p$ satisfies Same Memory Order for every $p$.

Same Memory Agreement: V satisfies (COH) by definition, and $\mathcal{S}$(V) maintains the order of V for all RV operations. The order of the all store instructions in $S_p(\mathcal{S}(\text{V}))$ is the same as the corresponding $\text{RV}_p$ operations in $\mathcal{S}$(V), ensuring Same Memory Agreement.

Release Agreement: The (WBR) rule ensures that all RV operations of a st.rel instruction are together and there is only one F(fence) operation in V. $\mathcal{S}$(V) does not affect the relative order of these operations. The order of the all store instructions in $S_p(\mathcal{S}(\text{V}))$ is the same as the corresponding $\text{RV}_p$ operations in $\mathcal{S}$(V) ensuring Release Agreement.

Release to Store Agreement: V satisfies (WBR) and (WO) and all write and fence operations are in the same order in $\mathcal{S}$(V) and V. If st.rel $\xrightarrow{S_p(\mathcal{S}(\text{V}))}$ $\text{st}_p(\cdot,\cdot)$, then $\text{RV}_p$(st.rel) $\xrightarrow{\mathcal{S}(\text{V})}$ $\text{RV}_p(\text{st}_p(\cdot,\cdot))$. So, by (WBR) and (WO), $\text{RV}_q$(st.rel) $\xrightarrow{\mathcal{S}(\text{V})}$ $\text{RV}_p(\text{st}_p(\cdot,\cdot))$. $\xrightarrow{\mathcal{S}(\text{V})}$ $\text{RV}_q(\text{st}_p(\cdot,\cdot))$ implying st.rel $\xrightarrow{S_q(\mathcal{S}(\text{V}))}$ $\text{st}_p(\cdot,\cdot)$ for any $q$ and thus ensuring Release to Store Agreement.

Cycle Free Agreement: (by contradiction) Assume there is a cycle of $i_1, i_2 \ldots i_k \in I(C)|w$ where $i_1 \in I(C)|p_1, i_2 \in I(C)|p_2 \ldots i_k \in I(C)|p_k$ and $k \leq n$ such that: $i_k \xrightarrow{S_1} i_1$, and $i_1 \xrightarrow{S_2} i_2$, and $i_2 \xrightarrow{S_3} i_3$ ... and $i_{k-1} \xrightarrow{S_k} i_k$. The relation $i_k \xrightarrow{S_1} i_1$ corresponds to $\text{RV}_1(i_k) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_1(i_1)$ and from the (WO) rule $\text{RV}_k(i_k) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_1(i_k) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_1(i_1)$ thus $\text{RV}_k(i_k) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_1(i_1)$. Similarly, the relation $i_{j-1} \xrightarrow{S_j} i_j$ corresponds to $\text{RV}_j(i_{j-1}) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_j(i_j)$ and from the (WO) rule, $\text{RV}_{j-1}(i_{j-1}) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_j(i_{j-1}) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_j(i_j)$ implying $\text{RV}_{j-1}(i_{j-1}) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_j(i_j)$. Combining these, gives the cycle $\text{RV}_k(i_k) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_1(i_1) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_2(i_2) \ldots \xrightarrow{\mathcal{S}(\text{V})}$ $\text{RV}_{k-1}(i_{k-1}) \xrightarrow{\mathcal{S}(\text{V})} \text{RV}_k(i_k)$. But this cycle cannot exist because $\mathcal{S}$(V) is a total order, providing the required contradiction. ∎

Consider the following computation:

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

It is straightforward to confirm that the following sequence are Itanium$_B$-verifying sequences for Computation 1:

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

As a notational convenience, we write $a \xrightarrow{(IR)} b$, where (IR) is any of the Itanium rules given in Subsection 2.3, to mean that (IR) requires $a \xrightarrow{V} b$. The orders that must be maintained by any Itanium visibility sequence that satisfies the Itaniumrules contains the following cycle: $\text{R}(3 \leftarrow \text{ld}_p(x)) \xrightarrow{MD:WAR}$ $\text{LV}(\text{st}_p(x,2)) \xrightarrow{(MD:RAW)} \text{R}(2 \leftarrow \text{ld.acq}_p(x)) \xrightarrow{(ACQ)} \text{LV}(\text{st}_p(y,4)) \xrightarrow{(WO)} \text{RV}_p(\text{st}_p(y,4)) \xrightarrow{(WO)} \text{RV}_q(\text{st}_p(y,4))$ $\xrightarrow{(RV2)} \text{R}(4 \leftarrow \text{ld.acq}_q(y)) \xrightarrow{(ACQ)} \text{LV}(\text{st}_q(x,3)) \xrightarrow{(WO)} \text{RV}_q(\text{st}_q(x,3)) \xrightarrow{(WO)} \text{RV}_p(\text{st}_q(x,3)) \xrightarrow{(RV2)} \text{R}(3 \leftarrow \text{ld}_p(x))$. Thus, Computation 1 does not satisfy Itanium because it does not have a Itanium visibility sequence.

**Theorem 3.3** Itanium *is strictly stronger than* Itanium$_B$.

**Proof:** By Lemma 3.2, any Itanium-based computation that satisfies Itanium also satisfies Itanium$_B$, and Computation 1 is an Itanium-based computation that satisfies Itanium$_B$ but not Itanium. ■

### 3.2 Itanium$_A$ computations satisfy Itanium

Let $P$ be any Itanium-based multiprogram. The proof that $\mathcal{C}(P, \text{Itanium}_\text{A})$ is a proper subset of $\mathcal{C}(P, \text{Itanium})$ is also constructive. Given an Itanium$_A$ computation $C$ for $P$, we construct a visibility order V for $C$ and prove that it satisfies all the requirements of Itanium. To show strict inequality, a computation is provided that satisfies Itanium but not Itanium$_A$.

For each $p \in P$, let $S_p$ be the sequence for processor $p$ in the set of Itanium$_A$-verifying sequences for $C$. First, we create a visibility sequence V($S_p$) for each $p \in P$. Then, these are merged to form a single visibility order.

From each $S_p$ create a sequence V($S_p$) as follows:

- replace each load instruction ld[.acq] by R(ld[.acq]).
- replace each store instruction st[.rel] such that Proc(st[.rel])= $p$ by the two contiguous operations: LV(st[.rel]), RV$_p$(st[.rel]).
- replace each remaining store instruction st[.rel] (Proc(st[.rel])= $q \neq p$) by RV$_p$(st[.rel]).
- replace each fence fence by F(fence).

Each instruction in each $S_p$ *corresponds* to the operation(s) that replaced it in V($S_p$).

Next, the sequences V($S_p$) are merged into one visibility order V as follows. Place a pointer $\downarrow_p$ at the first operation of each V($S_p$). Initially, each $\downarrow_p$ is *unblocked*. Advancing $\downarrow_p$ moves it to the next operation in V($S_p$). A pointer $\downarrow_p$ becomes null when it is advanced beyond the last operation in V($S_p$). V is initially an empty sequence.

While there is a not-null pointer $\downarrow_p$, choose an unblocked $\downarrow_p$:

1. If ($\downarrow_p = o$ = R(ld) or R(ld.acq) or LV(st) or LV(st.rel) or RV$_p$(st$_p$)) then:
   - Append $o$ to V and advance $\downarrow_p$
   - If $o$ = RV$_p$(st$_p$) then unblock any $\downarrow_q$ blocked on RV$_q$(st$_p$)
2. If ($\downarrow_p = o$ = RV$_p$(st$_q$)) then:
   - If RV$_q$(st$_q$) is in V then append RV$_p$(st$_q$) to V and advance $\downarrow_p$
   - else block $\downarrow_p$ on $o$
3. If $\downarrow_p = o$ = RV$_p$(st.rel$_q$) then:
   - If ($\forall t \in P$, $\downarrow_t$=RV$_t$(st.rel$_q$) then:
     - $\forall t \in P$ append RV$_t$(st.rel$_q$) to V starting with RV$_q$(st.rel$_q$)
     - $\forall t \in P$ advance and unblock $\downarrow_t$
   - else block $\downarrow_p$ on $o$
4. If $\downarrow_p = o$ = F(fence) then append $o$ to V and advance $\downarrow_p$

First, we show that this merge procedure generates a visibility order V, which contains all the operations of the sequences V($S_p$), for each $p \in P$. That is, we show it is deadlock-free. Observe that if $\downarrow_p$ is blocked on some $o$, then (1) $o$ = RV$_p$(st[.rel]$_q$), $q \neq p$, where st[.rel]$_q$ is either a st or a st.rel, and (2) there is some $\downarrow_q = o'$ and $o' \xrightarrow{S_q}$ RV$_q$(st[.rel]$_q$) (We say that $\downarrow_p$ *is blocked by* $\downarrow_q$).

**Lemma 3.4** *In the merge procedure if there is at least one not-null pointer, then there is at least one unblocked, not-null pointer.*

**Proof:** Assume for the sake of obtaining a contradiction that the merge is not complete (there is at least one not-null pointer) and all not-null pointers are blocked. Since no pointer is blocked by itself, all not-null pointers are blocked if each is participating in a 'blocked by' cycle and each 'blocked by' cycle has length $\geq 2$. The following claim shows that 'blocked by' cycles cannot be a result of st.rel instructions.

> **Claim 3.5** *In every 'blocked by' cycle there does not exist a pointer $\downarrow_p = RV_p(st.rel_t)$ for any $t \in P$.*
> **Proof:** If there is such a pointer $\downarrow_p = \mathrm{RV}_p(\mathrm{st.rel}_t)$, then since $p$ is in some 'blocked by' cycle, there must be some $q \neq p$ such that $\downarrow_q$ is blocked by $\downarrow_p$. So, one of the following cases apply:
>
> - $\downarrow_q = \mathrm{RV}_q(\mathrm{st}_p)$ and $\mathrm{RV}_p(\mathrm{st.rel}_t) \xrightarrow{S_p} \mathrm{RV}_p(\mathrm{st}_p)$: By release to store agreement we must also have $\mathrm{RV}_q(\mathrm{st.rel}_t) \xrightarrow{S_q} \mathrm{RV}_q(\mathrm{st}_p)$.
> - $\downarrow_q = \mathrm{RV}_q(\mathrm{st.rel}_r)$ and $\mathrm{RV}_p(\mathrm{st.rel}_t) \xrightarrow{S_p} \mathrm{RV}_p(\mathrm{st.rel}_r)$: By release agreement we have $\mathrm{RV}_q(\mathrm{st.rel}_t) \xrightarrow{S_q} \mathrm{RV}_q(\mathrm{st.rel}_r)$.
>
> In either case, $\downarrow_q$ must have advanced beyond $\mathrm{RV}_q(\mathrm{st.rel}_t)$. However by construction, $\downarrow_q$ could only make this advance if in the same step $\downarrow_p$ advances past $\mathrm{RV}_p(\mathrm{st.rel}_t)$, contradicting the position of $\downarrow_p$. ■

It remains to consider non-release stores, or every $\downarrow_p$ that is blocked at some $\mathrm{RV}_p(\mathrm{st}_q)$ for some $q \neq p$ and $p$ and $q$ are in the same 'blocked by' cycle, say $c$. It must be the case that $\forall p_i$ in cycle $c$, $\downarrow_{p_i} = \mathrm{RV}_{p_i}(\mathrm{st}_{p_{(i+1) mod\ k}})$ and $\mathrm{st}_{p_{(i+1) mod\ k}} \xrightarrow{S_{p_i}} \mathrm{st}_{p_i}$, contradicting the cycle free agreement property and proving the lemma. ■

Now that we have shown the merge procedure adds all operations to V, we show next that V satisfies Itanium.

**Lemma 3.6** *Each V($S_p$) satisfies (RV1), (RV2), and (RV3).*

**Proof:** Each $S_p$ is valid. That is, each ld or ld.acq instruction returns the value of the most recent preceding st or st.rel to the same shared variable, or the initial value if no such st or st.rel exists. Thus, the construction ensures that in V($S_p$), each R(ld[.acq]) is preceded by $\mathrm{RV}_p$(st[.rel]) where st[.rel] is the causally related instruction to ld[.acq] and there are no operations $\mathrm{RV}_p$(st[.rel]) between them, where st[.rel] is to the same variable as st[.rel]. ■

**Observation 3.7** *The merge of the V($S_p$) sequences, $\forall p \in P$, ensures the write order rule (WO) in V.*

**Observation 3.8** *The merge of the V($S_p$) sequences, $\forall p \in P$, ensures the write back release rule (WBR) in V.*

**Lemma 3.9** *The merge of the V($S_p$) sequences, $\forall p \in P$, ensures the program order rules (ACQ), (REL) and (FEN) in V.*

**Proof:** Each $S_p$ maintains Strong Acquire and Release parts of the Strong Orderable Order. The merge procedure does not advance $\downarrow_p$ until after an operation corresponding to an acquire instruction is in V yielding (ACQ). The merge procedure does not unblock $\downarrow_p$ on an operation corresponding to a release before all preceding operations are in V yielding (REL). Since each $S_p$ maintains the Strong Acquire part of the Strong Orderable Order, the merge procedure does not place a F(fence) operation on V until after operations that correspond to instructions that precede the fence in $S_p$. Since each $S_p$ maintains the Release parts of the Strong Orderable Order, the merge does not advance $\downarrow_p$ until after the F(fence) is in V. This yields (FEN).

■

**Lemma 3.10** *The merge of the V($S_p$) sequences, $\forall p \in P$, maintains the memory data dependence rules (MD:RAW), (MD:WAR) and (MD:WAW), and the coherence rule (COH).*

**Proof:** Each $S_p$ maintains the Same Memory part of the Strong Orderable Order. The merge ensures that the corresponding R and LV and $\text{RV}_p$ operations are placed in this order for all processors, $p$, yielding (MD:RAW) (MD:WAR) and (MD:WAW).

All $S_p$ maintain the Same Memory agreement property on store instructions. The merge procedure places all RV operations of store instructions and on the same variable in V in the same order as $S_p$ for all $p$, yielding (COH) . ■

**Lemma 3.11** *V is an* Itanium *verifying visibility order.*

**Proof:** By Lemma 3.4, V contains all the operations of the given computation. By Observation 3.7, V maintains (WO). By Lemma 3.9, V maintains (ACQ), (REL) and (FEN). By Lemma 3.10, V maintains (MD:RAW), (MD:WAR), (MD:WAW) and (COH). By Observation 3.8, V maintains (WO).

By Lemma 3.6 each V($S_p$) satisfies (RV1), (RV2), and (RV3); therefore, these are maintained in V. ■

Computation 2 satisfies Itanium consistency but not Itanium$_A$ consistency.

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

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

The Itanium$_A$ sequence, $S_p$, must extend: $\text{st}_q(y,4) \xrightarrow{valid} 4 \leftarrow \text{ld.acq}_p(y) \xrightarrow{sacq} \text{st}_p(x,5) \xrightarrow{ord} \text{st.rel}_p(z,2)$. The sequence $(I(C)|q \cup I(C)|w, S_q)$ must extend: $\text{st}_q(x,3) \xrightarrow{same\ memory} 3 \leftarrow \text{ld.acq}_q(x) \xrightarrow{strong\ acquire} \text{st}_q(y,4) \xrightarrow{cycle\ free} \text{st}_p(x,5) \xrightarrow{release} \text{st.rel}_p(z,2) \xrightarrow{valid} 2 \leftarrow \text{ld.acq}_q(z) \xrightarrow{strong\ acquire} 3 \leftarrow \text{ld}_q(x)$. This makes the final $3 \leftarrow \text{ld}_q(x)$ invalid.

**Theorem 3.12** Itanium$_A$ *memory consistency is strictly stronger than* Itanium *memory consistency.*

**Proof:** By Lemma 3.11, any Itanium-based computation that satisfies Itanium$_A$ also satisfies Itanium, and Computation 2 is an Itanium-based computation that satisfies Itanium but not Itanium$_A$. ■

# 4 Comparing Alternative Acquire Orders

Itanium$_B$ and Itanium$_A$ 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$_A$ yet still strictly stronger than Itanium.

## 4.1 Acquire order definitions

Define the *write-before-read* relation $(I(C), \xrightarrow{wbr})$ by: $i_1 \xrightarrow{wbr} i_2$ if, for some shared variable $x$, $i_1 \in I(C)|x|w$ and $i_2 \in I(C)|x|r$ and $i_2$ reads the same value written by $i_1$.

I addition to Acquire A and Acquire B defined in Subsection 2.2, define to additional acquire orders as follows. Let $i, j \in I(C)$ such that $i \xrightarrow{prog} j$.

**Acquire C:** $i_1 \in I(C)|acq$ and $i_2$ is a non-domestic load.

**Acquire D:** $i_1 \xrightarrow{wbr} i_3 \xrightarrow{prog} i_2$ and $i_3\ \in I(C)|acq$

These two partial orders give rise to two new definitions for Itanium consistency, in particular Itanium$_C$ (Definition 2.1 with $R$ = Acquire C) and Itanium$_D$ (Definition 2.1 with $R$ = Acquire D).

More variants of the Itanium memory consistency model are defined by combining the four basic acquire orders based either on intersection or conjunction as follows. Let $\gamma, \beta \in \{A, B, C, D\}$.

**Intersection:** A computation $C$ satisfies Itanium$_{\gamma\cap\beta}$ if $C$ satisfies Itanium$_\gamma$ and Itanium$_\beta$.

**Conjunction:** A computation $C$ satisfies Itanium$_{\gamma\wedge\beta}$ if $C$ satisfies Itanium$_{\gamma\cap\beta}$ and there is a set of Itanium$_\gamma$-verifying sequences for $C$ that are also Itanium$_\beta$-verifying sequences for $C$.

Note that the models Itanium$_{\gamma\cap\beta}$ allow the Itanium$_\gamma$-verifying sequences for $C$ to be different from Itanium$_\beta$-verifying sequences for $C$. Hence, Itanium$_{\gamma\wedge\beta}$ is stronger than Itanium$_{\gamma\cap\beta}$.

Since Itanium$_A$ is stronger than each of Itanium$_B$, Itanium$_C$, and Itanium$_D$, this introduces six new and distinct Itanium memory consistency models: Itanium$_{C\cap B}$, Itanium$_{C\cap D}$ Itanium$_{D\cap B}$ Itanium$_{C\wedge B}$, Itanium$_{C\wedge D}$ and Itanium$_{D\wedge B}$. Observe that Itanium$_A$ is also stronger than each of the models Itanium$_{C\wedge B}$, Itanium$_{C\wedge D}$ and Itanium$_{D\wedge B}$.

Figure 1 shows the relative strengths of these models. The next two subsections present computations and proofs that establish the relationships of Figure 1. Table 1 summarizes the arguments used for incomparable models in Figure 1.

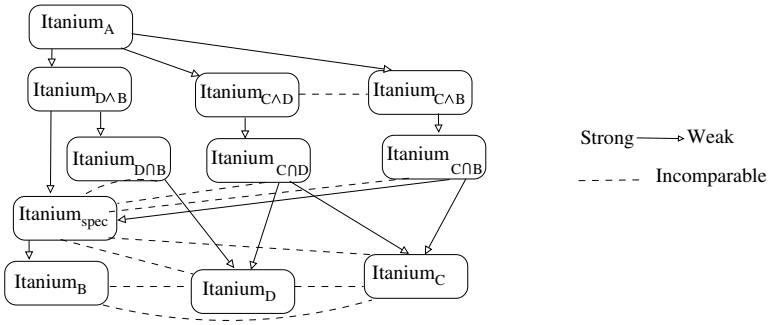


Figure 1: Relative Strength of Various Systems

## 4.2 Incomparable consistency models

Here also we abbreviate our notation: $a \xrightarrow{valid} b$ 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 Subsection 3.1 was shown to satisfy Itanium$_B$ but not Itanium (consequently, Itanium$_A$). 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$_{C\wedge D}$); 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 $S_p$ has a 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$_D$ as shown by the following verifying sequences:

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

Since Computation 1 satisfies both Itanium$_B$ 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 $S_p$ has a 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** $\begin{cases} p : \mathrm{st}(x,1)\ \ 2 \leftarrow \mathrm{ld.acq}(y)\ \ 1 \leftarrow \mathrm{ld}(x) \\ q : 1 \leftarrow \mathrm{ld}(x)\ \ \mathrm{st}(x,3)\ \ \mathrm{st.rel}(y,2) \end{cases}$

Computation 3 does not satisfy Itanium (consequently, Itanium$_A$) because V must extend: LV(st$_p(x,1)$) $\xrightarrow{(WO)}$ RV$_p$(st$_p(x,1)$) $\xrightarrow{(WO)}$ RV$_q$(st$_p(x,1)$) $\xrightarrow{(RV2)}$ R$(1 \leftarrow \mathrm{ld}_q(x))$ $\xrightarrow{(WAR)}$ LV(st$_q(x,3)$) $\xrightarrow{(WO)}$ RV$_q$(st$_q(x,3)$) $\xrightarrow{(WO)}$ RV$_p$(st$_q(x,3)$) $\xrightarrow{(REL)}$ RV$_p$(st.rel$_q(y,2)$) $\xrightarrow{(RV2)}$ R$(2 \leftarrow \mathrm{ld.acq}_p(y))$ $\xrightarrow{(ACQ)}$ R$(1 \leftarrow \mathrm{ld}_p(x))$. This, however, ensures that the R$(1 \leftarrow \mathrm{ld}_p(x))$ does not satisfy any of (RV1), (RV2), or (RV3).

Computation 3 does not satisfy Itanium$_B$ (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 st$_p(x,1)$ $\xrightarrow{valid}$ $1 \leftarrow \mathrm{ld}_q(x)$ $\xrightarrow{Same\ Mem}$ st$_q(x,3)$ $\xrightarrow{Release}$ st.rel$_q(y,2)$. Since st$_p(x,1)$ $\xrightarrow{S_q}$ st$_q(x,3)$, by the Same Memory Agreement we must have st$_p(x,1)$ $\xrightarrow{S_p}$ st$_q(x,3)$. So, sequence $S_p$ must maintain st$_p(x,1)$ $\xrightarrow{Same\ Mem\ Agree}$ st$_q(x,3)$ $\xrightarrow{Release}$ st.rel$_q(y,2)$ $\xrightarrow{valid}$ $2 \leftarrow \mathrm{ld.acq}_q(y)$ $\xrightarrow{Acquire\ B}$ $1 \leftarrow \mathrm{ld}_p(x)$, yielding an invalid $1 \leftarrow \mathrm{ld}_p(x)$.

Computation 3 satisfies Itanium$_{C\wedge D}$ (consequently, it also satisfies Itanium$_C$, Itanium$_D$, and Itanium$_{C\cap D}$) as shown by the following sequences:

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

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

Computation 4 does not satisfy Itanium nor Itanium$_A$. Had it been the case, V must extend: LV(st$_p(y,2)$) $\xrightarrow{(WO)}$ RV$_p$(st$_p(y,2)$) $\xrightarrow{(WO)}$ RV$_q$(st$_p(y,2)$) $\xrightarrow{(RV2)}$ R$(2 \leftarrow \mathrm{ld}_q(y))$ $\xrightarrow{(RAW)}$ LV(st$_q(y,4)$) $\xrightarrow{(WO)}$ RV$_q$(st$_q(y,4)$) $\xrightarrow{(WO)}$ RV$_p$(st$_q(y,4)$) $\xrightarrow{(REL)}$ RV$_p$(st.rel$_q(x,5)$) $\xrightarrow{(RV2)}$ R$(5 \leftarrow \mathrm{ld}_p(x))$ $\xrightarrow{(REL)}$ LV(st.rel$_q(x,1)$) $\xrightarrow{(RV1\ or\ 2)}$ R$(1 \leftarrow \mathrm{ld.acq}_p(x))$ $\xrightarrow{(ACQ)}$ R$(2 \leftarrow \mathrm{ld}_p(y))$. However this means that the final 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$_B$, and Itanium$_{C\cap B}$) as shown by the following sequences:

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

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

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

However, Computation 5 does not satisfy Itanium$_C$ nor Itanium$_D$. The sequence $S_q$ must extend st$_p(y,2)$ $\xrightarrow{valid}$ $2 \leftarrow \mathrm{ld}_q(y)$ $\xrightarrow{Same\ Mem}$ st$_q(y,3)$. Thus, the Same Memory Agreement requires that st$_p(y,2)$ $\longrightarrow$ st$_q(y,3)$ in all sequences. So, the sequence $S_p$ must extend st$_p(x,1)$ $\xrightarrow{Same\ Mem}$ $1 \leftarrow \mathrm{ld.acq}_p(x)$ $\xrightarrow{Acquire\ C}$ st$_p(y,2)$ $\xrightarrow{Same\ Mem\ Agree}$ st$_q(y,3)$ $\xrightarrow{Release}$ st.rel$_q(x,4)$ or st$_p(x,1)$ $\xrightarrow{Acquire\ D}$ st$_p(y,2)$ $\xrightarrow{Same\ Mem\ Agree}$ st$_q(y,3)$ $\xrightarrow{Release}$ st.rel$_q(x,4)$ . Thus, the Same Memory Agreement requires that st$_p(x,1)$ $\longrightarrow$ st.rel$_q(x,4)$

in all sequences. Observe that the final part of Same Memory Order requires that sequence $S_t$ maintains $4 \leftarrow$ld.acq$(x)$ before $1 \leftarrow$ld$(x)$ and thus it cannot be valid.

Consequently, Computation 5 does not satisfy any of Itanium$_{C\wedge D}$, Itanium$_{C\wedge B}$, Itanium$_{D\wedge B}$, Itanium$_{D\cap B}$, Itanium$_{C\cap D}$, or Itanium$_{C\cap B}$.

| Computation | Spec | $A$ | $B$ | $C$ | $D$ | $C \cap B$ | $C \cap D$ | $D \cap B$ | $C \wedge B$ | $C \wedge D$ | $D \wedge B$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | × | × | √ | × | √ | × | × | √ | × | × | × |
| 3 | × | × | × | √ | √ | × | √ | × | × | √ | × |
| 4 | × | × | √ | √ | × | √ | × | × | √ | × | × |
| 5 | √ | | √ | × | × | × | × | × | × | × | × |

Table 1: Summary of Computation-Model Satisfiability

**Theorem 4.1** Itanium *is incomparable to 1.* Itanium$_C$*, 2.* Itanium$_D$*, 3.* Itanium$_{C\wedge D}$*, 4.* Itanium$_{C\wedge B}$*, 5.* Itanium$_{D\cap B}$*, 6.* Itanium$_{C\cap B}$*, and 7.* Itanium$_{C\cap D}$

**Proof:** 1. Computation 3 does not satisfy Itanium but satisfies Itanium$_C$. Computation 5 does not satisfy Itanium$_C$ but satisfies Itanium. 2. Computation 1 does not Itanium but satisfies Itanium$_D$. Computation 5 does not satisfy Itanium$_D$ but satisfies Itanium. 3. Computation 3 does not satisfy Itanium but satisfies Itanium$_{C\wedge D}$. Computation 5 does not satisfy Itanium$_{C\wedge D}$ but satisfies Itanium. 4. Computation 4 does not satisfy Itanium but satisfies Itanium$_{C\wedge B}$. Computation 5 does not satisfy Itanium$_{C\wedge B}$ but satisfies Itanium. 5. Computation 1 does not satisfy Itanium but satisfies Itanium$_{D\cap B}$. Computation 5 does not satisfy Itanium$_{D\cap B}$ but satisfies Itanium. 6. Computation 4 does not satisfy Itanium but satisfies Itanium$_{C\cap B}$. Computation 5 does not satisfy Itanium$_{C\cap B}$ but satisfies Itanium. 7. Computation 3 does not satisfy Itanium but satisfies Itanium$_{C\cap D}$. Computation 5 does not satisfy Itanium$_{C\cap D}$ but satisfies Itanium. ■

**Theorem 4.2** Itanium$_B$ *is incomparable to* Itanium$_C$ *and* Itanium$_D$*.*

**Proof:** Computation 3 does not satisfy Itanium$_B$ but satisfies Itanium$_C$. Computation 1 does not satisfy Itanium$_C$ but satisfies Itanium$_B$. Computation 3 does not satisfy Itanium$_B$ but satisfies Itanium$_D$. Computation 5 does not satisfy Itanium$_D$ but satisfies Itanium$_B$. ■

**Theorem 4.3** Itanium$_C$ *is incomparable to* Itanium$_D$*.*

**Proof:** Computation 4 does not satisfy Itanium$_D$ but satisfies Itanium$_C$. Computation 1 does not satisfy Itanium$_C$ but satisfies Itanium$_D$. ■

**Theorem 4.4** Itanium$_{C\wedge D}$ *is incomparable to* Itanium$_{C\wedge B}$*.*

**Proof:** Computation 3 does not satisfy Itanium$_{C\wedge B}$ but satisfies Itanium$_{C\wedge D}$. Computation 4 does not satisfy Itanium$_{C\wedge D}$ but satisfies Itanium$_{C\wedge B}$. ■

### 4.3 A model strictly weaker than Itanium$_A$ and stronger than Itanium

We prove that Itanium$_{D\wedge B}$ is strictly stronger than Itanium. The proof mimics that of Theorem 3.12. Given $C \in \mathcal{C}(P, \text{Itanium}_{\text{D}\wedge\text{B}})$, we construct a visibility order V for $C$ and prove that it satisfies all the requirements of Itanium. To show strict inequality, a computation is provided that satisfies Itanium but not Itanium$_{D\wedge B}$.

We begin with a set of sequences, $S_p$, for each $p \in P$ that are valid total orders of the operations $I(C)|p \cup I(C)|w$ and meet the consistency requirements and Agreement properties of Itanium$_{D\wedge B}$. From each $S_p$ create a sequence Altered$(S_p)$ as follows: locate each ld.acq that has instructions $i$ such that ld.acq$\xrightarrow{prog} i$ but $i \xrightarrow{S_p}$ ld.acq. Move each of these ld.acq instructions to immediately precede the earliest such $i$ in $S_p$. From each Altered$(S_p)$ create a sequence V(Altered$(S_p)$) as follows:

- replace each ld[.acq] instruction by R(ld[.acq]).

- replace each st[.rel] instruction such that Proc(st[.rel])= $p$ by two contiguous operations: LV(st[.rel]), $\text{RV}_p$(st[.rel]).
- replace each remaining st[.rel] instruction (Proc(st[.rel])= $q \neq p$) by $\text{RV}_p$(st[.rel]).
- replace each fence by F(fence).

Use the same merge algorithm as was used in Subsection 3.2 to merge the V(Altered($S_p$)) sequences and call the resulting sequence V. By Lemma 3.4 the merge does not deadlock.

**Lemma 4.5** *Each V(*Altered($S_p$)*) satisfies (RV1), (RV2), and (RV3).*

**Proof:** Each $S_p$ is valid. That is, each ld[.acq] instruction returns the value of the most recent preceding st[.rel] to the same shared variable, or the initial value if no such st[.rel] exists. The validity of the ld[.acq] instructions that were not moved are unaffected. For any ld.acq L that was moved, Acquire B Order ensures that L is domestic and the Acquire D Order ensures that the instruction $i$ that caused L to move will follow its causally related instruction T in both program order and in $S_p$. The Same Memory Order ensures that L is between T and the next st[.rel] instruction to the same variable T'. That is, T $\xrightarrow{S_p}$ $i$ $\xrightarrow{S_p}$ L $\xrightarrow{S_p}$ T'. The altered sequence simply moved L forward of $i$: T $\xrightarrow{\text{Altered}(S_p)}$ L $\xrightarrow{\text{Altered}(S_p)}$ $i$ $\xrightarrow{\text{Altered}(S_p)}$ T'. Thus L is valid in Altered($S_p$).

The construction ensures that in V($S_p$), each R(ld[.acq]) is preceded by $\text{RV}_p$(st[.rel]) where st[.rel] is the causally related instruction to ld[.acq] and there are no operations $\text{RV}_p$(st[.rel]') between them, where st[.rel]' is to the same variable as st[.rel]. ■

**Lemma 4.6** *The merge of the V(*Altered($S_p$)*) sequences, $\forall p \in P$, ensures that (ACQ), (REL) and (FEN) are satisfied in V.*

**Proof:** Each Altered($S_p$) maintains Acquire B, Acquire D, and Release Order. The ld.acq instructions that are moved to create Altered($S_p$) are those that maintain only Acquire D Order and not Acquire B Order. This movement ensures that $i$ $\xrightarrow{\text{Altered}(S_p)}$ ld.acq for every instruction $i$ where $i$ $\xrightarrow{prog}$ ld.acq. Thus, the merge procedure does not advance $\downarrow_p$ until after an operation corresponding to an ld.acq instruction is in V yielding (ACQ). The merge procedure does not unblock $\downarrow_p$ on an operation corresponding to a st.rel before all preceding operations are in V yielding (REL). Since each $S_p$ maintains Acquire A Order, the merge procedure does not place a F(fence) operation on V until after the operations that correspond to instructions that precede the fence in $S_p$. Since each $S_p$ maintains Release Order, the merge does not advance $\downarrow_p$ until after the F(fence) is in V. This yields (FEN).

■

**Lemma 4.7** *V is an* Itanium *verifying visibility order.*

**Proof:** By Lemma 3.4, V contains all the operations of the given computation. By Observation 3.7, V maintains (WO). By Lemma 4.6, V maintains (ACQ), (REL) and (FEN). By Lemma 3.10, V maintains (MD:RAW), (MD:WAR), (MD:WAW) and (COH). By Observation 3.8, V maintains (WO).

By Lemma 4.5 each V(altered($S_p$)) satisfies (RV1), (RV2), and (RV3); therefore, these are maintained in V. ■

**Theorem 4.8** Itanium$_{D\wedge B}$ *memory consistency is strictly stronger than* Itanium *memory consistency.*

**Proof:** By Lemma 4.7, any Itanium-based computation that satisfies Itanium$_{D\wedge B}$ also satisfies Itanium, and Computation 5 is an Itanium-based computation that satisfies Itanium but not Itanium$_{D\wedge B}$. ■

So, Itanium$_{D\wedge B}$ is weaker than Itanium$_A$ but still stronger than Itanium. At present a programmer-centric consistency model that is equivalent to Itanium has not been identified.

# References


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

[2] 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.

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

[4] 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.

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

[6] L. Jackson. Phd thesis: Complete framework for memory consistency with applications to Itanium multiprocessors, In Preparation, 2006.

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

[8] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium memory ordering rules using logic programming and sat. Technical Report UUCS-03-010, University of Utah, 2003.