A Formal Specification of Intel® Itanium® Processor Family Memory Ordering

Application Note

October 2002

Document Number: 251429-001

Contents

Tables

1.0 Introduction

Memory ordering (also called *memory consistency*) is a property of shared-memory multiprocessors in which data can be distributed or replicated in different locations. A common form of such data replication is through memory caches, and a kind of memory ordering, called *cache coherence*, is important when such caches exist. In a system in which more than one processor can access a datum independently (i.e., each through its own cache), cache coherence ensures that processors appear to be accessing a single memory and not a set of separate caches. The requirements of cache coherence are often expressed by specifying the order in which processors perceive memory operations. For example, cache coherence usually specifies that all processors agree on the order of updates to a given cache line. It may also specify that processors see such updates in an order that is consistent with the order in which the processors' programs generated these updates.

Cache coherence is central to the correctness of many multithreaded programs and, for this reason, it has largely the same properties on most shared-memory multiprocessors. Memory ordering can be thought of as an extension of cache coherence. The requirements of cache coherence (see above) apply to memory accesses on a per-cache-line (or per-byte) basis. The memory ordering properties of a platform include cache coherence and additionally characterize the order in which processors may perceive accesses to the entire memory. They may relate a processor's accesses to different memory locations. Some memory ordering properties place requirements on the perceived ordering of accesses to different locations by different processors.

Memory ordering is more subtle and less well understood than cache coherence. Memory ordering properties tend to be processor or platform specific. These properties are not always clearly specified; for example, the memory ordering properties of some platforms can be discovered only by consulting microprocessor and chipset documentation, which often appears in different books that use different terminology and notation. It is difficult to compare even those platforms whose memory ordering properties have been clearly specified as each such platform is usually specified in its own definitional framework.

This document presents a formal and precise specification of the memory ordering properties provided by multiprocessors based on the Intel® Itanium® processor family microprocessors. It does so using a new definitional framework. While the framework was developed specifically for Itanium architecture-based platforms, it was designed to be flexible, and it can express the memory ordering properties of most known multiprocessors.

Before proceeding, we introduce the following terminology. A *memory ordering model* refers to a collection of executions; in general, we consider models that correspond to particular platforms (or types of platforms) and the associated executions are those that could be exhibited by those platforms. The *specification* of a memory ordering model is a definition of a model, typically given by a set of rules that must hold for any execution of the model. A *framework* is a structure for expressing memory ordering specifications. If the specifications of different memory ordering models use a common framework, they share a common structure, making them easier to compare.

Some earlier research defined memory ordering models operationally, referring explicitly to the real times at which certain implementation-specific events occur. While such definitions have value for the architects and implementers of the platforms whose models are being defined, this approach has significant limitations. Models defined in this way may require future platforms supporting the model to maintain compatibility with previous *implementations* instead of the *specifications* that these implementations satisfy. In addition, these operational specifications provide little intuition to help programmers understand the subtle distinctions between memory ordering models.

One alternative to operational definitions is to provide specifications that use *visibility orders*. A model can be specified by requiring that every execution of the model have a visibility order (or set of visibility orders) that *orders* the execution in a certain way. A visibility order is a total (linear) ordering of the memory operations (reads, writes, read-modify-writes, etc.) of the execution being ordered. The word "visibility" refers to the fact that the order indicates how changes to memory (e.g., writes) become *visible* to different observers. Such changes become visible through observations of memory (e.g., reads) that are also ordered. Because specifications based on visibility orders abstract away implementation-specific details and because they are less operational in nature, these definitions have greater value for architects, system implementers, and programmers.

Different memory ordering models can be distinguished by the number and nature of visibility orders that define them. The following items detail some general issues that are considered in the development of such definitions:

- *Consistency with program order.* In general, a definition's visibility order(s) should in some way respect the execution's *program order*; this is a per-processor order (usually a linear order) that reflects the order of the memory operations in the program being executed. One memory ordering model—sequential consistency—requires for each execution a visibility order that is completely consistent with program order (i.e., if A precedes B in program order, A must precede B in the visibility order). All other models allow some reordering; a reordering occurs if operation A precedes B in program order, but B precedes A in the visibility order.
- *Consistency of write visibility.* Some models require all processors to agree on the order of writes to memory; such models are defined with a single visibility order. Examples include sequential consistency, IBM 370*, and the SPARC models TSO, PSO, and RMO*. The single visibility order applies to all memory operations in the execution. Other models allow different processors to observe pairs of writes in different orders. Such models might allow a separate visibility order for each processor; examples include DASH processor consistency, and Intel® architecture memory ordering[1]). The visibility order associated with a processor orders only the operations that could be observed by that processor (e.g., its own reads and all writes).
- *Local bypassing: a special case of write visibility.* Many models allow a processor to appear to observe its own writes earlier than those performed by other processors; this behavior is sometimes called "local bypassing" or simply "bypassing". Models with this property include all those mentioned in the previous item, except sequential consistency and IBM 370. While this feature can be captured by using per-processor visibility orders, this is often not necessary. Instead, special consideration can be made for "local loads" (those that receive bypassed values) and how they are ordered.

The Itanium processor family memory ordering model also allows flexible ordering for a variety of operations. It supports two varieties of loads and stores, and the different varieties have different ordering properties. In addition, Itanium processor family microprocessors support several *memory attributes*, and the memory ordering properties of certain operations are determined, in part, by the attributes of the memory on which they are operating. This document describes a definitional framework that uses a single visibility order and uses this framework to specify the Itanium processor family memory ordering model.[2]

Section 2.0 gives an overview of the definitional framework. Section 3.0 gives a definition of the basic Itanium processor family memory ordering model in the framework. Section 4.0 gives a definition of the semaphore operations supported by the Itanium processor family memory ordering model. Appendix A gives ten sample executions from the *Intel® Itanium® Architecture Software Developer's Manual* and explains why each is allowed or not allowed by the Itanium processor family memory ordering model. Appendix B provides additional examples.

1. This document does not provide a thorough treatment of Intel® architecture memory ordering.
2. Specifications with per-processor visibility orders are also possible, though not discussed in this document. Such specifications do not easily capture the subtleties of the Itanium processor family memory ordering model, however.

1.1 Summary Tables

Section 3.0 and Section 4.0 introduce a large amount of terminology and a number of rules to describe the Itanium processor family memory ordering model. This section gathers a summary of this material into one place for reference. Table 1 summarizes the notation and terminology that appear in the model.

Table 1. Summary of the Notation in the Itanium® Processor Family Memory Ordering Model

Notation	Description	Section
Proc(X)	Processor that performs instruction, access, or operation X.	3.1
Rng(X)	Set of byte addresses on which access or operation X operates.	3.1
RdVal(I), RdVal(I;b)	Value(s) read by instruction or access I (from byte b).	3.1
WrVal(I), WrVal(I;b)	Value(s) written by instruction or access I (to byte b).	3.1
InitVal(b)	Initial value of byte b.	3.1
$Mod_{SEM}(r)$	Value written by semaphore SEM that reads value r.	4.2
A»B	Program Order Operator: A precedes B in program order (A, B are instructions or accesses).	3.1
A>B	Data-Flow Order Operator: A precedes B in data-flow order (A, B are instructions or accesses).	3.1
A→B	Visibility Order Operator: A precedes B in visibility order (A, B are operations).	3.1
R, W	Instruction or access with read or write semantics.	3.1
ACQ, REL, FEN	Instruction or access with acquire, release, or fence ordering semantics.	3.1
LA, UL	Acquire and unordered load instructions or accesses.	3.1
SR, US	Release and unordered store instructions or accesses.	3.1
mf	Memory fence instruction or access.	3.1
sem	Semaphore instructions or accesses.	4.3.1

Table 2 summarizes the operators in the Itanium processor family memory ordering model.

Table 2. A Summary of the Operators in the Itanium® Processor Family Memory Ordering Model

Operator	Description	Section
R(R)	Operation that represents visibility of read access R.	3.2
LV(W), $RV_p(W)$	Operations that represent the visibility of write access W locally and at remote processor p.	3.2
F(FEN)	Fence operation for access FEN.	3.2

Table 3 summarizes the rules in the Itanium processor family memory ordering model.

Table 3. A Summary of the Rules in the Itanium® Processor Family Memory Ordering Model

Rule	Description	Section
(WO)	Write operation ordering.	3.3.1
(ACQ), (REL), (REL)	Program order for acquire, release, and fence ordering semantics.	3.3.2
(MD:RAW), (MD:WAR), (MD:WAW)	Memory-data dependence.	3.3.3
(DF:RAR), (DF:RAW), (DF:WAR), (DF:WAW)	Data-flow dependence.	3.3.4
(COH)	Coherence.	3.3.5
(RV1), (RV2), (RV3)	Read value.	3.3.6
(WBR)	“Total” ordering of releases to WB memory location.	3.3.7.1
(UC1), (UC2), (UC3), (UC4)	Sequentiality of operations on UC memory locations.	3.3.7.2
(NC)	UC memory locations do not bypass.	3.3.7.3
(SM1), (SM2), (SM3)	Semaphore atomicity and behavior.	4.3.1

1.2 Reference Documents

- Adve, Sarita V., and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.
- Gharachorloo, Kourosh, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.
- *IA-32 Intel® Architecture Software Developer’s Manual*, Volume 3: System Programming, (http://developer.intel.com/design/Pentium4/manuals/).
- *Intel® Itanium® Architecture Software Developer’s Manual*, Volume 2: System Architecture, (http://developer.intel.com/design/itanium).
- International Business Machines, *IBM System/370 Principles of Operation*. May 1983. IBM Publication Number GA22-7000-9, File Number S370-01.
- Lamport, Leslie. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
- Peterson, Gary L. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, 12(3):115–116, June 1981.
- SPARC International, Inc. (David L. Weaver and Tom Germond, editors). *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994.

2.0 Overview of the Framework

This chapter gives an overview of the new framework developed for specifying memory ordering models. The framework is characterized by the following features.

A definition in the framework specifies the system executions allowed by the model being defined. An execution is considered to be a set of sequences of instructions, one sequence per processor.[1] Each sequence contains the instructions performed by the associated processor listed in program order. Program order is a total order when restricted to the instructions of a single processor.

Each instruction is specified by its *type* (e.g., unordered load, memory fence), the *range* of memory locations on which it operates (a set of byte addresses), and any *values* that it may have read from or written to that range. Instructions can read values from memory, write values to memory, do both in one operation, or neither read nor write memory.

Instructions are decomposed into *accesses*. For the purposes of this document, there is a one-to-one correspondence between instructions and accesses, and the two terms may be used interchangeably. The program order of accesses is derived from that of instructions.

Accesses are decomposed into *operations*. This is done to allow a finer specification of the ordering properties of instructions. 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 access's operations correspond to different aspects of the access or the visibility of the access at different processors.

Every execution of the model being defined has a single associated *visibility order*. 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. Some of these rules specify when operations must appear in an order consistent with program order (i.e., consistent with the program order of the accesses that compose the execution). Other rules may require some consistency between the ordering of the operations that compose a single memory access. Some rules constrain the values read from memory by load operations.

As noted earlier, each specification in the framework includes a set of rules that constrain the ordering of the operations of an execution. These rules can be classified into the following set of types:

- *Operation order*. Such rules specify requirements on the ordering of different operations of each individual access. They do not constrain the ordering of operations of different accesses.
- *Program order*. These rules specify how an execution's visibility order must respect the execution's program order and the memory ordering semantics. For this reason, they apply only to pairs of accesses (or operations) of individual processors. Typically, a program-order rule specifies that, if accesses A_1 and A_2 are such that A_1 precedes A_2 in program order, then some operations of A_1 must precede some operations of A_2 in the execution's visibility order. Such rules may depend on the types of accesses A_1 and A_2. For example, some specifications include such a rule if A_1 and A_2 are both writes, but not if A_1 is a write and A_2 is a read. Program-order rules do not constrain the ordering of operations from different processors.
- *Memory-data dependence*. Memory-data dependence rules are similar to program-order rules in that they constrain only the order of operations by an individual processor. They differ from program-order rules in that they apply only to accesses whose ranges intersect (or overlap).
- *Data-flow dependence*. Data-flow dependence rules are similar to memory-data dependence rules. They apply to pairs of accesses by a processor between which there is data flow.

1. In this document, the term "processor" refers to a logical processor in the case of a multithreaded CPU.

The next two rule types apply to instructions by different processors:

- *Coherence*. Coherence rules are used in the specification of all models that enforce a global ordering of writes to each memory location.
- *Read value*. Read-value rules indicate, for a given global visibility order, the values that any access that reads from memory should return. For a given operation, the values are usually specified on a byte-by-byte basis.

The Itanium architecture ascribes to each physical address a *memory attribute*, which controls (among other things) the cacheability of the data located at that address. This document includes support for the following memory attributes of the Itanium architecture: writeback (WB) memory; write-coalescing (WC) memory; uncacheable (UC) memory; and uncacheable-exported (UCE) memory. The Itanium processor family memory ordering model includes a few rules that are specific to *memory attributes*. Unless otherwise noted, this document will treat UC and UCE as the same attribute and uses only the notation UC.

Although memory attributes are associated with physical addresses, they are determined by entries in the translation lookaside buffer (TLB), which translates virtual address to physical addresses. If more than one virtual address translated to the same physical address, it is possible that the physical address would have different memory attributes depending on the virtual address used to access it. It is also possible that the TLBs on different processors might ascribe different memory attributes to the same physical address. These situations are called *memory attribute aliasing*. This document assumes that there is no memory attribute aliasing: for each physical address there is one memory attribute such that any processor that has a translation to that address does so only with that memory attribute.

3.0 The Basic Itanium® Processor Family Memory Ordering Model

A definition of the Itanium processor family memory ordering model is given in the *Intel® Itanium® Architecture Software Developer's Manual*. However, that document lacks a formal definition of memory ordering.

This chapter presents the basic Itanium processor family memory ordering model within the framework established in Section 2.0. The basic model includes support only for loads, stores, and memory fences and assumes that all memory addresses are aligned. Section 4.0 extends the basic model to support semaphore instructions.

3.1 Terminology and Notation

As Section 2.0 outlines, every system execution contains a set of sequences of instructions, one per processor, that are decomposed into accesses (one access per instruction) and then into operations. The basic Itanium processor family memory ordering model adopts the following notation and terminology to describe aspects of instructions and operations:

- For any instruction, access, or operation x, Proc(x) is the processor that performs x.
- For any instruction, access, or operation x, Rng(x) is the *range* of the instruction, access, or operation; this is the set of byte addresses on which x operates. Ordering rules using Rng(x) assume that all bytes addressed by Rng(x) have the same memory attribute. If the bytes addressed by Rng(x) span multiple memory attributes, the order in which instructions, accesses, or operations occur is undefined.

Note: The rest of this document uses the term “byte” and “byte address” interchangeably.

- If instruction, access, 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). If b∈ Rng(X), RdVal(X;b) is the value read by X for byte b and we say that X is a read from b.

  We use “R” to stand for an instruction, access, or operation with read semantics.

- If instruction, access, or operation X writes to memory, we say that X has *write semantics* or is a *write*. WrVal(X) is the value written by X to Rng(X). If b∈ Rng(X), WrVal(X;b) is the value written by X to byte b and we say that X is a write to b.

  We use “W” to stand for an instruction, access, or operation with write semantics.

- 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). For all the examples in Appendix A and Appendix B, InitVal(b)=0 for all bytes b.

To describe the ordering relationships in the global visibility order or per-processor (program-order) sequences, the basic Itanium processor family memory ordering model adopts the following notation:

- For any two instructions or accesses I_1 and I_2, if p=Proc(I_1)=Proc(I_2) and I_1 precedes I_2 in program order, we write $I_1 \gg I_2$.[1]
- For any two instructions or accesses I_1 and I_2, if $I_1 \gg I_2$ and I_1 precedes I_2 in data-flow order (i.e., there is a data-flow dependence of I_2 on I_1), we write $I_1 \rightarrow I_2$.
- For any two operations O_1 and O_2, if O_1 precedes O_2 in the visibility order, we write $O_1 \rightarrow O_2$.

Note: Program order and data-flow order apply only to instructions and accesses, while visibility orders apply only to operations. As an example, if program order applies to a load and a store instruction, then the visibility order applies to the corresponding read (load) and write (for store) operations.

The Itanium architecture supports a relaxed memory ordering model that provides unordered memory instructions, explicitly ordered memory instructions, and fence instructions that software can use to implement stronger ordering. The *ordering semantics* and program order of an instruction determine the constraints the system must respect when placing operations from the same processor in the visibility order.

The Itanium architecture defines four ordering semantics for instructions and accesses:

- *Acquire*. If access I_1 follows in program order an access I_2 with acquire semantics, then the operations of I_1 in general follow those of I_2 in the execution’s visibility order.

  This document uses “ACQ” to represent an instruction with acquire semantics.

- *Release*. If access I_1 precedes in program order an access I_2 with release semantics, then the operations of I_1 in general precede those of I_2 in the execution’s visibility order.

  This document uses “REL” to represent an instruction with release semantics.

Note: All instructions with release semantics also have write semantics.

- *Fence*. If access I_1 precedes (respectively, follows) in program order an access I_2 with fence semantics, then the operations of I_1 in general precede (respectively, follow) those of I_2 in the execution’s visibility order.

  This document uses “FEN” to represent an instruction with fence semantics.

1. Because of the close correspondence between instructions and accesses, this document uses the notation “I” to refer to both. In most cases, it refers to accesses. The notation “A” is reserved for reference to operations of instructions and accesses with acquire semantics.

- *Unordered (or weak).* Instructions with unordered semantics do not place constraints on the visibility order. However, operations from instructions with acquire, release, or fence semantics do constrain the operations from an instruction with unordered semantics.

The constraints that the ordering semantics impose on accesses' operations are in addition to those imposed by other rules such as memory-data dependence, data-flow dependence, etc.

3.2 Instructions and Operations

The basic Itanium processor family memory ordering model uses five instruction types. These instructions compose the basic set of application-level memory instructions in the Itanium architecture. For each type, we indicate the associated semantics (which apply to the instruction's access) and the component operations. The instructions in the basic Itanium processor family memory ordering model include the following:

- *Load-acquires (LA).* These instructions have read semantics and *acquire ordering semantics*.

  A load-acquire access LA is decomposed into a single operation R(LA) that represents the visibility of the load-acquire LA. In general, all accesses with read semantics are decomposed in similar fashion.

- *Store-releases (SR).* These instructions have write semantics and *release ordering semantics*. For WB and WC memory, store-releases are bypassing writes in the sense that a processor may appear to observe its own writes earlier than those performed by other processors (see "local bypassing" on page 6). For UC memory, store-releases are not bypassing. Store-releases to WB memory also have the property that they become visible in the same order at all remote processors in a coherence domain.[1]

  A store-release access SR is decomposed into several operations: local visibility operation LV(SR) that represents the visibility of SR to the local processor, and, for each processor p, remote visibility operation RV_p(SR) that represents the visibility of SR to processor p. In general, all instructions with write semantics are decomposed in similar fashion.

  Note that, for an instruction W with write semantics, there is a "remote" RV_p(W) even for p=Proc(W). The interval between LV(W) and RV_p(W) (with p=Proc(W)) represents the period of time during which a read by p can receive a local bypass for bytes written by W.

- *Unordered loads (UL).* These instructions have read semantics and *unordered semantics*.

  Like load-acquires, an unordered load access UL is decomposed into one read operation, R(UL).

- *Unordered stores (US).* These instructions have write semantics and *unordered semantics*. For WB and WC memory, unordered stores are bypassing writes; for UC memory, they are not bypassing.

  Like store-releases, an unordered store access US is decomposed into LV(US) and RV_p(US) operations (one RV_p(US) for each processor p).

- *Memory fences (MF).* These instructions have neither read nor write semantics and they have *fence ordering semantics*.

  A memory fence, MF, is decomposed into a single fence operation F(MF).

Section 3.1 and this section introduce a variety of notation for instructions. For example, "R", "ACQ", or "LA" might be used to denote a load-acquire. We use "R" when we want to emphasize that it is a read; we use "ACQ" to emphasize its ordering semantics; we use "LA" to explicitly denote that it is a load-acquire.

1. A coherence domain is a collection of processors and memory for which the hardware ensures that all members of the domain observe changes in memory values. Identifications of the collections comprising coherence domains are platform specific.

3.3 Visibility Order Rules

This section presents the set of rules for the basic Itanium processor family memory ordering model that an execution's visibility order must satisfy. If there is no visibility order for an execution that satisfies all of these rules, the execution is not permitted by the Itanium architecture.

3.3.1 Write Operation Order

There is one write operation rule that specifies how the RV and LV operations that compose an access with write semantics must be ordered:

(WO) No write can become visible remotely before it becomes visible locally.
For every write access W, $LV(W) \rightarrow RV_p(W)$ for p=proc(W), and $RV_p(W) \rightarrow RV_q(W)$ for p=proc(W) and q≠proc(W).

3.3.2 Program Order

The program-order rules specify constraints on how each processor's operations must become visible based on the program order of their accesses and the memory ordering semantics of their instructions. If two accesses *from a processor* are program ordered, it may be necessary for their operations to become visible in that order. These rules place no requirements on the order of operations from different processors. In addition, they do not depend on the range of memory being accessed (rules dependent on address range are given in Section 3.3.3).

Program-order rules fall into three categories, corresponding to the different ordering semantics that constrain the ordering of operations (acquire, release, and fence):

(ACQ) No operation can become visible before a preceding acquire.
If ACQ»I, A is an operation of instruction ACQ, and O is an operation of instruction I, then A→O.

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

- If I»REL, instruction I does not have write semantics, and operation O is an operation of I, then O→LV(REL).
- If I»REL and instruction I has write semantics, then LV(I)→LV(REL) and $RV_p(I) \rightarrow RV_p(REL)$ for each processor p.

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

(REL) Operations become visible in-order with respect to memory fences.

- If FEN»I and O is an operation of instruction I, then F(FEN)→O.
- If I»FEN and O is an operation of instruction I, then O→F(FEN).

Notice that either case implies that any two memory fences become visible in program order: if FEN_1»FEN_2, then $F(FEN_1) \rightarrow F(FEN_2)$.

There are no rules that explicitly mention operations of accesses with unordered memory ordering semantics (i.e., unordered loads and stores). These operations have no intrinsic ordering requirements. The ordering of their visibility is constrained only by how they are ordered with respect to acquires, releases, and fences, as well as the rules given in the other subsections.

3.3.3 Memory-Data Dependence

Memory-data dependence rules specify constraints on how the operations of a processor's read and write accesses *to common locations* (address locations with overlapping ranges) become visible (to that processor) based on the accesses' program order. Like the program-order rules, these rules apply only to pairs of operations from the same processor. These rules may specify orderings even

for unordered loads and stores (which the program-order rules of Section 3.3.2 do not explicitly mention). In general, operations of two accesses from a processor to a common location must become visible to that processor in program order unless both operations are reads.[1] Memory-data dependence rules do not affect the ordering of the remotely visible operations of write accesses.

There are three memory-data dependence rules that express the read-after-write, write-after-read, and write-after-write requirements that the Itanium architecture imposes on instructions with read semantics (R) and write semantics (W):

(MD:RAW) No read may become visible locally before an earlier write to a common location.
- If $Rng(W) \cap Rng(R) \neq \emptyset$ and $W \gg R$, then $LV(W) \rightarrow R(R)$.

(MD:WAR) No write may become visible locally before an earlier read of a common location.
- If $Rng(R) \cap Rng(W) \neq \emptyset$ and $R \gg W$, then $R(R) \rightarrow LV(W)$.

(MD:WAW) Writes by a processor to a common location become visible to that processor in program order.
- If $Rng(W_1) \cap Rng(W_2) \neq \emptyset$ and $W_1 \gg W_2$, then $LV(W_1) \rightarrow LV(W_2)$ and $RV_p(W_1) \rightarrow RV_p(W_2)$ for processor $p=Proc(W_1)=Proc(W_2)$.

Notice that there is no read-after-read requirement in the Itanium architecture. Thus, if UL_1 and UL_2 are two unordered loads, the following is possible: $UL_1 \gg UL_2$, $Rng(UL_1) \cap Rng(UL_2) \neq \emptyset$, (even $Rng(UL_1)=Rng(UL_2)$), and $R(UL_2) \rightarrow R(UL_1)$.

Notice also that these rules constrain only the local visibility of writes. Thus, (MD:WAW) does *not* imply that W_1 and W_2 become visible remotely in program order. However, if W_1 and W_2 are both to WB memory or both to UC memory, then the coherence rule (see Section 3.3.5) will ensure that they become visible in program order to all processors.

3.3.4 Data-Flow Dependence

An execution's *data-flow order* is a subset of its program order. That is, if accesses I_1 and I_2 are data-flow ordered ($I_1 \rightarrow I_2$), then they are also program ordered ($I_1 \gg I_2$). Two program-ordered instructions are also data-flow ordered if the associated instruction semantics are such that I_2 must follow I_1. For example, I_1 may be a load into a register and I_2 may be a store that uses that register as address or data. Alternatively, I_1 might be a load and I_2 an instruction that follows a branch that is conditional on the value returned by that load. A formal definition of data-flow order is external to the framework adopted in this document and can be found in the pseudo-code of the *Intel® Itanium® Architecture Software Developer's Manual*.

There are four data-flow dependence rules that ensure that a processor's operations become internally visible in data-flow order.

(DF:RAR) If $R_1 \rightarrow R_2$, then $R(R_1) \rightarrow R(R_2)$.

(DF:RAW) If $W \rightarrow R$, then $LV(W) \rightarrow R(R)$.

(DF:WAR) If $R \rightarrow W$, then $R(R) \rightarrow LV(W)$.

(DF:WAW) If $W_1 \rightarrow W_2$, then $LV(W_1) \rightarrow LV(W_2)$.

As in Section 3.3.3, these rules constrain only the local visibility of writes. (DF:WAW) does not imply that W_1 and W_2 become visible remotely in program order. This will be ensured by the memory-data dependence rule (MD:WAW) and the coherence rule (see Section 3.3.5) if W_1 and W_2 are to common locations and both to WB memory or both to UC memory.

1. Because of (ACQ) in Section 3.3.2, if the first read has acquire semantics, then the read operations must appear in program order.

3.3.5 Coherence

The coherence rule constrains the visibility of writes to a common location. It ensures that all processors agree on the order in which such writes become remotely visible. The coherence rule controls only operations to addresses with the same memory attribute, where that attribute is either WB or UC.

The rules states that, if two writes to a location in either WB or UC memory become visible to a processor in some order, then they become visible to all processors in that order.

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

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

Notice that the second part of (COH) applies even if $proc(W_1) \neq proc(W_2)$. This is the first rule presented that can constrain the ordering of operations of accesses from different processors.

If unordered stores US_1 and US_2 do not write to a common location (i.e., $Rng(US_1) \cap Rng(US_2)=\emptyset$), then they may become visible to processors in different orders (e.g., $RV_p(US_1) \rightarrow RV_p(US_2)$ and $RV_q(US_2) \rightarrow RV_q(US_1)$), if they are not subject to other constraints.

Suppose that unordered stores US_1 and US_2 are both writes to UC memory or both writes to WB memory and that $p=Proc(US_1)=Proc(US_2)$. If $Rng(US_1) \cap Rng(US_2) \neq \emptyset$ and $US_1 \gg US_2$, then it must be that $RV_q(US_1) \rightarrow RV_q(US_2)$ for all processors q. This is because (MD:WAW) implies $LV(W_1) \rightarrow LV(W_2)$; (COH) then implies the desired result.

3.3.6 Read Value

This section uses the following definition. A read access R is *local for byte b* if $b \in Rng(R)$ and there is a write W to b with $p=Proc(W)=Proc(R)$ such that $LV(W) \rightarrow R(R) \rightarrow RV_p(W)$. If R is local for b, this means that R will return a value for b that is being "bypassed" from a local write.

Informally, if a read access is local for a byte, then it reads the value written by the latest preceding write by the same processor (RV1). If it is not local, it reads the value written by the latest preceding write that is remotely visible to the reading processor (RV2); if there is no such write, the read returns the initial value (RV3).

Formally, the value returned by each read access R with $Proc(R)=p$ is determined using the following for each byte $b \in Rng(R)$:

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

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

(RV3) Suppose R is not local for b and there is no write W to b such that $RV_p(W) \rightarrow R(R)$. Then $RdVal(R;b)=InitVal(b)$.

3.3.7 Rules Specific to Memory Attribute

This section considers properties that apply only to operations on certain memory attributes.

3.3.7.1 Total Ordering of WB Releases

Store-releases to writeback (WB) memory have a property that does not apply to the other memory attributes: they become remotely visible to all processors in the same order. This is enforced by requiring that WB releases have a property commonly referred to as *remote write atomicity*:

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

- If SR writes to WB memory and $RV_p(SR) \rightarrow O \rightarrow RV_q(SR)$ for processors p and q, then $O=RV_r(SR)$ for some processor r.

3.3.7.2 Sequentiality of UC Operations

Sequentiality is a property that applies only to operations on UC memory. Each location in UC memory belongs to a *peripheral domain*.[1] Sequentiality ensures that a processor's operations on a peripheral domain become visible in program order. These rules are very similar to the memory data-dependence rules of Section 3.3.3 but they also apply to pairs of reads:

(UC1) If R_1 and R_2 are read accesses from UC memory in the same peripheral domain and $R_1 \text{»} R_2$, then $R(R_1) \rightarrow R(R_2)$.

(UC2) If R and W are accesses to UC memory in the same peripheral domain and R»W, then $R(R) \rightarrow LV(W)$.

(UC3) If W and R are accesses to UC memory in the same peripheral domain and W»R, then $LV(W) \rightarrow R(R)$.

(UC4) If W_1 and W_2 are write accesses to UC memory in the same peripheral domain and $W_1 \text{»} W_2$, then $LV(W_1) \rightarrow LV(W_2)$.

3.3.7.3 No UC Bypassing

Uncacheable (UC) memory is *non-cacheable*. With regard to memory ordering, this means that there can be no local bypassing from UC writes:

(NC) If W is a write to UC memory, $p=Proc(W)$, and $LV(W) \rightarrow R(R) \rightarrow RV_p(W)$, then either $p \neq Proc(R)$ or $Rng(W) \cap Rng(R) = \varnothing$.

4.0 Semaphores

Documentation of the Itanium architecture uses the term *semaphore instructions* (or, simply, *semaphores*) to refer to those with read-modify-write functionality. That is, semaphores atomically read a location in memory, perform some function on the value read, and write the modified data back to memory. The Itanium architecture supports three operations: exchange (`xchg`), compare-and-exchange (`cmpxchg`), and fetch-and-add (`fetchadd`). In the Itanium architecture, semaphores operate only on WB memory, with the exception of `fetchadd`, which can also operate on UCE memory.

1. For memory addresses that control memory-mapped I/O, a peripheral domain is a platform-specific subset of the platform's I/O subsystem that all observes memory accesses in a common order. Two UC addresses that map to system memory (instead of memory-mapped I/O) are considered to be in the same peripheral domain if they are in the same coherence domain.

Semaphores differ from each other in their ordering semantics and the function that they use to modify the value being read. The ordering semantics is the main subject of this subsection. The functions that semaphores use to modify values are not important to the model but are included here for completeness: `xchg` writes into the memory location the value of a specified register; `cmpxchg` does the same but only if the old value of the location equals the contents of the CCV application register; and `fetchadd` increments the old value of the memory location that it accesses.

4.1 Extensions to the Notation and Terminology

Supporting semaphore instructions in the memory ordering model requires additions to the notation and terminology introduced for the basic Itanium processor family memory ordering model:

- Each semaphore access SEM has a modification function Mod_{SEM} that computes the new value of the memory location from the original value of the location.
- Because semaphores have both read and write semantics, they have both a read value, RdVal(SEM), and a write value, WrVal(SEM).

4.2 Extensions to the Instructions and Operations

Supporting semaphores requires extending the basic set of instructions and accesses with two additional types:

- *Acquire semaphores*. These accesses have read semantics, write semantics, and acquire semantics. In the Itanium architecture, acquire semaphores may apply exchange (`xchg`), compare-and-exchange (`cmpxchg`), and fetch-and-add (`fetchadd`) functions to memory.
- *Release semaphores*. These accesses have read semantics, write semantics, and release semantics. In the Itanium architecture, release semaphores may apply compare-and-exchange (`cmpxchg`) and fetch-and-add (`fetchadd`) functions to memory.

This document uses SEM to refer to a semaphore access. Each semaphore SEM is decomposed into the following operations: R(SEM) that represents the visibility of the semaphore's read; LV(SEM) represents the visibility of the semaphore's write to the local processor; and, for each processor p, RV_p(SEM) represents the visibility of the semaphore's write to processor p. Note that this set of operations follows as one would expect from the operations for accesses that perform reads or writes.

4.3 Extensions to the Visibility Order Rules

This section details the additions to the visibility order rule set as well as the interactions between semaphores and the existing rules in the basic Itanium processor family memory ordering model.

4.3.1 Semaphores

Despite having multiple operations, each semaphore appears to execute atomically; this is called *semaphore atomicity* and requires an extension to the rules of the basic Itanium processor family memory ordering model:

(SM1) Nothing can intervene between a semaphore's operations:

- If s_1 and s_2 are operations of SEM and $s_1 \rightarrow o \rightarrow s_2$, then o is also a operation of SEM.

Semaphores must obey the following semaphore read-write rule:

(SM2) A semaphore access's read operation precedes its locally visible write operation:

- For semaphore access SEM, R(SEM)→LV(SEM).

In addition, all semaphores must obey the following read-modify-write rule:

(SM3) A semaphore writes the value computed by its modification function on its read value.

- For any semaphore SEM, WrVal(SEM)=Mod_{SEM}(RdVal(SEM)).

Rule (SM1) implies that semaphores have a single total order; they become remotely visible to all processors in the same order. Rule (SM3) does not directly relate to the ordering of memory operations; it uses neither » nor →. Its treatment of the read and write values of semaphores is included here only for completeness.

Note: RdVal(SEM) is determined by applying read-value rules (RV1) – (RV3) to R(SEM).

4.3.2 Interactions with Rules of the Basic Itanium® Processor Family Memory Ordering Model

As described earlier, the Itanium architecture allows a semaphore to have either acquire ordering semantics or release ordering semantics, but not both. Semaphores with acquire semantics obey acquire program-order rule (ACQ) given in Section 3.3.2 above; similarly, those with release semantics obey release program-order rule (REL) of that section.

As semaphore accesses contain both read and write operations, they must obey the rules for reads and writes that were presented in Section 3.3. These items are not new rules that semaphores must obey. Rather, they simply observe that semaphores must obey the rules in Section 3.3 for reads and writes.

- *Write operation order rule* (WO). Because all semaphores have write semantics, they are subject to this rule.
- *Acquire program-order rule* (ACQ). As noted above, this applies to semaphores with acquire ordering semantics.
- *Release program-order rule* (REL). As noted above, this applies to semaphores with release ordering semantics.
- *Memory-fence program order rule* (REL). Semaphores must obey these rules in the same way that other instructions and operations do. However, semaphores themselves are *not* memory fences.
- *Memory-data dependence rules* (MD:RAW), (MD:WAR), *and* (MD:WAW). Because all semaphores have read and write semantics, they are subject to all these rules.
- *Data-flow dependence rules* (DF:RAR) – (DF:WAW). Because all semaphores have read and write semantics, they are subject to all these rules.
- *Coherence rule* (COH). Because all semaphores have write semantics and because all semaphores are to either WB or UCE memory, they are subject to this rule.
- *Read-value rules* (RV1) – (RV3). Semaphores follow these rules both in their satisfaction of their read semantics (i.e., these rules determine the value that a semaphore reads) and of their write semantics (i.e., the rules determine how the value written by a semaphore can be read).
- *UC-operation rules* (UC1) – (UC4). Because they have read and write semantics, `fetchadd` operations on UCE memory must obey these rules.

4.4 Semaphores Do Not Allow Bypassing

The following results show certain properties of semaphores with respect to bypassing. Lemma 1 shows that no semaphore can read a value through bypassing. Lemma 2 shows that no read can be bypassed a value from a semaphore. These results apply to semaphores with both acquire and release semantics.

Lemma 1: *No semaphore can be local for any byte that it reads.*

Proof: Recall that b must be in either WB memory or UC memory. Semaphore SEM by p=Proc(SEM) is local for b (as a read) only if b∈ Rng(SEM) and there is a write W to b by p such that Proc(W)=p and LV(W)→R(SEM)→RV_p(W). Suppose this is the case.

Semaphore read-write rule (SM2) implies R(SEM)→LV(SEM); thus, LV(W)→LV(SEM). Memory-data dependence rule (MD:WAW) then implies RV_p(W)→RV_p(SEM). This means R(SEM)→RV_p(W)→RV_p(SEM), which contradicts semaphore-atomicity rule (SM1). QED

Since a semaphore cannot be local for any byte that it reads, it cannot receive values bypassed from other local writes.

Lemma 2: *No read that is local for a byte can read the value written to that byte by a semaphore*.

Proof: Let R be a load-acquire or unordered load that is local for b and let p=Proc(R) (Lemma 1 showed that no semaphore can be local for b). Assume for a contradiction that R reads the value written to b by semaphore SEM to b with Proc(SEM)=p. This implies that b must be in either WB memory or UC memory. Read-value rule (RV1) implies that LV(SEM)→R(R) and that there is no write W′ to b with Proc(W′)=p and that LV(SEM)→LV(W′)→R(R). Let W be the write to b that makes R local; that is, Proc(W)=p and LV(W)→R(R)→RV_p(W). Consider the following two cases:

- W=SEM. In this case, we have R(R)→RV_p(SEM).
- W≠SEM. Since there is no write W′ to b with Proc(W′)=p and LV(SEM)→LV(W′)→R(R), it must be that LV(W)→LV(SEM). Memory-data dependence rule (MD:WAW) implies RV_p(W)→RV_p(SEM). The transitivity of → then gives R(R)→RV_p(SEM).

In either case, R(R)→RV_p(SEM). Since LV(SEM)→R(R), this contradicts semaphore-atomicity rule (SM1). QED

Appendix A Examples

This appendix presents ten of the sample executions that were given in the *Intel® Itanium® Architecture Software Developer's Manual* and shows why each is or is not allowed by the Itanium processor family memory ordering model as defined in this document.

Some of the examples illustrate "weaknesses" in the Itanium processor family memory ordering model, showing that certain behaviors are allowed, a behavior being a certain execution coupled with the values returned to its read instructions. Behaviors are shown to be allowed by exhibiting a visibility order that is consistent with the rules of Section 3.3; these examples apply to all memory attributes. After the first such example, arguments regarding a visibility order's consistency with these rules will be abbreviated; rules from Section 3.3 will be cited only for emphasis.

Other examples illustrate "strengths" in the Itanium processor family memory ordering model, showing that certain behaviors are not allowed. This is done by assuming that the behavior is allowed and then using the read values and the rules from Section 3.3 to draw conclusions about the execution's visibility order. These conclusions will be contradictory, implying that the original assumption (that the execution is allowed by the Itanium processor family memory ordering model) was incorrect. These examples will apply to all memory attributes unless noted otherwise.

The examples assume that all instructions are properly aligned. Unless stated otherwise, it is assumed that all separately named memory locations (x, y, etc.) are disjoint in memory.

A.1 Itanium® Architecture Provides a Relaxed Ordering Model

Table 4 shows the weakness of unordered loads and stores and how they can be reordered.

Table 4. Itanium® Architecture Provides a Relaxed Ordering Model

p			q		
US_1:	st	[x] = 1	UL_1:	ld	r1 = [y]
US_2:	st	[y] = 1	UL_2:	ld	r2 = [x]

This execution has the following program order relations: US_1»US_2 and UL_1»UL_2. In addition, WrVal(US_1)=WrVal(US_2)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model allows the return values of r1=1 and r2=0. That is, it allows RdVal(UL_1)=1 and RdVal(UL_2)=0.

The indicated read values are supported by the following visibility order:

- $LV(US_1) \rightarrow LV(US_2) \rightarrow RV_p(US_1) \rightarrow RV_p(US_2) \rightarrow RV_q(US_2) \rightarrow R(UL_1) \rightarrow R(UL_2) \rightarrow RV_q(US_1)$.

This order respects write operation rule (WO). The program-order rules of Section 3.3.2 do not apply because there are only unordered loads and unordered stores in this execution. The memory data-dependence rules of Section 3.3.3 do not apply because neither processor has two operations on a single data location. The data-flow dependence rules of Section 3.3.4 do not apply because there is no data-flow ordering in this execution. Regardless of the memory attributes of the physical addresses involved, coherence rule (COH) does not apply because there is only one write to each location. Neither load is local (in the sense of Section 3.3.6); RdVal(UL_1)=1 by read-value rule (RV2); and RdVal(UL_2)=0 by read-value rule (RV3). Finally, the rules specific to memory attribute from Section 3.3.7 do not apply because all operations are on WB memory and none are store-releases.

There are several other visibility orders that would justify the indicated return values.

A.2 Enforcing Basic Ordering

Table 5 shows that the reorderings of the previous example can be eliminated with load-acquires and store-releases.

Table 5. Acquire and Release Semantics Order Itanium® Architecture Memory Operations

p			q		
US:	st	[x] = 1	**LA**:	ld.acq	r1 = [y]
SR:	st.rel	[y] = 1	**UL**:	ld	r2 = [x]

This execution has the following program order relations: US»SR and LA»UL. In addition, WrVal(US)=WrVal(SR)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model does not allow the return values of r1=1 and r2=0. That is, it does not allow RdVal(LA)=1 and RdVal(UL)=0.

Assume that the execution with the indicated read values is allowed by the Itanium processor family memory ordering model. The rules given in Section 3.3 imply the following orderings:

- Consider RV_q(US) and RV_q(SR). Since US»SR, program-order rule (REL) implies $RV_q(US) \rightarrow RV_q(SR)$.
- Consider R(LA). Because q does not write to y, LA is not local for y and read-value rule (RV1) does not apply. Because RdVal(LA)=1≠InitVal(x), read-value rule (RV3) does not apply. This means that LA must follow the remote visibility (to q) of some write to y. The only choice is SR, so $RV_q(SR) \rightarrow R(LA)$.
- Consider R(UL). Because LA»UL, program-order rule (ACQ) implies $R(LA) \rightarrow R(UL)$.
- Consider RdVal(UL). Because q does not write to x, UL is not local for x and read-value rule (RV1) cannot apply. From the items above, we have $RV_q(US) \rightarrow RV_q(SR) \rightarrow R(LA) \rightarrow R(UL)$. Because → is transitive, $RV_q(US) \rightarrow R(UL)$. Finally, since Rng(US)=Rng(UL), read-value rule (RV2) must apply to UL. Since US is the only write to x, (RV2) implies RdVal(UL)=WrVal(US)=1. This contradicts the assumption RdVal(UL)=0, and we conclude that the execution is not allowed by the Itanium processor family memory ordering model.

A.3 Allow Loads to Pass Stores to Different Locations

Table 6 illustrates that even load-acquires and store-releases can be reordered in some cases.

Table 6. Loads May Pass Stores to Different Locations

p			q		
SR_1:	st.rel	[x] = 1	**SR_2**:	st.rel	[y] = 1
LA_1:	ld.acq	r1 = [y]	**LA_2**:	ld.acq	r2 = [x]

This execution has the following program order relations: SR_1»LA_1 and SR_2»LA_2. In addition, WrVal(SR_1)=WrVal(SR_2)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model allows the return values of r1=r2=0. That is, it allows RdVal(LA_1)=RdVal(LA_2)=0.

The indicated read values are supported by the following visibility order:

- $R(LA_1) \rightarrow R(LA_2) \rightarrow LV(SR_1) \rightarrow LV(SR_2) \rightarrow RV_p(SR_1) \rightarrow RV_q(SR_1) \rightarrow RV_q(SR_2) \rightarrow RV_p(SR_2)$.

There are no rules that prevent a load-acquire operation from passing any operation of a preceding store-release. The memory-data dependence rules of Section 3.3.3 do not apply because, for each processor, its two operations are to disjoint memory ranges.

A.4 Preventing Loads From Passing Stores to Different Locations

Table 7 shows how memory fences can prevent loads and stores from being reordered.

Table 7. Loads May Not Pass Stores in the Presence of a Memory Fence

p			q		
US_1:	st	[x] = 1	US_2:	st	[y] = 1
MF_1:	mf		MF_2:	mf	
UL_1:	ld	r1 = [y]	UL_2:	ld	r2 = [x]

This execution has the following program order relations: $US_1 \text{»} MF_1 \text{»} UL_1$ and $US_2 \text{»} MF_2 \text{»} UL_2$. In addition, WrVal($US_1$)=WrVal($US_2$)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium architecture does not allow the return values of r1=r2=0. That is, it does not allow RdVal(LA)=RdVal(UL)=0.

Assume that the execution with the indicated read values is allowed by the Itanium architecture. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_q(US_1)$ and $F(MF_1)$. By program-order rule (REL), $RV_q(US_1) \rightarrow F(MF_1)$.
- Consider $R(UL_1)$. By program-order rule (REL), $F(MF_1) \rightarrow R(UL_1)$.
- Consider $RV_p(US_2)$. Because p does not write to y, UL_1 is not local for y and read-value rule (RV1) does not apply to UL_1. If $RV_p(US_2) \rightarrow R(UL_1)$, read-value rule (RV2) will imply RdVal(UL_1)=WrVal(US_2), which is not the case. Thus, $R(UL_1) \rightarrow RV_p(US_2)$.
- Consider $F(MF_2)$. By program-order rule (REL), $RV_p(US_2) \rightarrow F(MF_2)$.
- Consider $R(UL_2)$. By program-order rule (REL), $F(MF_2) \rightarrow R(UL_2)$.
- Consider RdVal(UL_2). Because q does not write to x, UL_2 is not local for x and read-value rule (RV1) cannot apply. From the items above, we have the following requirement on $\rightarrow$:
  - $RV_q(US_1) \rightarrow F(MF_1) \rightarrow R(UL_1) \rightarrow RV_p(US_2) \rightarrow F(MF_2) \rightarrow R(UL_2)$.

Because $\rightarrow$ is transitive, $RV_q(US_1) \rightarrow R(UL_2)$. Finally, since Rng($US_1$)=Rng($UL_2$), read-value rule (RV2) must apply to UL_2. Since US_1 is the only write to x, (RV2) implies RdVal(UL_2)=WrVal(US_1)=1. This contradicts the assumption RdVal(UL_2)=0, and we conclude that the execution is not allowed in the Itanium architecture.

A.5 Data Dependency Does Not Establish MP Ordering

The tables in this section illustrates some subtle properties of the memory-data and data-flow dependence rules.

Table 8 emphasizes that these rules affect only *local* visibility.

Table 8. Data Dependencies Do Not Establish MP Ordering (1)

p			q		
US_1:	st	[x] = 1	LA:	ld.acq	r2 = [y]
UL_1:	ld	r1 = [x]	UL_2:	ld	r3 = [x]
US_2:	st	[y] = r1			

This execution has the following program order relations: $US_1 \text{»} UL_1 \text{»} US_2$ and $LA \text{»} UL_2$. There is one data-flow relation: $UL_1 \text{›} US_2$. Finally, $WrVal(US_1)=1$. (The value of $WrVal(US_2)$ is $RdVal(UL_1)$, which depends on the execution.)

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model allows the return values of r1=r2=1 and r3=0. That is, it allows $RdVal(UL_1)=RdVal(LA)=1$ and $RdVal(UL_2)=0$.

The indicated read values are supported by the following visibility order:

- $LV(US_1) \rightarrow R(UL_1) \rightarrow LV(US_2) \rightarrow RV_p(US_2) \rightarrow RV_q(US_2) \rightarrow R(LA) \rightarrow R(UL_2) \rightarrow RV_p(US_1) \rightarrow RV_q(US_1)$.

There are no program-order rules that apply to p's operations. Memory-data dependence rule (MD:RAW) is respected because $LV(US_1) \rightarrow R(UL_1)$. Data-flow dependence rule (DF:WAR) is respected because $R(UL_1) \rightarrow LV(US_2)$. However, there is no rule that infers $RV_q(US_1) \rightarrow RV_q(US_2)$ from $LV(US_1) \rightarrow LV(US_2)$ (memory-data dependence rule (MD:WAW) and cache-coherence rule (COH) do not apply here because $Rng(US_1) \cap Rng(US_2) = \varnothing$).

Information in Table 9 is similar to that in Table 4 and shows that, if certain data-flow dependencies exist, an unordered load can function locally as if it were a load-acquire.

Table 9. Data Dependencies Do Not Establish MP Ordering (2)

p			q		
US:	st	[x] = 1	UL_1:	ld	r1 = [y]
SR:	st.rel	[y] = x	UL_2:	ld	r2 = [r1]

This execution has the following program order relations: $US \text{»} SR$ and $UL_1 \text{»} UL_1$. There is one data-flow relation: $UL_1 \text{›} UL_2$. Finally, $WrVal(US)=1$ and $WrVal(SR)=x$ (i.e., the memory address).

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model does not allow the return values of r1=x and r2=0. That is, it does not allow $RdVal(UL_1)=x$ and $RdVal(UL_2)=0$.

Assume that the execution with the indicated read values is allowed by the Itanium processor family memory ordering model. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_q(US)$ and $RV_q(SR)$. Since $US \text{»} SR$, program-order rule (REL) implies $RV_q(US) \rightarrow RV_q(SR)$.

- Consider R(UL_1). Because q does not write to y, UL_1 is not local for y and read-value rule (RV1) does not apply. Because RdVal(UL_1)=x≠InitVal(y), read-value rule (RV3) does not apply. This means that UL_1 must follow the remote visibility (to q) of some write to y. The only choice is SR, so $RV_q(SR) \rightarrow R(UL_1)$.
- Consider R(UL_2). Because $UL_1 \rightarrow UL_2$, data-flow dependence rule (DF:RAR) implies $R(UL_1) \rightarrow R(UL_2)$.
- Consider RdVal(UL_2). Because q does not write to x, UL_2 is not local for x and read-value rule (RV1) cannot apply. From the items above, we have $RV_q(US) \rightarrow RV_q(SR) \rightarrow R(UL_1) \rightarrow R(UL_2)$. Because → is transitive, $RV_q(US) \rightarrow R(UL_2)$. Finally, since Rng(US)=Rng(UL_2), read-value rule (RV2) must apply to UL_2. Since US is the only write to x, (RV2) implies RdVal(UL_2)=WrVal(US)=1. This contradicts the assumption RdVal(UL_2)=0, and we conclude that the execution is not allowed by the Itanium architecture.

A.6 Store Buffers May Satisfy Local Loads

Table 10 illustrates what can happen in an execution in which some loads are satisfied locally and others are not (see Section 3.3.6): a remotely satisfied load can appear to pass an earlier load-acquire that is satisfied locally.

Table 10. Store Buffers May Satisfy Loads if the Stored Data is Not Yet Globally Visible

p			q		
SR_1:	st.rel	[x] = 1	**SR_2**:	st.rel	[y] = 1
LA_1:	ld.acq	r1 = [x]	**LA_2**:	ld.acq	r3 = [y]
UL_1:	ld	r2 = [y]	**UL_2**:	ld	r4 = [x]

This execution has the following program order relations: $SR_1 \rightarrow LA_1 \rightarrow WR_1$ and $SR_2 \rightarrow LA_2 \rightarrow WR_2$. In addition, WrVal($SR_1$)=WrVal($SR_2$)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model allows the return values of r1=r3=1 and r2=r4=0. That is, it allows RdVal(LA_1)=RdVal(LA_2)=1 and RdVal(UL_1)=RdVal(UL_2)=0.

The indicated read values are supported by the following visibility order:

- $LV(SR_1) \rightarrow R(LA_1) \rightarrow LV(SR_2) \rightarrow R(LA_2) \rightarrow R(UL_1) \rightarrow R(UL_2) \rightarrow RV_p(SR_1) \rightarrow RV_q(SR_1) \rightarrow RV_q(SR_2) \rightarrow RV_p(SR_2)$.

Memory data-dependence rule (MD:RAW) ensures $LV(SR_1) \rightarrow R(LA_1)$ and $LV(SR_2) \rightarrow R(LA_2)$. However, it does not keep the load-acquires from passing the remotely visible operations of the store-releases. Notice that the two load-acquires are each local and are thus satisfied by read-value rule (RV1). The two unordered loads are each satisfied by read-value rule (RV3).

A.7 Preventing Store Buffers From Satisfying Local Loads

Table 11 shows how memory fences can be used to prevent a load from being satisfied locally.

Table 11. Preventing Store Buffers from Satisfying Local Loads

p			q		
US_1:	st	[x] = 1	US_2:	st	[y] = 1
MF_1:	mf		MF_2:	mf	
LA_1:	ld.acq	r1 = [x]	LA_2:	ld.acq	r3 = [y]
UL_1:	ld	r2 = [y]	UL_2:	ld	r4 = [x]

This execution has the following program order relations: $US_1 \text{»} MF_1 \text{»} LA_1 \text{»} WR_1$ and $US_2 \text{»} MF_2 \text{»} LA_2 \text{»} WR_2$. In addition, $WrVal(US_1)=WrVal(US_2)=1$. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model does not allow the return values of r1=r3=1 and r2=r4=0. That is, it does not allow $RdVal(LA_1)=RdVal(LA_2)=1$ and $RdVal(UL_1)=RdVal(UL_2)=0$.

Assume that the execution with the indicated read values is allowed by the Itanium architecture. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_q(US_1)$ and $F(MF_1)$. By program-order rule (REL), $RV_q(US_1) \rightarrow F(MF_1)$.
- Consider $R(LA_1)$. By program-order rule (REL), $F(MF_1) \rightarrow R(LA_1)$.
- Consider $R(UL_1)$. By program-order rule (ACQ), $R(LA_1) \rightarrow R(UL_1)$.
- Consider $RV_p(US_2)$. Because p does not write to y, UL_1 is not local for y and read-value rule (RV1) does not apply to UL_1. If $RV_p(US_2) \rightarrow R(UL_1)$, read-value rule (RV2) will imply $RdVal(UL_1)=WrVal(US_2)$, which is not the case. Thus, $R(UL_1) \rightarrow RV_p(US_2)$.
- Consider $F(MF_2)$. By program-order rule (REL), $RV_p(US_2) \rightarrow F(MF_2)$.
- Consider $R(LA_2)$. By program-order rule (REL), $F(MF_2) \rightarrow R(LA_2)$.
- Consider $R(UL_2)$. By program-order rule (ACQ), $R(LA_2) \rightarrow R(UL_2)$.
- Consider $RdVal(UL_2)$. Because q does not write to x, UL_2 is not local for x and read-value rule (RV1) cannot apply. From the items above, we have the following requirement on $\rightarrow$:
  - $RV_q(US_1) \rightarrow F(MF_1) \rightarrow R(LA_1) \rightarrow R(UL_1) \rightarrow RV_p(US_2) \rightarrow F(MF_2) \rightarrow R(LA_2) \rightarrow R(UL_2)$.

Because $\rightarrow$ is transitive, $RV_q(US_1) \rightarrow R(UL_2)$. Finally, since $Rng(US_1)=Rng(UL_2)$, read-value rule (RV2) must apply to UL_2. Since US_1 is the only write to x, (RV2) implies $RdVal(UL_2)=WrVal(US_1)=1$. This contradicts the assumption $RdVal(UL_2)=0$, and we conclude that the execution is not allowed in the Itanium architecture.

A.8 Ordered Cacheable Operations Seen in Same Order

Table 12 illustrates WB release atomicity: two WB store-releases must be remotely observed in the same order globally. This section applies only to WB memory.

Table 12. Enforcing the Same Visibility Order to All Observers in a Coherency Domain

p			q			r			s		
SR_1:	st.rel	[x] = 1	LA_1:	ld.acq	r1 = [x]	SR_2:	st.rel	[y] = 1	LA_2:	ld.acq	r3 = [y]
			UL_1:	ld	r2 = [y]				UL_2:	ld	r4 = [x]

This execution has the following program order relations: LA_1»UL_1 and LA_2»UL_2. In addition, WrVal(SR_1)=WrVal(SR_2)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model does not allow the return values of r1=r3=1 and r2=r4=0. That is, it does not allow RdVal(LA_1)=RdVal(LA_2)=1 and RdVal(UL_1)=RdVal(UL_2)=0.

Assume that the execution with the indicated read values is allowed by the Itanium processor family memory ordering model. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_s(SR_1)$ and R(LA_1). Because q does not write to x, LA_1 is not local for x and read-value rule (RV1) does not apply to LA_1. If R(LA_1)→$RV_q(SR_1)$, read-value rule (RV3) will imply RdVal(LA_1)=0, which is not the case. Thus, $RV_q(SR_1)$→R(LA_1). If R(LA_1)→$RV_s(SR_1)$, we have $RV_q(SR_1)$→R(LA_1)→$RV_s(SR_1)$, contradicting the remote atomicity of WB store-releases (WBR). Thus, $RV_s(SR_1)$→R(LA_1).
- Consider R(UL_1). By program-order rule (ACQ), R(LA_1)→R(UL_1).
- Consider $RV_q(SR_2)$. Because q does not write to y, UL_1 is not local for y and read-value rule (RV1) does not apply to UL_1. If $RV_q(SR_2)$→R(UL_1), read-value rule (RV2) will imply RdVal(LA_1)=WrVal(SR_2), which is not the case. Thus, R(UL_1)→$RV_q(SR_2)$
- Consider R(LA_2). Because s does not write to y, LA_2 is not local for y and read-value rule (RV1) does not apply to LA_2. If R(LA_2)→$RV_s(SR_2)$, read-value rule (RV3) will imply RdVal(LA_2)=0, which is not the case. Thus, $RV_s(SR_2)$→R(LA_2). If R(LA_2)→$RV_q(SR_2)$, we have $RV_s(SR_2)$→R(LA_2)→$RV_q(SR_2)$, contradicting the remote atomicity of WB store-releases (WBR). Thus, $RV_q(SR_2)$→R(LA_2).
- Consider R(UL_2). By program-order rule (ACQ), R(LA_2)→R(UL_2).
- Consider RdVal(UL_2). Because s does not write to x, UL_2 is not local for x and read-value rule (RV1) cannot apply. From the items above, we have the following requirement on →:
  - $RV_s(SR_1)$→R(LA_1)→R(UL_1)→$RV_q(SR_2)$→R(LA_2)→R(UL_2).

Because → is transitive, $RV_s(SR_1)$→R(UL_2). Finally, since Rng(SR_1)=Rng(UL_2), read-value rule (RV2) must apply to UL_2. Since SR_1 is the only write to x, (RV2) implies RdVal(UL_2)=WrVal(SR_1)=1. This contradicts the assumption RdVal(UL_2)=0, and we conclude that the execution is not allowed in the Itanium architecture.

A.9 Obeying Causality

Table 13 shows how WB release atomicity can be important even in executions with only one store-release. It shows an execution in which the Itanium processor family memory ordering model must respect causality. This section applies only to WB memory.

Table 13. Itanium® Processor Family Memory Ordering Obeys Causality

p			q			r		
SR:	st.rel	[x] = 1	**LA_1**:	ld.acq	r1 = [x]	**LA_2**:	ld.acq	r2 = [y]
			US:	st	[y] = 1	**UL**:	ld	r3 = [x]

This execution has the following program order relations: LA_1»US and LA_2»UL. In addition, WrVal(SR)=1 and WrVal(US)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model does not allow the return values of r1=1, r2=1, and r3=0. That is, it does not allow the following: RdVal(LA_1)=1; RdVal(LA_2)=1; and RdVal(UL)=0.

Assume that the execution with the indicated read values is allowed by the Itanium processor family memory ordering model. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_q(SR)$ and $R(LA_1)$. Because q does not write to x, LA_1 is not local for x and read-value rule (RV1) does not apply. Because $RdVal(LA_1)=1 \neq InitVal(x)$, read-value rule (RV3) does not apply. This means that LA_1 must follow the remote visibility (to q) of some write to x. The only choice is SR, so $RV_q(SR) \rightarrow R(LA_1)$.
- Consider $RV_r(SR)$. If $R(LA_1) \rightarrow RV_r(SR)$, we have $RV_q(SR) \rightarrow R(LA_1) \rightarrow RV_r(SR)$, which contradicts WB store-release atomicity (WBR). Thus, $RV_r(SR) \rightarrow R(LA_1)$.
- Consider $RV_r(US)$. Since $LA_1 \text{»} US$, program-order rule (ACQ) implies $R(LA_1) \rightarrow RV_r(US)$.
- Consider $R(LA_2)$. Because r does not write to y, LA_2 is not local for y and read-value rule (RV1) does not apply. Because $RdVal(LA_2)=1 \neq InitVal(y)$, read-value rule (RV3) does not apply. This means that LA_2 must follow the remote visibility (to r) of some write to y. The only choice is US, so $RV_r(US) \rightarrow R(LA_2)$.
- Consider R(UL). Because $LA_2 \text{»} UL$, program-order rule (ACQ) implies $R(LA_2) \rightarrow R(UL)$.
- Consider RdVal(UL). Because r does not write to x, UL is not local for x and read-value rule (RV1) cannot apply. From the items above, we have the following:
  - $RV_r(SR) \rightarrow R(LA_1) \rightarrow LV(US) \rightarrow RV_r(US) \rightarrow R(LA_2) \rightarrow R(UL)$.

Because $\rightarrow$ is transitive, $RV_r(SR) \rightarrow R(UL)$. Finally, since Rng(SR)=Rng(UL), read-value rule (RV2) must apply to UL. Since SR is the only write to x, (RV2) implies RdVal(UL)=WrVal(SR)=1. This contradicts the assumption RdVal(UL)=0, and we conclude that the execution is not allowed by the Itanium architecture.

Appendix B Additional Examples

This appendix presents a number of other sample executions that highlight certain features of the Itanium processor family memory ordering model.

As in Appendix A, all instructions in the examples are assumed to be properly aligned. The examples that indicate allowed behaviors apply to all memory attributes. The examples that indicate disallowed behaviors apply to all memory attributes unless noted otherwise. It is assumed that all separately named memory locations (x, y, etc.) are disjoint in memory unless otherwise noted.

B.1 Store Buffers May Satisfy Local Loads: A Variant

Table 14 is another example of local loads being satisfied from store buffers. It is given here because it is this example that causes Peterson's algorithm for mutual exclusion to fail on Itanium architecture-based platforms.

Table 14. Store Buffers May Satisfy Local Loads: A Variant

p			q		
SR_1:	st.rel	[x] = 1	SR_2:	st.rel	[y] = 1
SR_3:	st.rel	[z] = 1	SR_4:	st.rel	[z] = 2
LA_1:	ld.acq	r1 = [z]	LA_2:	ld.acq	r2 = [z]
UL_1:	ld	r3 = [y]	UL_2:	ld	r4 = [x]

This execution has the following program order relations: SR_1»SR_3»LA_1»UL_1 and SR_2»SR_4»LA_2»UL_2. In addition, WrVal(SR_1)=WrVal(SR_2)=WrVal(SR_3)=1 and WrVal(SR_4)=2. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=InitVal(z)=0, the Itanium processor family memory ordering model allows the return values of r1=1, r2=2, and r3=r4=0. That is, it allows RdVal(LA_1)=1, RdVal(LA_2)=2, and RdVal(UL_1)=RdVal(UL_2)=0.

The indicated read values are supported by the following visibility order:

- $LV(SR_1)\rightarrow LV(SR_2)\rightarrow LV(SR_3)\rightarrow LV(SR_4)\rightarrow R(LA_1)\rightarrow R(LA_2)\rightarrow R(UL_1)\rightarrow R(UL_2)\rightarrow RV_p(SR_1)\rightarrow RV_q(SR_1)\rightarrow RV_q(SR_2)\rightarrow RV_p(SR_2)\rightarrow RV_p(SR_3)\rightarrow RV_q(SR_3)\rightarrow RV_q(SR_4)\rightarrow RV_p(SR_4)$.

B.2 Ordered Cacheable Operations Seen in Same Order: A Variant

Table 15 gives an example similar to that given in Appendix A.8. In this case, however, there is no write operation that is observed by more than one processor. Like that section, this one applies only to WB memory.

Table 15. Ordered Cacheable Operations Seen in Same Order: A Variant

p			q			r			s		
US_1:	st	[w] = 1	LA_1:	ld.acq	r1 = [x]	US_2:	st	[y] = 1	LA_2:	ld.acq	r3 = [z]
SR_1:	st.rel	[x] = 1	UL_1:	ld	r2 = [y]	SR_2:	st.rel	[z] = 1	UL_2:	ld	r4 = [w]

This execution has the following program order relations: US_1»SR_1, LA_1»UL_1, US_2»SR_2, and LA_2»UL_2. In addition, WrVal(US_1)=WrVal(SR_1)=WrVal(US_2)=WrVal(SR_2)=1. There are no data-flow relations.

We wish to prove that, if InitVal(w)=InitVal(x)=InitVal(y)=InitVal(z)=0, the Itanium processor family memory ordering model does not allow the return values of r1=r3=1 and r2=r4=0. That is, it does not allow RdVal(LA_1)=RdVal(LA_2)=1 and RdVal(UL_1)=RdVal(UL_2)=0.

Assume that the execution with the indicated read values is allowed by the Itanium processor family memory ordering model. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_s(US_1)$ and $RV_s(SR_1)$. Because US_1»SR_1, program-order rule (REL) implies $RV_s(US_1)$→$RV_s(SR_1)$.
- Consider $RV_q(SR_1)$. If $RV_q(SR_1)$→$RV_s(US_1)$, we have $RV_q(SR_1)$→$RV_s(US_1)$→$RV_s(SR_1)$, which contradicts WB store-release atomicity (WBR). Thus, $RV_s(US_1)$→$RV_q(SR_1)$.
- Consider $R(LA_1)$. Because q does not write to x, LA_1 is not local for x and read-value rule (RV1) does not apply to LA_1. If $R(LA_1)$→$RV_q(SR_1)$, read-value rule (RV3) will imply RdVal(LA_1)=0, which is not the case. Thus, $RV_q(SR_1)$→$R(LA_1)$.
- Consider $R(UL_1)$. By program-order rule (ACQ), $R(LA_1)$→$R(UL_1)$.
- Consider $RV_q(US_2)$. Because q does not write to y, UL_1 is not local for x and read-value rule (RV1) does not apply to UL_1. If $RV_q(US_2)$→$R(UL_1)$, read-value rule (RV2) will imply RdVal(LA_1)=WrVal(US_2), which is not the case. Thus, $R(UL_1)$→$RV_q(US_2)$.
- Consider $RV_q(SR_2)$. Because US_2»SR_2, program-order rule (REL) implies $RV_q(US_2)$→$RV_q(SR_2)$.
- Consider $RV_s(SR_2)$. If $RV_s(SR_2)$→$RV_q(US_2)$, we have $RV_s(SR_2)$→$RV_q(US_2)$→$RV_q(SR_2)$, which contradicts WB store-release atomicity (WBR). Thus, $RV_q(US_2)$→$RV_s(SR_2)$.
- Consider $R(LA_2)$. Because s does not write to y, LA_2 is not local for y and read-value rule (RV1) does not apply to LA_2. If $R(LA_2)$→$RV_s(SR_2)$, read-value rule (RV3) will imply RdVal(LA_2)=0, which is not the case. Thus, $RV_s(SR_2)$→$R(LA_2)$.
- Consider $R(UL_2)$. By program-order rule (ACQ), $R(LA_2)$→$R(UL_2)$.
- Consider RdVal(UL_2). Because s does not write to x, UL_2 is not local for x and read-value rule (RV1) cannot apply. From the items above, we have the following requirement on →:
  - $RV_s(US_1)$→$RV_q(SR_1)$→$R(LA_1)$→$R(UL_1)$→$RV_q(US_2)$→$RV_q(SR_2)$→$R(LA_2)$→$R(UL_2)$.

Because → is transitive, $RV_s(US_1)$→$R(UL_2)$. Finally, since Rng(US_1)=Rng(UL_2), read-value rule (RV2) must apply to UL_2. Since US_1 is the only write to w, (RV2) implies RdVal(UL_2)=WrVal(SR_1)=1. This contradicts the assumption RdVal(UL_2)=0, and we conclude that the execution is not allowed in the Itanium architecture.

B.3 Using Memory Fences to Control Disjoint Accesses to a Location

This section gives an example in which there are writes to disjoint parts of a single memory location. It shows how memory fences can be used to control the order in which these writes are observed (see Table 16 and Table 17).

Table 16. Using Memory Fences to Control Disjoint Accesses to a Location

p			q		
US_1:	st1	[b0] = 0x11	US_2:	st1	[b1] = 0x22
MF_1:	mf		MF_2:	mf	
UL_1:	ld2	r1 = [w]	UL_2:	ld2	r2 = [w]

In this execution, b0 and b1 are two bytes that together compose word w. The execution has the following program order relations: US_1»MF_1»UL_1 and US_2»MF_2»UL_2. In addition, WrVal(US_1)=0x11 and WrVal(US_2)=0x22. There are no data-flow relations.

We wish to prove that, if InitVal(b0)=InitVal(b1)=0x00 (i.e., InitVal(w)=0x0000), the Itanium architecture does not allow the return values of r1=0x0011 and r2=0x2200. That is, it does not allow RdVal(UL_1)=0x0011 and RdVal(UL_2)=0x2200.

Assume that the execution with the indicated read values is allowed by the Itanium architecture. The rules given in Section 3.3 imply the following orderings:

- Consider $RV_q(US_1)$ and $F(MF_1)$. Because US_1»MF_1, program-order rule (REL) implies $RV_q(US_1)$→$F(MF_1)$.
- Consider $R(UL_1)$. Because MF_1»UL_1, program-order rule (REL) implies $F(MF_1)$→$R(UL_1)$.
- Consider $RV_p(US_2)$. Because p does not write to b1, UL_1 is not local for b1 and read-value rule (RV1) does not apply to UL_1 (with regard to the value it returns for b1). If $RV_p(US_2)$→$R(LA_1)$, read-value rule (RV2) would imply that UL_1 returns 0x22 for b1, which it does not (RdVal(UL_1)=0x0011). Thus, $R(UL_1)$→$RV_p(US_2)$.
- Consider $F(MF_2)$. By program-order rule (REL), $RV_p(US_2)$→$F(MF_2)$.
- Consider $R(UL_2)$. Because MF_2»UL_2, program-order rule (REL) implies $F(MF_2)$→$R(UL_2)$.
- Consider the value UL_2 returns for b0. Because q does not write to b0, UL_2 is not local for b0 and read-value rule (RV1) cannot apply. From the items above, we have the following requirement on →:
  - $RV_q(US_1)$→$F(MF_1)$→$R(UL_1)$→$RV_p(US_2)$→$F(MF_2)$→$R(UL_2)$.

Because → is transitive, $RV_q(US_1)$→$R(UL_2)$. Finally, since b0∈Rng(US_1)∩Rng(UL_2), read-value rule (RV2) must apply to UL_2. Since US_1 is the only write to b0, (RV2) implies that UL_2 returns for b0 the value written by US_1, which is 0x11. This contradicts the assumption RdVal(UL_2)=0x2200, and we conclude that the execution is not allowed in the Itanium architecture.

The execution in Table 17 shows that both memory fences are required to prevent the undesired results in the previous return values.

Table 17. Two Memory Fences are Necessary to Control Disjoint Accesses to a Location

p			q		
SR_1:	st.rel	[b0] = 0x11	SR_2:	st.rel	[b1] = 0x22
MF_1:	mf				
UL_1:	ld	r1 = [w]	UL_2:	ld	r2 = [w]

This execution has the following program order relations: SR_1»MF_1»UL_1 and SR_2»UL_2. As before, WrVal(SR_1)=0x11 and WrVal(SR_2)=0x22 and there are no data-flow relations.

In this case, it is possible, if InitVal(b0)=InitVal(b1)=0x00, to have RdVal(UL_1)=0x0011 and RdVal(UL_2)=0x2200. In particular, this is allowed by the following visibility order:

- $LV(SR_2) \rightarrow R(UL_2) \rightarrow LV(SR_1) \rightarrow RV_p(SR_1) \rightarrow RV_q(SR_1) \rightarrow F(MF_1) \rightarrow R(UL_1) \rightarrow RV_q(SR_2) \rightarrow RV_p(SR_2)$.

For UL_1 to return the required value, it must appear before $RV_p(SR_2)$ and, because of WB release atomicity (WBR), $RV_q(SR_2)$. Because of MF_1, it must appear after $RV_q(SR_1)$. For UL_2 to return the required value, it must appear after $LV(SR_2)$ and before $RV_q(SR_1)$. This means that it must also appear before $RV_p(SR_2)$ and $RV_q(SR_2)$. Because there is no memory fence at q, UL_2 may pass the remotely visible operations of access SR_2.

B.4 The Global Nature of Memory Fences

This section gives an example that illustrates the global nature of memory fences in that their presence can indirectly place restrictions on the visibility of WB store-releases by other processors (see Table 18 and Table 19). This section applies only to WB memory.

Table 18. The Global Nature of Memory Fences

p			q			r		
US:	st	[x] = 1	**SR**:	st.rel	[y] = 1	**LA**:	ld.acq	r2 = [y]
MF:	mf					**UL_2**:	ld	r3 = [x]
UL_1:	ld	r1 = [y]						

This execution has the following program order relations: US»MF»UL_1 and LA»UL_2. In addition, WrVal(US)=WrVal(SR)=1. There are no data-flow relations.

We wish to prove that, if InitVal(x)=InitVal(y)=0, the Itanium architecture does not allow the return values of r1=r3=0 and r2=1. That is, it does not allow RdVal(UL_1)=RdVal(UL_2)=0 and RdVal(LA)=1.

Assume that the execution with the indicated read values is allowed by the Itanium processor family memory ordering model. The rules given in Section 3.3 imply the following orderings:

- Consider RV_r(US) and F(MF). Because US»MF, program-order rule (REL) implies $RV_r(US) \rightarrow F(MF)$.
- Consider R(UL_1). Because MF»UL_1, program-order rule (REL) implies $F(MF) \rightarrow R(UL_1)$.
- Consider RV_p(SR). Because p does not write to y, UL_1 is not local for y and read-value rule (RV1) does not apply to UL_1. If $RV_p(SR) \rightarrow R(UL_1)$, read-value rule (RV2) would imply that RdVal(UL_1)=WrVal(SR)=1, which is not true. Thus, $R(UL_1) \rightarrow RV_p(SR)$.
- Consider RV_r(SR). If $RV_r(SR) \rightarrow R(UL_1)$, $RV_r(SR) \rightarrow UL_1 \rightarrow RV_p(SR)$, violating WB release atomicity (WBR). Thus, $R(UL_1) \rightarrow RV_r(SR)$.
- Consider R(LA). Because r does not write to y, LA is not local for x and read-value rule (RV1) does not apply to LA. If $R(LA) \rightarrow RV_r(SR)$, read-value rule (RV3) and the fact that there is no other write to x would imply that RdVal(LA)=InitVal(x)=0, which is not true. Thus, $RV_r(SR) \rightarrow R(LA)$.
- Consider R(UL_2). Because LA»UL_2, program-order rule (ACQ) implies $R(LA) \rightarrow R(UL_2)$.
- Consider RdVal(UL_2). Because r does not write to y, UL_2 is not local for y and read-value rule (RV1) cannot apply. From the items above, we have the following requirement on →:
  - $RV_r(US) \rightarrow F(MF) \rightarrow R(UL_1) \rightarrow RV_r(SR) \rightarrow R(LA) \rightarrow R(UL_2)$.

Because → is transitive, $RV_r(US_1) \rightarrow R(UL_2)$. Because Rng(US)=Rng($UL_2$), read-value rule (RV2) applies to UL_2. Since US is the only write to y, (RV2) implies that RdVal(UL_2)=WrVal(US)=1. This contradicts the assumption RdVal(UL_2)=0, and we conclude that the execution is not allowed in the Itanium architecture.

The execution in Table 19 is a modification of the one given in Table 17. It changes only which store is unordered and which is a store-release. With the changed labeling, the return values given above are now allowed.

Table 19. Limitations on the Global Nature of Memory Fences

p			q			r		
SR:	st.rel	[x] = 1	**US**:	st	[y] = 1	**LA**:	ld.acq	r2 = [y]
MF:	mf					**UL_2**:	ld	r3 = [x]
UL_1:	ld	r1 = [y]						

This execution has the following program order relations: SR»MF»UL_1 and LA»UL_2. In addition, WrVal(SR)=WrVal(US)=1. There are no data-flow relations.

In this case, if InitVal(x)=InitVal(y)=0, the Itanium processor family memory ordering model allows the return values of r1=r3=0 and r2=1. That is, it allows RdVal(UL_1)=RdVal(UL_2)=0 and RdVal(LA)=1. In particular, this is allowed by the following visibility order:

- $LV(US) \rightarrow RV_q(US) \rightarrow RV_r(US) \rightarrow R(LA) \rightarrow R(UL_2) \rightarrow LV(SR) \rightarrow RV_p(SR) \rightarrow RV_q(SR) \rightarrow RV_r(SR) \rightarrow F(MF) \rightarrow R(UL_1) \rightarrow RV_p(US)$.

Because q's store is unordered, it can become visible at p long after it becomes visible at r; the memory fence at p has little effect in this case.

B.5 Supposedly "Flickering" Writes

This section gives an example in which there are writes to different parts of a multi-byte memory location (see Table 20). It shows that, while the order of writes to each byte is unique by coherence, processors may not perceive a common order even of overlapping writes.

Table 20. Supposedly "Flickering" Writes

p			q		
UL_1:	ld2	r1 = [x]	**UL_3**:	ld2	r4 = [x]
SR_1:	st2.rel	[x] = 0xFFFF	**SR_2**:	st1.rel	[x] = 0x77
LA_1:	ld2.acq	r2 = [x]	**LA_2**:	ld2.acq	r5 = [x]
UL_2:	ld2	r3 = [x]	**UL_4**:	ld2	r6 = [x]

In this execution, the address x is accessed by both 1-byte and 2-byte instructions. In what follows, we use x to refer to the byte addressed by address x and y to refer to the other byte accessed by the two-byte instructions. The execution has the following program order relations: UL_1»SR_1»LA_1»UL_2 and UL_3»SR_2»LA_2»UL_4. In addition, WrVal(SR_1)=0xFFFF and WrVal(SR_2)=0x77. There are no data-flow relations.

We wish to prove that, if InitVal(x)=0x0000, the Itanium processor family memory ordering model allows the return values of r1=r4=0x0000, r2=0xFFFF, r5=0x0077, and r3=r6=0xFF77. That is, it allows RdVal(UL_1)=RdVal(UL_3)=0x0000, RdVal(LA_1)=0xFFFF, RdVal(LA_2)=0x0077 and RdVal(UL_2)=RdVal(UL_4)=0xFF77.

These return values are of interest for the following reason. Clearly, the final (2-byte) value of x is 0xFF77. Processor q sees first 0x0000, 0x0077, and finally 0xFF77. To q, it seems that its store (of 0x77) takes place first (as evidenced by the return value 0x0077) but later it appears that its store took place second (as evidenced by 0xFF77). The store by q appears to “flicker”, taking place twice.

In reality, there is nothing truly anomalous about these return values. The fact that they are allowed follows from local bypassing at both processors. In particular, they are allowed by the following visibility order:

- $R(UL_1) \rightarrow R(UL_3) \rightarrow LV(SR_1) \rightarrow LV(SR_2) \rightarrow R(LA_1) \rightarrow R(LA_2) \rightarrow RV_p(SR_1) \rightarrow RV_q(SR_1) \rightarrow RV_q(SR_2) \rightarrow RV_p(SR_2) \rightarrow R(UL_2) \rightarrow R(UL_4)$.

We consider the value returned by each load operation:

- $RdVal(UL_1)$, $RdVal(UL_3)$. Clearly, neither (RV1) nor (RV2) apply to these unordered loads as they precede all writes. By (RV3), they each return the initial values of x and y, which are both 0, so $RdVal(UL_1)=RdVal(UL_3)=0x0000$, as desired.
- $RdVal(LA_1)$. Since $Rng(LA_1)=Rng(SR_1)=\{x,y\}$, $LV(SR_1) \rightarrow R(LA_1) \rightarrow RV_p(SR_1)$, and $Proc(LA_1)=Proc(SR_1)$, LA_1 is local for both x and y. Thus, by read-value rule (RV1), $RdVal(LA_1)=WrVal(SR_1)=0xFFFF$, as desired.
- $RdVal(LA_2)$. For LA_2, the values returned for x and y must be considered separately:
  - x. Since $Rng(LA_2) \cap Rng(SR_2)=\{x\}$, $LV(SR_2) \rightarrow R(LA_2) \rightarrow RV_p(SR_2)$, and $Proc(LA_2)=Proc(SR_2)$, LA_2 is local for x. Thus, by read-value rule (RV1), LA_2 returns $WrVal(SR_2)=0x77$ for x.
  - y. Since $y \notin Rng(SR_2)$, neither (RV1) nor (RV2) apply to y. By (RV3), LA_2 returns $InitVal(y)=0x00$ for y.

  Thus, $RdVal(LA_2)=0x0077$, as desired.
- $RdVal(UL_2)$, $RdVal(UL_4)$. Clearly, (RV1) applies to these unordered loads as they are not local for any byte. Moreover, (RV2) applies to each read for each byte read as follows:
  - x. Each read sees SR_2 to be the most recent write to x and thus returns 0x0077 for x.
  - y. Each read sees SR_1 to be the most recent write to y and thus returns 0xFF for y.

  Thus, $RdVal(UL_2)=RdVal(UL_4)=0xFF77$, as desired.

Appendix C Glossary

The following is a list of some of the technical terms used in this document.

Bypassing (or Local Bypassing)	A feature of some memory ordering models by which write (or store) operations by a processor appear to become visible to that processor before others.
Coherence Domain	A collection of processors and memory for which the hardware ensures that all members of the domain observe changes in memory values. Identifications of the collections comprising coherence domains are platform specific.
Linear (Total) Order	A binary relation on a set that is irreflexive, transitive, and that relates any two elements in the set.
Memory Ordering Model	A collection of multiprocessor executions corresponding to a particular platform architecture.
Memory Ordering Specification	The definition of a memory ordering model, typically given by a set of rules that must hold for any execution in the model.
Peripheral Domain	For memory addresses that control memory-mapped I/O, a peripheral domain is a platform-specific subset of the platform's I/O subsystem that all observes memory accesses in a common order. Two UC addresses that map to system memory (instead of memory-mapped I/O) are considered to be in the same peripheral domain if they are in the same coherence domain.
Program Order	The per-processor order (usually, but not always, linear) that reflects the order of memory accesses in the program being executed.
Visibility Order	A linear order of operations in a system execution that corresponds to the order in which the operations become visible to the processors.