SGI® Altix® Architecture Considerations for Linux® Device Drivers

007–4763–001

The information in Chapter 2, “Memory Operation Ordering on SGI Altix Systems,” was originally authored by Jesse Barnes.

Record of Revision

Version	Description
001	August 2005 Original publication

Contents

Figures

About This Guide

This manual contains information about device drivers specific to SGI Altix systems or Silicon Graphics Prism Visualization Systems running an SGI ProPack 4 for Linux Service Pack 2 release. For information about device drivers on the SGI ProPack 3 for Linux release, see the *Linux Device Driver Programmer's Guide - Porting to SGI Altix Systems*.

If you are writing or porting a device driver to an SGI Altix or Silicon Graphics Prism system, see *Linux Device Drivers*, third edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, February 2005 (ISBN: 0-596-00590-3).

It provides detailed information about supporting computer peripherals under the Linux operating system based on the 2.6 kernel. It provides information on how to write drivers for a wide variety of devices. It is available at the following location:

http://www.oreilly.com/catalog/linuxdrive3/

Related Resources

The resources listed in this section contain additional information that might be helpful.

Developer Program

Information and support are available through the SGI Developer Program. To join the program, contact the Developer Response Center at 800-770-3033 or e-mail devprogram@sgi.com.

Internet Resources

A great deal of useful material can be found on the Internet. Some starting points are in the following list.

http://docs.sgi.com	SGI technical manuals to read or download

Note: Make sure you search in the entire Technical Publications Library (TPL) to view Linux and Altix systems documentation.

http://www.pcisig.com	Home page of the PCI bus standardization organization

SGI Altix System Documentation

For additional information on SGI Altix system documentation, see the following:

- *SGI ProPack 4 for Linux Start Here*

  Provides a comprehensive list of SGI Altix system hardware and software documentation.

- http://docs.sgi.com

  SGI technical manuals to read or download

  Note: Make sure you search in the entire Technical Publications Library (TPL) to view Linux and Altix systems documentation.

Standards Documents

The following documents are the official standard descriptions of buses:

- *PCI Local Bus Specification, Version 2.1*, available from the PCI Special Interest Group, P.O. Box 14070, Portland, OR 97214 (fax: 503-234-6762).
- *ANSI/IEEE standard 1014-1987* (VME Bus), available from IEEE Customer Service, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331.

Intel Compiler Documentation

Documentation for the Intel compilers is located on your system in the `/docs` directory of the directory tree where your compilers are installed. If you have installed the Intel compilers, the following documentation is available:

- *Intel C++ Compiler User's Guide* (`c_ug_lnx.pdf`)
- *Intel Fortran Compiler User's Guide* (`for_ug_lnx.pdf`)
- *Intel Fortran Programmer's Reference* (`for_prg.pdf`)
- *Intel Fortran Libraries Reference* (`for_lib.pdf`)

Other Intel Documentation

The following documents describe the Itanium (previously called "IA-64") architecture and other topics of interest:

- *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, available online at the following location:

http://developer.intel.com/design/itanium2/manuals/251110.htm

- *Intel Itanium Architecture Software Developer's Manual*, available online at the following location:

http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm

- *Introduction to Itanium Architecture*, available online at the following location:

http://shale.intel.com/softwarecollege/CourseDetails.asp?courseID=13

(secure channel required)

Additional Reading

The following additional publications may be useful:

- *Linux Device Drivers*, third edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, February 2005 (ISBN: 0-596-00590-3) available online at the following location:

  http://www.oreilly.com/catalog/linuxdrive3/

- *PCI-X System Architecture* by Tom Shanley, First edition, 2001. Mindshare Inc. Addison-Wesley, ISBN 0-2-1-72682-3.
- *PCI System Architecture* by Tom Shanley and Don Anderson, Third edition. Mindshare Inc.

Obtaining Publications

You can obtain SGI documentation as follows:

- See the SGI Technical Publications Library at http://docs.sgi.com. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, enter `infosearch` at a command line or select **Help > InfoSearch** from the Toolchest.
- On IRIX systems, you can view release notes by entering either `grelnotes` or `relnotes` at a command line.
- On Linux systems, you can view release notes on your system by accessing the `README.txt` file for the product. This is usually located in the `/usr/share/doc/`*productname* directory, although file locations may vary.
- You can view man pages by typing `man` *title* at a command line.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  http://docs.sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.
- Send mail to the following address:

  Technical Publications
  SGI
  1500 Crittenden Lane, M/S 535
  Mountain View, California 94043–1351

SGI values your comments and will respond to them promptly.

Altix Architecture and Linux Device Drivers

This document provides a description of issues that affect Linux device drivers executing on SGI Altix series systems or Silicon Graphics Prism Visualization Systems. SGI Altix systems use a global-address-space cache-coherent multiprocessor that can scale up to 512 processors in a cache-coherent domain.

This manual contains important SGI Altix architectural information about device drivers on SGI Altix systems or Silicon Graphics Prism Visualization Systems running SGI ProPack 4 for Linux Service Pack 2 release. For information about device drivers on SGI ProPack 3 for Linux SPx releases, see the *Linux Device Driver Programmer's Guide - Porting to SGI Altix Systems*.

If you are writing or porting a device driver to an SGI Altix or Silicon Graphics Prism system, see *Linux Device Drivers*, third edition, by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman, February 2005 (ISBN: 0-596-00590-3).

It provides detailed information about supporting computer peripherals under the Linux operating system based on the 2.6 kernel. It provides information on how to write drivers for a wide variety of devices. It is available at the following location:

http://www.oreilly.com/catalog/linuxdrive3/

The *IA-64 Linux Kernel Design and Implementation*, provides details on the implementation of IA-64 Linux on the Intel Itanium family of processors, which is the architecture on which the SGI Altix is based. Authors and publishers of these books are listed in the Preface ("About This Guide").

Caution: Drivers developed by using the information contained in this guide are the responsibility of the user. SGI does not extend any warranty to devices not officially supported by SGI. For information on devices officially supported on SGI and the support terms associated with them, see your support agreement.

This chapter addresses the following topics:

Chapter 2, "Memory Operation Ordering on SGI Altix Systems" on page 11, provides a description of memory operation ordering on systems using Intel Itanium 2 processors.

Legacy Functionality

Certain “legacy” methods are available to device drivers on other Linux systems that SGI Altix systems do not support (for example, using legacy I/O port numbers 0 through 64K, reading and using peripheral component interconnect (PCI) configuration base address registers (BARs) or interrupt requests (IRQs) directly from the card’s configuration space, and so on). Drivers that use legacy methods are not portable and they will not execute correctly on SGI Altix systems.

The SGI Altix system does not impose upon a Linux device driver the use of additional or different sets of Linux DKIs to function correctly on this platform. However, the SGI Altix system is a large, complex system, and for drivers to successfully invoke the full parallelism of the hardware and hence achieve optimal performance, it might well be necessary to invoke services and paradigms that are not available in the standard Linux DKI. **For specific information, see your SGI support representative.**

SGI Altix systems, which run Linux, provide the same I/O capabilities as the SGI Origin series systems, which run IRIX, except for the Intel processor and the little endian platform. The following list describes the legacy functionalities that are **not** available on the SGI Altix platform.

Legacy Functionality	Description
I/O ports	SGI Altix I/O subsystems do not support legacy I/O ports from either the Linux kernel or user level applications. If you use legacy I/O port numbers 0 –

	65K in I/O port macros such as `inb()` and `outb()`, the system will generate an exception.
Expansion ROM	SGI Altix systems do not read and execute basic input/output systems (BIOS) in expansion read-only memory (ROM), even if the ROM is present. Drivers and cards that depend on initialization by these BIOS might not function correctly on this platform. All initialization must be done by the drivers when the Linux kernel calls them to initialize.
RAM VGA video memory	SGI Altix systems do not support legacy video random access memory (RAM).
IRQs in PCI configuration space	The device driver cannot use the IRQ byte in the PCI configuration space. Device drivers are required to retrieve the IRQ number initialized by the kernel for that device in the `pci_dev` structure.
Base Address Registers	SGI Altix I/O subsystem PCI bridges cannot generate a "dual address" cycle for programmable I/O addresses on the PCI-X bus. As such, only 32 bits of the BARs can be initialized. However, the platform also requires a PIO address to be 64 bits wide. As such, the values in the BARs are not the same as the addresses that the device driver uses on the CPU. The addresses on the CPU have been mapped. Reading the BAR and using it in any I/O macros will cause an exception. PCI-X I/O and memory addresses for the devices are provided in the `pci_dev` structure. These values are already mapped and using them will correctly target the relevent PCI-X device.
Peripheral buses	The peripheral buses that SGI Altix systems support are PCI, PCI-X, and AGP buses. SGI Altix systems do not support traditional legacy I/O space such as I/O ports. PCI-X I/O resource space and memory resource space

are presented to the device driver as uncached virtual addresses.

Special Architectural Considerations

The following sections describe special architectural characteristics of SGI Altix systems.

Programmable I/O Write Operations

Programmable I/O (PIO) write operations on SGI Altix I/O subsystems can be cached in various components of the system, from the CPU to the PCI-X bridges. PIO write requests from the same CPU are guaranteed to be issued in program order. However, they are not synchronous. PIO write operations on this platform are posted. To guarantee that PIO write operations have completed, device drivers are required to push all prior PIO write operations out to the device by issuing a PIO read operation to the same controller after the last write operation before releasing a semaphore. This will prevent another CPU from acquiring the semaphore and having its PIO transactions complete before the previous holder of the semaphore.

PIO access and system memory access use different paths and hardware components on SGI Altix I/O subsystems. A `get/release` operation on a memory-based lock can complete before a PIO write request.

You are strongly advised to program device drivers to flush all relevent PIO write operations with a PIO read operation to the same controller prior to releasing the relevent memory-based locks.

PIO write operation caching is a performance feature. Making each PIO write operation synchronous incurs unneccessary performance penalty. Other Linux based platforms also require the device driver to explicitly execute PIO write flushing for correct operation.

Direct Memory Access

SGI Altix I/O subsystems provide support for posted direct memory access (DMA). With posted DMA capability, the host bridge can respond to the requester that the request is complete prior to actually transferring the data to target memory. This is a performance feature. DMA data is not guaranteed to arrive in memory “in-order”.

Device Interrupts and Posted DMAs

SGI Altix I/O subsystems use the interrupt mechanism to flush all posted DMA data to target memory. This is the only mechanism currently available to ensure that all posted DMAs are flushed into the target memory.

PIO Reads and Posted DMAs

PCI-X bridge chipsets on SGI Altix systems do not automatically flush Posted DMA writes on any PIO reads. For information regarding software flushing of posted DMA write buffers, see "Posted PIO `write` Calls" on page 10.

Polling Memory for Completion and Posted DMA Data

On SGI Altix systems, direct memory access (DMA) data from controller cards to system memory is not guaranteed to arrive “in-order”. If a device driver is polling a memory location for completion status and the completion status is the last DMA operation by the controller card, it is not guaranteed that all the prior DMA data will arrive in memory before the DMA completion status word. If you are polling memory for completion status, you must use consistent mapping routines for this “Completion Status”. Consistent mapping routines provide a DMA handle to flush all DMA data.

Using the appropriate mapping routine will ensure that all prior DMA data has arrived in memory before the “Completion Status” DMA data.

The Linux operating system provides two separate DMA mapping interfaces:

- Consistent mapping
- Streaming mapping

Itanium 2 Processors and Altix System Addresses

All addresses on an Altix system are 64 bits long . Drivers have to ensure that any structures that are allocated to store any addresses must be 64 bits long.

Note: It is very important to note that if you want to translate a virtual memory address into a bus address (DMA for the card), using the following macros for translation **WILL NOT** work:

```
bus_to_virt()
virt_to_bus()
```

An **Example** such as the following, will **not** work:

```
/* This will NOT work ... */
dmabuf = kmalloc(size, GFP_KERNEL);
writel(virt_to_bus(dmabuf), card's_dma_addr_reg);
```

System Physical Memory Addresses

An SGI Altix system does not have system physical memory smaller than or equal to 32 bits. To the device driver, system physical memory addresses are always 64 bits long.

The following macros will provide proper translation from physical-to-virtual or virtual-to-physical:

```
phys_to_virt()
virt_to_phys()
```

Note: A system physical address is not the same as a bus address. Therefore, system physical addresses cannot be used by the card for DMA, as is (see "Bus Addresses - PCI/PCI-X Buses" on page 6).

Bus Addresses - PCI/PCI-X Buses

Bus addresses are addresses that allow the device to perform DMA operations from the card into system physical memory. An SGI Altix system supports either a 64–bit bus address or a 32–bit bus address. These bus addresses must be obtained from the various `pci_map_XXX()` routines. See the section on direct memory access ddresses

(DMA). Legacy macros like `virt_to_bus`() and `bus_to_virt`() do **not** provide the correct mappings or translation.

Note: On an Altix system, there is no way to translate a bus address to virtual address. Drivers are responsible to save the the corresponding virtual address to the mapped DMA address. For more information, see "Direct Memory Access Addresses (DMA)" on page 9.

Programmable IO Read/Write Addresses

The following legacy routines do almost no work on Intel Itanium 2 platforms:

- `ioremap`() — Adds the IA64 uncache OFFset
- `iounmap`() — Does nothing
- `ioremap_nocache`() — Calls the `ioremap`() function

Drivers must use the IO addresses provided in the `pci_dev` structure for the device.

An **Example** such as the following, will **not** work:

```
/* This will not work .. */
pci_read_config_dword(pci_dev, PCI_BASE_ADDRESS_0, &ioaddr);
cards_regs = ioremap(ioaddr, 0x1000);
writel(0x60002,(cards_regs + (PCI_INT_CFG/PltfMsk)));
```

Base address registers in the PCI Configuration Space of a card cannot be used, as is, by the device driver for PIO. Device Drivers have to use addresses initialized in the `pci_dev` structure allocated by the system for that device via this routine, as follows:

```
pci_resource_start(dev,bar)
```

Other resource routines of interest are, as follows:

```
pci_resource_end(dev,bar)
pci_resource_flags(dev,bar)
pci_resource_len(dev,bar)
```

On an SGI Altix system, these addresses are 64 bits long, regardless of whether they are PCI IO or memory resources. PCI IO resource addresses can then be used in the following macros:

```
inb/inw/inl/outb/outw/outl
insb/insw/insl/outsb/outsw/outsl
```

Note: Hardcoded legacy addresses for example, IO Port Number 0x360, used in IN/OUT macros will not work, for example, `inb(0x360)`, and so on.

PCI memory resource addresses can then be used in the following macros:

```
readb/readw/readl/readq/writeb/writew/writel/writeq
```

Device Driver Interrupt Registration - IRQs

Device drivers register their interrupt handling routines by calling the following code:

```
int request_irq(unsigned int irq,
         void (*handler)(int, void *, struct pt_regs *),
         unsigned long irqflags,
         const char * devname,
         void *dev_id)
```

Of particular interest here is `irq` integer, which traditionally is the interrupt line in the PCI configuration space. Device drivers should **not** be reading this value from the PCI configuration space to get the `irq` value for the `request_irq`(). Instead, device drivers should use the `irq` number as allocated in the `pci_dev` structure by the Linux operating system, as follows:

```
pci_dev->irq
```

An **Example** such as the following, will **not** work:

```
/* This will not work .. */
pci_read_config_byte(pci_dev, PCI_INTERRUPT_LINE, &irq);
request_irq(irq, ...);
```

Direct Memory Access Addresses (DMA)

DMA addresses (bus addresses) on Altix system are either 64 bits or 32 bits, nothing in-between. Requests for DMA addresses between 33 and 63 bits are given 32 bits DMA addresses.

Device drivers cannot use legacy macros, such as the following:

```
bus_to_virt()
virt_to_bus()
```

Before calling any of the DMA mapping routines, a device driver should query the system for the DMA address size that the platform supports, using the following:

```
pci_dma_supported()
```

By default, the `dma_mask` is set by the Linux operating system to be `0xffffffff`, which means 32 bits.

On an Altix system, there no calls to convert addresses from bus-to-virtual or virtual-to-bus. If the driver requires the corresponding virtual address of a bus address, it should save the virtual address.

Linux provides the following routines for mapping Virtual Address to DMA address:

```
pci_alloc_consistent()
pci_free_consistent()
pci_map_single()
pci_unmap_single()
pci_map_sg()
pci_unmap_sg()
pci_dma_sync_single()
pci_dma_sync_sg()
```

See `linux/Documentation/DMA-mapping.txt` for more details.

The `pci_alloc_consistent`() routine, by default, returns a 32 bit DMA address to the caller. On an Altix system, there is an exception. If your card is a PCIX card running in PCIX mode, only 64–bit DMA addresses are returned. For cards running in PCIX mode, please use the following: `pci_set_consistent_dma_mask`() to set the consistent mask bits to `0xffffffffffffffff`. Otherwise, your call to `pci_alloc_consistent`() will fail.

Posted PIO `write` Calls

For performance reasons, PIO write calls are posted. That is, on return from a PIO `write` call for example, `outb(X)`, an Altix system does not guarantee that the PIO has arrived and been received by the designated device. To ensure that a PIO write has actually been delivered and received by the designated device, device drivers are required to perform a PIO read to a safe register on the device, for example reading the vendor's identification, and so on:

```
outb(X);
outb(XX);
inb(safe register address);
```

Note: Currently, the `sn_mmiob`() macro is only available on SGI Altix platforms.

On the SGI Altix platform, a faster PIO write flush macro is available, as follows:

```
outb(X);
outb(XX);
sn_mmiob();
```

Note: IO writes are delivered as soon as possible. In a ccNUMA architecture like used in an Altix system, if the system is very busy, a PIO write can be buffered by the IO chipsets.

The same rules apply to PIOs using the `readb`() family of macros.

For more information on synchronization issues regarding PIOs and memory references, see the *Linux Device Drivers Guide*.

Memory Operation Ordering on SGI Altix Systems

Memory operation ordering is a complicated set of rules with issues that are not specific to SGI Altix systems but rather to any Linux platforms with Intel Itanium 2-based processors. Similarly, this topic is not related to PIO posted operations.

The compiler can reorder instructions and also optimize away instructions that appear to be superflous or are not used. One technique it might use is to preload some registers, whose contents might or might not be valid by the time they are needed and used.

One optimization feature of Intel Itanium 2 processors is that they can reorder instructions such that some instructions are scheduled and completed not exactly in the order that they appear in your program. For more information regarding memory ordering, memory fences, and so on, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual* on for additional information on MP coherence and synchronization.

This appendix describes the following memory operation aspects of SGI Altix systems:

- Memory ordering
- Release semantics
- Acquire semantics
- Memory fencing

Memory Ordering

Memory load and store operations on SGI Altix platforms will not necessarily complete (that is, be visible in memory to other CPUs) in program order. For example, consider the following code snippet (program order):

```
1:  ld r1=[r2]  // r1 = *r2
2:  st [r4]=r6   // *r4 = r6
3:  ld r8=[r9]   // r8 = *r9
4:  st [r22]=r3  // *r22 = r3
```

This code could actually execute in the following order:

1. Register `r1` is set to the value at memory address `r2`.
2. Register `r8` is set to the value at memory address `r9`.
3. The address in `r22` is set to the value in `r3`.
4. The address in `r4` is set to the value in `r6`.

Note: This is a separate issue from compiler reordering, as it occurs at runtime. This also assumes that the pointers in question point to non-overlapping addresses. The kind of reordering shown in the previous example can expose bugs of various types, some of them very similar to the PIO ordering and coherency issues explained in this document.

Release Semantics

Using release semantics on an Intel Itanium 2 processor, the programmer can ensure that all previous memory accesses are made visible prior to the `st.rel` process, though subsequent memory accesses may "float up" above `st.rel`. For example, consider the following code sample:

```
1:  st [r1]=r2        // cannot move below 2
2:  st.rel [r4]=r6    // will be visible only after 1 is visible
3:  ld r8=[r9]        // may be reordered
4:  st [r22]=r3       // may be reordered
```

The processor will guarantee that the memory reference on line 1 is visible before the the `st.rel` on line 2; that is, the following sequence could be the actual execution order:

1. The address in `r1` is set to the value in `r2`.
2. The address in `r22` is set to the value in `r3`.
3. The address in `r4` is set to the value in `r6` (will happen after one register `r8` is set to the value at memory address `r9`).

In other words, no prior memory references (in program order) are allowed to propagate below a store with release semantics, but memory references following an `st.rel` **might** "float up" above the `st.rel` instruction.

Release semantics is a one-directional fence that prevents “Downward” drift as shown in Figure 2-1 on page 13.

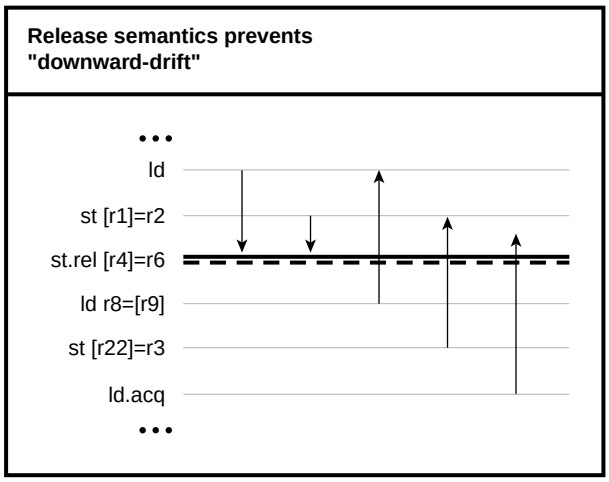

Figure 2-1 Release Semantics One-Directional Fence

For more information on release semantics, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer’s Manual*.

Acquire Semantics

Using so-called “acquire” semantics, the programmer can ensure that a load is made visible before all subsequent data accesses, though previous memory accesses can propagate below an `ld.acq` process. For example, consider the following code sample:

```
1:  ld r44=[r23]      // *can* move below 2
2:  ld.acq r1=[r2]    // will be visible before 3
3:  ld r8=[r9]        // cannot move above 2
4:  st [r4]=r6        // cannot move above 2
5:  st [r22]=r3       // cannot move above 2
```

The processor will ensure that the memory accesses prior to line 3 (in program order) are made visible before any subsequent accesses. So the following sequence could be executed by the processor:

1. Register `r1` is set to the value at memory address `r2` (will happen before 2).
2. Register `r8` is set to the value at memory address `r9`.
3. The address in `r4` is set to the value in `r6`.
4. Register `r44` is set to the value at memory address `r23`.
5. The address in `r22` is set to the value in `r3`.

Acquire semantics is a one-directional fence that prevents “Upward” drift as shown in Figure 2-2 on page 14.

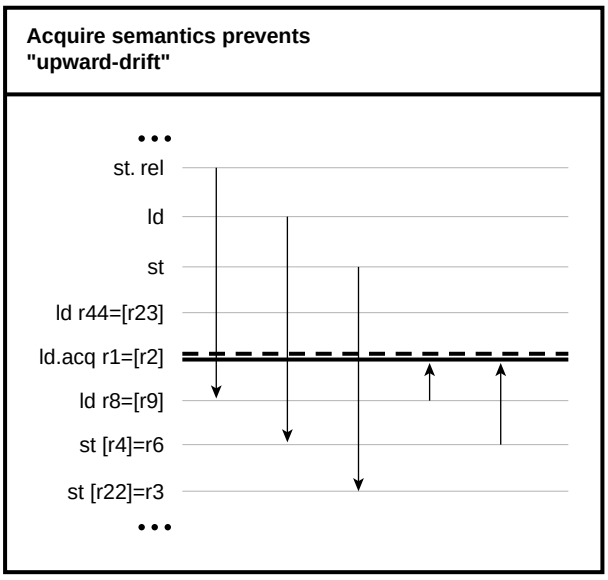

Figure 2-2 Acquire Semantics One-Directional Fence

.

For more information on acquire semantics, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual*.

Memory Fencing

A memory fence acts as a simple, two-way barrier for memory operations as shown in Figure 2-3 on page 15. For example, consider the following snippet:

```
1:  ld r1=[r2] <--\
2:  st [r4]=r6 <--- neither can move below 3
3:  mf
4:  ld.acq r8=[r9] <-- neither can move above 3
5:  st [r22]=r3 <----/
```

Lines 1 and 2 are guaranteed to be visible before any subsequent memory accesses (like those on lines 4 and 5), and memory accesses following the fence **will not** be visible to instructions before the memory fence (in program order).

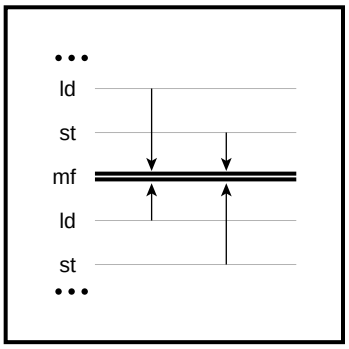

Figure 2-3 Two-dimensional Memory Fence (`mf`)

For more information on memory fencing semantics, see the *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization* and *Intel Itanium Architecture Software Developer's Manual*.

Index