PowerPC storage model and AIX programming

What AIX programmers need to know about how their software accesses shared storage

Michael Lyons
Bill Hay
Brad Frey

November 16, 2005
(First published August 28, 2002)

Are you an AIX programmer writing device drivers or using shared storage with multithreaded applications for POWER4 systems like the p690? It’s essential that you know and follow the architectural rules so that your programs produce expected results when accessing shared storage. The authors spell it out for you.

Introduction

The PowerPC architecture defines a storage model for the ordering of storage accesses that is called “weakly consistent.” This model provides an opportunity for improved performance, but it requires that your software programs explicitly order accesses to storage that is shared by other threads or I/O devices. To make sure those accesses are performed in program order, you must place the appropriate memory barrier instructions between storage accesses to shared storage. Programs that don't contain memory barrier instructions in appropriate places might execute correctly on older platforms, but they might fail on new platforms (such as POWER4 systems like the p690), because the new processors aggressively reorder memory accesses to improve performance. It’s essential that AIX programmers writing device drivers or using shared storage with multithreaded applications follow the architectural rules.

We'll give you the information you need to review your code for architectural compliance. We begin by summarizing salient features of the PowerPC architecture, and we include pseudo-code examples to illustrate real-world application of the concepts.

Concepts and terminology

First, we want to extract some definitions from the architecture books and introduce some relevant concepts. Don't worry -- we don't intend to describe every architectural nuance, but we want to provide enough foundation for discussion later in this article, where we'll show more directly how these concepts apply to AIX drivers.
The term *storage access* means an access to memory caused by a load, a store, a DMA write, or a DMA read.

There are three orderings to consider:

<table>
<thead>
<tr>
<th>Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Order defined by the sequential execution model</td>
<td>In this model, each instruction appears to complete before the next instruction starts. In general, from the view of a program that does not access shared memory, it appears that instructions are executed in the order specified by the program, that is, the program order.</td>
</tr>
<tr>
<td>Order of execution of instructions</td>
<td>The sequential execution model doesn't require that the processor execute instructions in program order. A program that doesn't access shared memory can't detect that a sequence of instructions is executed in an order different than that specified in the program, even though modern processors frequently execute instructions in a different order.</td>
</tr>
<tr>
<td>Order in which storage accesses are performed</td>
<td>The order in which memory accesses are performed may be different from both the program order and the order in which the instruction that caused the accesses are executed. Consider a program that contains a sequence of loads from locations A, B, and C in that order. The processor might execute these instructions in a different order (for example, B, A, C) and perform the accesses caused by the instructions in yet a different order (for example, C, B, A).</td>
</tr>
</tbody>
</table>

The PowerPC architecture defines four storage control attributes:

- Write through required
- Caching inhibited
- Memory coherence required
- Guarded

In an AIX environment, programs may access:

- Storage with attributes of Caching Inhibited and Guarded. For example, registers or memory on an I/O adapter.
- Storage that is neither Caching Inhibited nor Guarded but for which all accesses are coherent. For example, main storage that can be cached and may be read speculatively; unlike I/O storage, reading main storage causes no side effects.

The ordering rules governing a storage access are dependent on the control attributes of the storage location. When you're writing device drivers, you're typically concerned with two classes of storage:

- System memory -- this has the Memory Coherence Required attribute.
- Device memory -- this has the Caching Inhibited and Guarded attributes. This is the memory you get addressibility to via the AIX iomem_att or io_map kernel services. In Linux, this is the memory you get addressibility to via the ioremap service.

For simplicity, we'll use the terms *device memory* and *system memory* in the remainder of this article to describe these two storage classes.
If the operand of a load or store instruction is located in device memory, the access to the operand might be performed in a single operation or it might be performed as a sequence of operations, depending on the alignment of the operand, the size of the operand, and the types of operations supported along the path (i.e., on the buses) between the processor and the device memory. For example, if the operand is a single byte, the access is performed by a single operation.

However, if the operand is a doubleword and is not aligned on a doubleword boundary, the access could be performed by a sequence of three to eight bus operations. This could be a problem depending on the device.

On newer p-Series systems, the storage accesses caused by string instructions, instructions that specify an effective address that is not a multiple of the storage operand size, Load Multiple instructions, or Store Multiple instructions are all also subject to "stuttering." This occurs when the access is partially performed by the processor and then restarted after being interrupted (by an external interrupt or some event internal to the processor, for example). From the device's viewpoint, the execution of the instruction could cause the addressed storage to be accessed more than once (possibly causing a loss of information if the accesses cause side-effects). String and multiple operations to device memory are very slow (have poor performance) on newer processor implementations. It is recommended that drivers avoid using these instructions on these platforms, and that they align all I/O to device memory to natural boundaries.

**Ordering rules and mechanisms**

The architecture provides some rules that apply to ordering of storage accesses. Additionally, the Enforce In-order Execution of I/O (eieio) and Synchronize (sync) instructions provide explicit control.

A storage access to *device memory* is performed only if the access is caused by an instruction that is known to be required by the sequential execution model. This doesn't mean the access must occur in program order (for example, the order you'd expect from reading a program listing), just that the access can't be speculatively executed.

If a load instruction depends on the result returned by a preceding load instruction, the corresponding storage accesses are performed in program order.

Atomic accesses to the same storage location are ordered.

Two store instructions to *device memory* are performed in program order. This was an architecture change made in February 1999. Before then, the architecture allowed for stores to device memory to be reordered. As a result, many current drivers have eieio instructions (more on eieio in a minute) to control the order in which store accesses are made. Although these instructions were added for architectural correctness under the original architecture definition, in fact there were never any processor implementations that required this. This was confirmed for RS/6000 at the time the architecture change was made, and it was also recently confirmed for the Motorola 7400, 7410, 7450, and 82xx processors. For driver writers developing new code, we suggest you omit instructions whose sole purpose is to provide ordering between two stores to device memory. For
existing code, you can delete the instructions if performance work is being done, but make sure you perform regression testing on all supported processor implementations. In most cases, it's probably okay to leave the existing code as is.

Also, the ordering of accesses to device memory is deterministic only in the case where all the accesses are to the same device. To order stores to devices on two different PCI buses requires additional software protocols, such as performing a load from one device to ensure that a store has completed before permitting a store to a second device that is on a different PCI bus. The io_flush example later in this paper further illustrates this point.

All these rules don't prevent stores to device memory from being issued out of program order with respect to loads from device memory, or vice versa. Instead, the memory-barrier instructions, `eieio` and `sync`, must be used when out of order operations could yield incorrect results.

**Memory barrier instructions**

The `eieio` instruction and the `sync` instruction each create a memory barrier that orders storage access instructions. The memory barrier divides instructions into those that precede the barrier-creating instruction and those that follow it. The accesses caused by instructions that precede the barrier-creating instruction must be performed prior to those that are caused by instructions which occur after the barrier-creating instruction. While `sync` orders all storage access, `eieio` orders only particular subsets of storage accesses.

There are several points to keep in mind. First, the execution or completion of the `eieio` instruction does not imply that the storage accesses caused by instructions preceding the `eieio` have completed. The barrier created by the instruction will ensure that all accesses separated by the barrier are performed in the specified order, but accesses issued prior to the barrier might not be performed until long after the `eieio` instruction completes. A second critical aspect regarding `eieio` is that it further divides the storage access instructions into two classes: those to system memory and those to device memory. Accesses to one class of memory are ordered independent of accesses to the other class of memory. So load and store accesses to device memory that follow the `eieio` instruction are guaranteed to be performed after the load and store accesses to device memory resulting from instructions before the `eieio`. And store accesses to system memory on either side of the barrier are likewise ordered. However, `eieio` has no effect on the order in which two accesses are performed if one access is to device memory and the other is to system memory. Also, `eieio` has no effect in ordering loads to system memory, and it's not recommended for ordering stores to system memory. As a final note, `eieio` is not cumulative for device memory accesses. This is in contrast to the `sync` instruction, as described below.

The barrier created by the `sync` instruction is more comprehensive in that it orders all storage accesses regardless of class. There are two versions of the `sync` instruction.

- The first form is heavy-weight `sync`, or commonly just referred to as `sync`. This form is often needed by adapter device drivers, for ordering system memory accesses made by a driver critical section with accesses made to its I/O adapter. Executing `sync` ensures that
all instructions preceding the `sync` instruction have completed before the `sync` instruction completes, and that no subsequent instructions are initiated until after the `sync` instruction completes. This does **not** mean that the previous storage accesses have completed before the `sync` instruction completes. A store access has not been performed until a subsequent load is guaranteed to see the results of the store. Similarly, a load is performed only when the value to be returned can no longer be altered by a store.

The memory barrier created by the `sync` instruction provides "cumulative" ordering. This means that after a program executing on processor P0 observes accesses performed by a program executing on processor P1 and then executes a `sync` instruction, subsequent storage accesses performed by the program executing on processor P0 will be performed with respect to other processors after the accesses it observed to have been performed by the program executing on processor P1. For example, a program executing on processor P3 will not see a sequence of storage accesses that would conflict with the order of the sequence observed and performed by the program executing on processor P0.

A heavy-weight `sync` occurs within the AIX kernel's unlock services (for example, `unlock_enable`). Device drivers typically `unlock_enable` before leaving their interrupt handler, and the underlying heavy weight `sync` has guaranteed that storage accesses to device memory are made visible with respect to other processors correctly.

- The second form of the `sync` instruction is light-weight sync, or `lwsync`. This form is used to control ordering for storage accesses to **system memory** only. It does **not** create a memory barrier for accesses to device memory. This kind of `sync` occurs within the kernel's `unlock_mem` services (for example, `unlock_enable_mem`). Software needing coordination between shared system memory access by different CPUs, without concern for related ordering of accesses to device memory, can make use of `lwsync`. An example later in the document shows an additional potential use for `lwsync` in device drivers.

The following table summarizes use of the memory barrier instructions:

<table>
<thead>
<tr>
<th>Storage barriers and ordering of storage accesses</th>
<th>Caching Enabled (&quot;system memory&quot;)</th>
<th>Caching Inhibited &amp; Guarded (&quot;device memory&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction -&gt; Storage accesses¹</td>
<td>sync²</td>
<td>lwsync³</td>
</tr>
<tr>
<td>load/load</td>
<td>Yes</td>
<td>Recommended</td>
</tr>
<tr>
<td>load/store</td>
<td>Yes</td>
<td>Recommended</td>
</tr>
<tr>
<td>store/load</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>store/store</td>
<td>Yes</td>
<td>Recommended Note 6</td>
</tr>
<tr>
<td></td>
<td>sync²</td>
<td>lwsync³</td>
</tr>
</tbody>
</table>

**Notes:**

1. The terms "Yes", "Recommended", and "N/A" apply when both accesses are to storage with the
2. **sync (sync 0)** orders all accesses regardless of memory attributes (for example, it will order a caching-inhibited load with respect to a caching-enabled store).

3. **lwsync (sync 1)** has no effect on the ordering of caching-inhibited accesses. It should be used only to order accesses to caching-enabled storage.

4. **eieio** does not order accesses with differing storage attributes (for example, if an **eieio** is placed between a caching-enabled store and a caching-inhibited store, the accesses could still be performed in an order different than specified by the program).
Synchronization instructions

The synchronization instructions include Instruction Synchronize (\texttt{isync}), Load and Reserve (\texttt{lwarx/ldarx}), and the Store Conditional (\texttt{stwcx/stdcx}) instructions.

The Load and Reserve and Store Conditional instructions are used by AIX to implement the high-level synchronization functions that are needed by application programs and device drivers. Applications and device drivers should use the AIX-provided interfaces, rather than use the Load and Reserve and Store Conditional instructions directly.

An \texttt{isync} instruction causes the processor to complete execution of all previous instructions and then to discard instructions (which may have begun execution) following the \texttt{isync}. After the \texttt{isync} is executed, the following instructions then begin execution. \texttt{isync} is used in locking code to ensure that the loads following entry into the critical section are not performed (because of aggressive out-of-order or speculative execution in the processor) until the lock is granted.

Cache management instructions

The PowerPC architecture provides a number of cache management instructions. These include \texttt{dcbf} (data cache block flush) and \texttt{dcbst} (data cache block store). Using these instructions to flush data cache lines prior to the start of an adapter DMA can have a positive or negative impact on
performance, depending on the underlying implementation. I/O is coherent regardless of whether these instructions are present. And it is important to note that `dcbf` and `dcbst` do not ensure anything about instruction completion or ordering. They are subject to the visibility and ordering rules previously described. For storage access ordering purposes, they are treated as loads (from system memory). If a device driver is using `dcbf` or `dcbst` prior to a device memory access that causes DMA to the flushed location, an intervening `sync` instruction will be required. See the "Pull" Adapter model section for additional explanation.

When you are performance tuning your device driver, we suggest you examine any pre-existing usage of cache management instructions. It’s possible some usages originated due to misunderstandings that the instructions were required for coherency reasons, or that the instructions had properties similar to a `sync`. If the instructions are intended to improve performance, analysis should be done to decide if these instructions should be used in a common driver source base that runs on numerous implementations, or only on certain implementations. In general the preferred outcome is a single driver source base that has no special code for unique implementations.

As a final cache management note for driver writers, older versions of the PowerPC architecture included the `dcbi` (data cache block invalidate) instruction. Driver writers should never use this instruction. It may perform poorly, incorrect usage can result in a security exposure, and on newer platforms it will result in an Illegal Instruction interrupt that will not be handled by AIX. If you believe flushing the cache is desirable, use `dcbf` instead of `dcbi`.

**Non-atomic accesses**

The PowerPC architecture states that no ordering should be assumed among the storage accesses caused by a single instruction for which the access is not atomic. And there is no method for controlling that order. As previously mentioned, you should be very circumspect in using multiple, string, or unaligned storage accesses to device memory (and avoid completely on newer platforms, for performance reasons if nothing else). On new platforms, using these instructions to access device memory (even aligned multiple or string instructions) can cause an Alignment interrupt, causing the access to be emulated via software, and resulting in performance that is hundreds of times slower than an aligned load or store instruction.

**Base AIX support**

The AIX operating system provides a collection of basic functions, built upon the previously described instructions, to perform important operations needed for shared memory synchronization. These includes locks, semaphores, atomic update primitives, etc. These are available as system calls, as kernel services, or via libraries. Most applications that execute on AIX/pSeries SMP systems already use the various libraries available to safely manage the shared memory communication. Examples of such libraries include:

- AIX pthread library
- ESSL SMP library
- Compiler SMP runtime library
Even so, there are instances of application code that needs to ensure that proper synchronization occurs, as some of the following examples illustrate.

AIX adapter device drivers typically need to concern themselves with CPU accesses to both device memory and system memory, often in conjunction with DMA accesses made to system memory by the adapter they control. The examples that follow show architecturally compliant ways to address common problems.

**Examples**

**Global thread flag**

One thread may need to indicate to other threads that it has completed some part of its computation by setting a shared flag variable to a particular value. The setting of this flag must not occur until after all the computed data has been stored to system memory. Otherwise, some other thread (on another processor) could see the flag being set and access data locations which have not yet been updated. To prevent this, a `sync` or `lwsync` must be placed between the data stores and the flag store. `lwsync` is the preferred instruction in this case.

```
< compute and store data >
 ..
 lwsync
<store flag>
```

**Waiting on flag to signal data has been stored**

In this example, an `isync` is used to prevent the processor from using stale data from the block. The `isync` prevents speculative execution from accessing the data block before the flag has been set. And in conjunction with the preceding load, compare, and conditional branch instructions, the `isync` guarantees that the load on which the branch depends (the load of the flag) is performed prior to any loads that occur subsequent to the `isync` (loads from the shared block). `isync` is not a memory barrier instruction, but the load-compare-conditional branch-isync sequence can provide this ordering property. Several additional examples will further illustrate this sequence.

```
Loop:   load global flag
        Has flag been set?
        No:  goto Loop
        Yes: continue
        isync
        <use data from shared block>
```

Note that a `sync` instruction could have been used, but an `isync` is a much less expensive instruction, in terms of effect upon performance.

"Pull" adapter model

It is very common for device drivers to use the `d_map` kernel services to share a portion of system memory with an I/O adapter. Typically, the device driver fills out some command metadata in
system memory, and does a memory-mapped I/O (PIO) to the I/O adapter (IOA). The I/O adapter in turn DMA reads the system memory address for the command-specific information ("pulls" the cmd info from system memory), and executes the command. A sync is required to correctly program this sequence:

```
store cmd_info to system memory addr X;            /* shared storage */
__iospace_sync();         /* compiler built-in for sync instruction */
store new_cmd_trigger to device memory addr Y;    /* pio to device */
```

device does a DMA read to memory location X as a result of the PIO

Without any intervening memory-barrier instruction (for example, without the sync in the above example), this sequence has been observed to fail on a p690. Latencies can be such that the PIO to the adapter and the adapter's DMA can occur before the CPU's store to memory location X has made it into the coherency domain. The result is that the adapter reads stale data. Note that an eieio instruction would not be sufficient in this case, because eieio orders storage accesses to device memory and system memory separately. Also, using dcbf or dcbst to store the cmd_info to system memory does not eliminate the need for the sync - without the sync the adapter DMA can still get stale data.

"Push" adapter model

Other I/O adapters rely more on a "push" scheme, in which the CPU PIOs all of the command information to device memory before starting a command. Since earlier versions of the architecture (or at least the driver community's understanding of it) did not require two stores to device memory to be issued in program order, it is common in current AIX driver code to see sequences like:

```
store cmd_info1 to device memory addr X;
__iospace_eieio();        /* compiler built-in for eieio instruction */
store start_cmd to device memory addr Y;
```

The first store sets up a new command. The second store starts the command. If the stores were not delivered to the IOA in order, the IOA could potentially start a new command before it had been correctly programmed. Now that the architecture clearly requires stores like the above to be performed in program order, the eieio is unnecessary.

load/store reordering

There can be instances in a device driver's protocol with its IOA that a register must be written before valid data can be read from another register. This case requires the use of eieio:

```
store enable_read_val to device memory address X
__iospace_eieio();    /* the above store needs to be presented to
* the IOA before the following load is
* presented to the IOA. */
val = load from device memory address Y
```

io_flush
Sometimes a driver needs to be sure that a storage access to device memory has completed. AIX encountered an instance of this problem several years ago, and produced the `io_flush` macro as part of the solution. On a system where the interrupt controller was on a different PCI bus than the interrupting IOA, the following was observed with this typical execution sequence on the CPU:

1. `dd` interrupt handler loads from device interrupt register
2. `dd` interrupt handler stores to device register to clear interrupt
3. `dd` interrupt handler returns to system external interrupt subsystem, indicating interrupt was serviced
4. System external interrupt subsystem stores to interrupt controller, to clear interrupt level into processor

The store to the interrupt controller completed before the store to the device occurred. As a result, the device was still presenting its interrupt to the interrupt controller, and a second interrupt resulted. By the time the operating system started to service this second interrupt, the original store in step 2 above had completed, so the device driver reported that its device was not interrupting, and an unclaimed interrupt resulted. To solve this problem, a code sequence is needed to ensure that the store in step 2 actually completes before the store in step 4 is initiated. One solution to this race involves the `io_flush` macro, which is currently in the `inline.h` AIX header file. Its usage in an AIX driver interrupt handler might look like this:

```c
pio read to see if its your intr
return INTR_FAIL if not your intr
pio write to clear intr
eieio to enforce pio ordering of last write before subsequent pios
   val = pio read from same address space as the pio write (it must be to address space owned by the same PCI bridge). By spec, the bridge must complete the above write to clear the interrupt before completing this read
   io_flush(val); /* make sure the read completes */
return INTR_SUCC
system external interrupt subsystem will pio write to MPIC or PPC interrupt controller
```

The goal is to get the pio write to clear the interrupt out to the adapter before the system's pio write out to the interrupt controller (which in the example is under a different PCI host bridge, or PHB). `eieio` won't guarantee this, it can only guarantee the writes leave the coherency domain (reach the PHBs) in order. So a read is used to flush the first write (once at the PHB, loads can't pass stores), and `io_flush` ensures the read completes (via a conditional branch based on the value read) and guards (via an `isync`) against any chance of a really aggressive processor starting to execute the system external interrupt subsystem's instruction sequence that stores to the interrupt controller.

**DMA completion**
Often an IOA DMA writes data to system memory it shares with its device driver, and then posts an interrupt. Because the interrupt can bypass the DMA data written by the IOA, at the time the interrupt is handled the data may not yet be in system memory.

The RPA specifies that all DMA write data in the PHB buffers is written to memory prior to returning data in response to a load operation that occurs after the DMA write operations. This means the device driver **must** do a PIO load to its IOA before accessing the system memory locations that are the target of the DMA write. Such a load naturally appears at the beginning of most interrupt service routines, in the form of a load from the IOA's interrupt status register to determine the type of interrupt being presented.

But the device driver needs to guard against reordered or speculatively executed accesses to the shared memory, or else its accesses to the shared memory block can still return stale data. The load from the IOA that serves to flush the data to memory must be performed prior to any loads from the shared memory block.

The best solution is to use the io_flush construct. The isync in the io_flush macro follows a conditional branch based on the value being loaded. In order for the conditional branch to complete, the load from the IOA must complete. And the isync guarantees the subsequent load (from the system memory address) will be performed after the load required by the conditional branch.

```c
interrupt_handler:
    val = load from IOA interrupt status register
    io_flush(val);
    cmd_result = load from system memory address
                 that was written by IOA prior to interrupting
```

Depending on the interrupt handler's logic flow, it may be possible to use one io_flush macro invocation to resolve this DMA completion problem as well as the previously described End Of Interrupt race.

**Indexed command completion**

This somewhat complicated example is fundamentally similar to the "Waiting on Flag" and "DMA Completion" examples already described. Some device drivers follow a model in which there are a list of status buffers in system memory. The IOA updates a status buffer each time a command completes, and then updates an ioa_index variable in shared memory to indicate the latest status buffer written by the IOA. The device driver maintains a separate dd_index variable indicating the last status buffer the driver has processed. The driver compares the values of the two index variables, to see if another command has completed and it needs to process the results that have been written (by the IOA) to the next status buffer:
volatile int ioa_index; /* written by IOA */
int dd_index; /* written by driver */

volatile struct cmd_status buffer[NUM_STATUS_BUFS]; /* written by IOA */
struct cmd_status status_var; /* local variable */

while (dd_index != ioa_index) {
    status_var = buffer[dd_index];
    <process status_var>
    next_index(dd_index); /* increment dd_index, handle wrapping back to start of list */
}

This type of logic flow is also exposed to reordering and speculative execution. The IOA's two DMA writes (of the cmd_status information followed by the new ioa_index) will be ordered, per the RPA. But the processor must not be allowed to read the cmd_status location until after it has loaded and compared the ioa_index information, or it may load stale cmd_status information. The flow to be avoided is:

read dd_index
read buffer[dd_index]     IOA DMA write to buffer[dd_index]
IOA DMA write to ioa_index
read ioa_index
compare dd_index and ioa_index, which no longer match

The solution again is to follow the load requiring ordering (the load of ioa_index) with a compare-branch-isync sequence, such as the one provided by io_flush:

while (dd_index != ioa_index) {
    io_flush(ioa_index);
    status_var = buffer[dd_index];
    <process status_var>
    next_index(dd_index);
}

Note that depending on the specific assembly sequence generated by the compiler, the compare and branch may already be present, in which case only the isync would need to be inserted.

**Device control timing loops**

Another interesting case occurs when a driver needs to delay between PIOs to a device. An example might be doing a PIO store to cause an IOA to assert some signal to child devices, then the driver needs to delay to ensure the signal is asserted some minimum period before it issues a second PIO store to deassert the signal. The architecture specifies the order of accesses, but avoids describing any operations with respect to real time. So the two PIO stores will be delivered to the device in order, but the two stores could be delivered on sequential cycles (i.e., no delay between the stores). The solution is similar to the io_flush approach - a load to the same device (with eieio used to order the load if it's not to the same device address as the prior store) and a
**data dependency** need to be inserted after the initial store and prior to the delay code, so that the driver does not start its delay until it is sure the initial store has been completed.

**Example combining sync and lwsync**

Assume an I/O programming model where the device driver is creating a list of commands to be executed by an I/O device, in an environment where commands may be added to the list at the same time that the I/O device is fetching commands from the list.

Assume that the device driver has built a list of five commands in system memory, executed a heavyweight `sync` instruction, and then performed a store to device memory to cause the device to initiate processing of the command list.

Before the device has fetched all the commands, the device driver begins to append another three commands to the list. Assume each command element is composed of two doublewords of command/data and a doubleword pointer to link to the next element (a null pointer indicates the end of the list). To ensure that the command read by the device is consistent, the device driver must execute three stores to update the new list element, then execute a `sync` before executing the store that updates the pointer in the previous element to add the new element to the list. Because all these accesses are to system memory, a `lwsync` is the preferred instruction.

After adding the three elements to the command list, the device driver must execute a heavyweight `sync` instruction before executing the store to device memory that informs the device that additional commands have been appended to the list. Note that in this example the device may have already started fetching the three new command elements before the second store to device memory, but the `lwsync` instruction ensured the device would read consistent values. The second store to device memory is needed to handle the case that the three new commands were not seen during the device's initial pass through the command list, and a heavyweight `sync` is again required to enforce ordering between the system memory updates for the command list and the store to the device to reinitiate processing of the list.

**Using the C "volatile" attribute**

Device driver programs often require accesses to shared storage (that is, storage that is read or written by other program threads or I/O devices). To ensure correct program operation, it is worth emphasizing the importance for device driver writers to correctly use the C "volatile" attribute. You usually need the "volatile" attribute:

1. when the value in some variable (for example, a location in system or device memory) is examined or set asynchronously, or
2. when it is necessary to ensure that every load from and store to device memory is performed in device memory, due to side-effects that result from the accesses

So the "volatile" attribute is usually required with memory-mapped I/O. Additional examples where "volatile" must be used include:

- locations shared between processes (or processors)
• storage-based clocks or timers
• locations or variables accessed or modified by signal handlers (which the primary code examines)
• a lock cell (a special case of the first case)

The IBM compiler generates a load or store appropriately for every reference to a "volatile" memory location and does not reorder such references. The result is that the contents of a volatile object are read from memory each time its value is used, and are written back to memory each time the program modifies the object.

It's usually wrong to copy a pointer to a volatile object to a pointer to a non-volatile object, and then use the copy. The code sequence below is incorrect, and caused a bug in a Microchannel driver in an older version of AIX. The problem is that a "pointer to volatile char" is assigned to two other "pointer to char" variables, which are then used to touch the device registers. Because these pointers are not declared with the volatile attribute, their order was swapped by the compiler's instruction scheduler, and resulted in buggy behavior by the adapter.

The specific sequence was:

```
volatile char *pptr;
char *poscmd, *posdata, poll;
    :
    :
pptr = ....
poscmd = pptr + 6;
posdata = pptr + 3;
pptr += 7;
*poscmd = 0x47;       /*  This store gets interchanged with */
poll = *posdata;      /*  this load by the compiler         */
```

**Conclusion**

We've described and illustrated how the underlying PowerPC storage architecture can impact AIX application and device driver code. The underlying architecture hasn't changed, but newer implementations based on the POWER4 processor are more aggressive in exploiting the architecture for performance gains. You need to examine any implicit software assumptions about order of execution and I/O latencies in your code and change it if the implicit assumptions don't match the architectural guarantees.

**Appendix A: Implementation notes**

The C and C++ compilers from IBM provide built-in functions (**__iospace_sync** and **__iospace_eieio**) to generate sync and eieio instructions inline. The compiler will use a sync for **__iospace_eieio()** if the compile mode is not PowerPC (for example, the compiler generates eieio for **__iospace_eieio()** only when -qarch=ppc). New releases of these compilers are planned to provide a richer set of built-ins which will permit convenient generation of **isync** and **lwsync**.

Assembler-coded functions can be written which contain just the required instruction and a return.

The AIX header file **inline.h** provides the appropriate symbols and pragmas to inline the instructions, but it's not a shipped header, so it's available only to developers with access to the
AIX build environment. However, the techniques used in inline.h can be replicated. It uses a facility of the compiler which permits generation of any instruction in-line. This facility is known as "mc_func" (machine-code function), and requires specification of the instruction(s) in hex. This facility has been used to replace a call to an external assembler-coded function with the instruction(s) contained in the function, without changing the program semantics of the call.

For the convenience of external developers (who don’t have access to inline.h), here are the file contents that relate to the material that we’ve covered. Simply copy these lines into your C source to use the mc_func facility to inline calls to isync, eieio, sync, lwsync, or io_flush:

```c
void isync(void);
void eieio(void);
void SYNC(void);
void LWSYNC(void);

#pragma mc_func isync { "4c00012c" } /* isync */
#pragma mc_func eieio { "7c0006ac" } /* eieio */
#pragma mc_func SYNC { "7c0004ac" } /* sync */
#pragma mc_func LWSYNC { "7c2004ac" } /* lwsync */

#pragma reg_killed_by isync
#pragma reg_killed_by eieio
#pragma reg_killed_by SYNC
#pragma reg_killed_by LWSYNC

/*
 * The following is used on PCI/PowerPC machines to make sure that any state
 * updates to I/O ports have actually been flushed all the way to the device.
 * A read from something on the same bridge, an operation on the value read,
 * and a conditional branch and an isync are needed to guarantee this. This
 * inline will do the last three parts. More generically, the inline can also
 * be used to ensure that the load of the val parameter is performed
 * before storage accesses subsequent to the io_flush sequence are performed.
 */
void io_flush(int val);

#pragma mc_func io_flush { 
 "7c031800" /* not_taken: cmp cr0, r3, r3 */ \ 
 "40a2fffc"  /* bne- cr0, not_taken */ \ 
 "4c00012c"  /* isync   */ \ 
 } 
#pragma reg_killed_by io_flush     gr3,cr0
```

Appendix B: Combining, merging and collapsing

Besides ordering, you should also consider the possibilities of combining, merging, and collapsing. The following definitions come from PCI, but it's useful to think of the concepts at all stages between the processor and the I/O adapter. In PCI, combining occurs when sequential, non-overlapping writes are converted into a single (multiword) transaction. Byte merging occurs when a sequence of individual memory writes are merged into a single word access, and it's not permitted if one of the bytes is accessed by more than one of the writes in the sequence. The implied order is preserved in combining, but in byte merging the order is not necessarily preserved (a sequence in which bytes 3, 1, 0, and 2 in the same word-aligned address are written can be byte merged into a single transaction). Finally, collapsing occurs if a sequence of memory writes to the same location are collapsed into a single bus transaction.
Here is a brief summary of the rules for combining, merging, and collapsing.

Combining of sequential writes (non-overlapping) where the implied ordering is maintained:

- Processor architecture - allowed except if separated by a sync or if guarded and separated by an eieio (must be sequential writes from an ordering standpoint)
- PCI architecture - allowed unconditionally to PCI memory space and encouraged, not allowed to I/O or config; implied ordering must be maintained
- RPA architecture - says everything in the coherency domain will implement the PowerPC semantics - that includes Hubs but not bridges
- Hub implementations - sync or eieio to guarded space will prevent combining at Hub since they are part of coherency domain
- Bridge implementations - Combining appropriate and encouraged by PCI architecture to PCI memory address space, not allowed to I/O or config (per PCI architecture)
- Device drivers - will have to prevent it to the PCI memory space by use of some method other than sync or eieio (since bridges allowed to combine) if the PCI device cannot handle it and will have to prevent it to PCI I/O space by use of sync or eieio to guarded space (config space not a problem due to use of RTAS call)
- Firmware - needs to use sync or eieio to guarded space to the config space

Merging of a sequence of individual memory writes into a single word (to be merged, the individual memory writes cannot overlap and cannot address the same byte, but reordering is allowed):

- Processor architecture - processor architecture does not allow this if the operations are non-sequential or overlapping, or if separated by a sync or eieio with the guarded bit on, but does allow it if sequential, non-overlapping, and not separated by sync or eieio with guarded bit
- PCI architecture - disallowed to non-prefetchable memory and to I/O spaces and config space, allowed to prefetchable memory spaces; PCI architecture doesn't require that PHBs implement non-prefetchable versus non-prefetchable spaces
- RPA architecture - doesn't require the bridges implement prefetchable versus non-prefetchable
- Bridge implementations - given the last two bullets, our platforms had better not implement merging to any of the PCI address spaces
- Device drivers - device driver has to prevent processor from merging if to PCI non-prefetchable memory space or to PCI I/O space by use of sync or eieio to guarded space (config space not a problem due to use of RTAS call)
- Firmware - needs to use sync or eieio to guarded space to access the config space

You don't have to worry about the collapsing of a sequence of individual memory writes to the same location into one bus transaction. Each individual store in a sequence of stores to the same location in device memory will be delivered to the adapter (for example, a sequence of stores to a FIFO buffer will produce the expected result). See the section above that discusses "volatile" objects.