Hardware Cache Coherent Input/Output

Hardware cache coherent I/O is a new feature of the PA-RISC architecture that involves the I/O hardware in ensuring cache coherence, thereby reducing CPU and memory overhead and increasing performance.

by Todd J. Kjos, Helen Nusbaum, Michael K. Traynor, and Brendan A. Voge

A new feature, called hardware cache coherent I/O, was introduced into the HP PA-RISC architecture as part of the HP 9000 J/K-class program. This feature allows the I/O hardware to participate in the system-defined cache coherency scheme, thereby offloading the memory system and processors of unnecessary overhead and contributing to greater system performance. This paper reviews I/O data transfer, introduces the concept of cache coherent I/O from a hardware perspective, discusses the implications for HP-UX* software, illustrates some of the benefits realized by HP's networking products, and presents measured performance results.

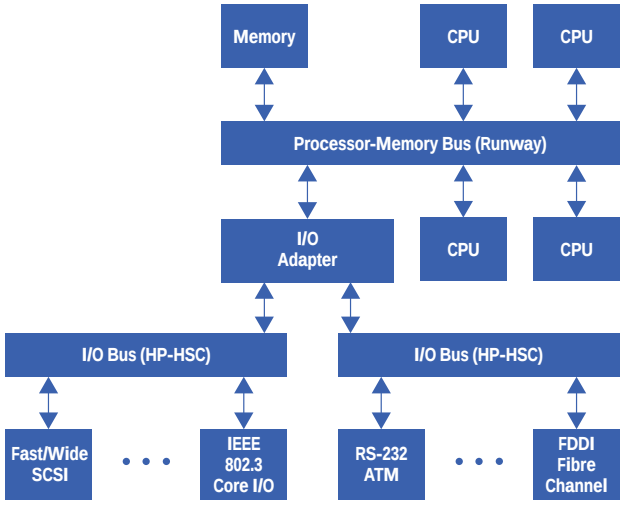
I/O Data Transfer

To understand the impact of the HP 9000 J/K-class coherent I/O implementation, it is necessary to take a step back and get a high-level view of how data is transferred between I/O devices and main memory on HP-UX systems.

There are two basic models for data transfer: direct memory access (DMA) and programmed I/O (PIO). The difference between the two is that a DMA transfer takes place without assistance from the host processor while PIO requires the host processor to move the data by reading and writing registers on the I/O device. DMA is typically used for devices like disks and LANs which move large amounts of data and for which performance is important. PIO is typically used for low-cost devices for which performance is less important, like RS-232 ports. PIO is also used for some high-performance devices like graphics frame buffers if the programming model requires it.

All data transfers move data either to main memory from an I/O device (inbound) or from main memory to an I/O device (outbound). These transfers require one or more transactions on each bus between the I/O device and main memory. Fig. 1 shows a typical PA-RISC system with a two-level bus hierarchy. PA-RISC processor-to-memory buses typically support transactions in sizes that are powers of 2, up to 32 bytes, that is, READ4, WRITE4, READ8, WRITE8, READ16, WRITE16, READ32, WRITE32, where the number refers to the number of bytes in the transaction. Each transaction has a master and a slave; the master initiates the transaction and the slave must respond. Write transactions move data from the master to the slave, and read transactions cause the slave to respond with data for the master.

Fig. 1. *Typical PA-RISC system with a two-level bus hierarchy.*



The processor is always the master for PIO transactions to the I/O device. An I/O device is always the master for a DMA transaction. For example, if a software device driver is reading (PIO) a 32-bit register on the fast/wide SCSI device shown in Fig. 1, it causes the processor to master a READ4 transaction to the device, which results in the I/O adapter mastering a READ4 transaction on the I/O bus, where the fast/wide SCSI device responds with the four bytes of data. If the Fibre Channel

interface card is programmed to DMA transfer 4K bytes of data from memory to the disk, it will master 128 READ32 transactions to get the data from memory. The bridge forwards transactions in both directions as appropriate.

Because PIO transactions are not in memory address space and are therefore not a coherency concern, the rest of this article discusses DMA transactions only. The coherent I/O hardware has no impact at all on I/O software device drivers that interact with devices via PIO exclusively.

Hardware Implications

Cache memory is defined as a small, high-speed block of memory located close to the processor. On the HP PA 7200 and PA 8000 processors, a portion of the software virtual address (called the virtual index) is used as the cache lookup. Main memory is much larger and slower than cache. It is accessed using physical addresses, so a virtual-to-physical address translation must occur before issuing any request to memory. Entries in the PA 7200 and PA 8000 caches are stored in lines of 32 bytes. Since data that is referenced once by the processor is likely to be referenced again and nearby data is likely to be accessed as well, the line size is selected to optimize the frequency with which it is accessed while minimizing the overhead associated with obtaining the data from main memory. The cache contains the most recently accessed lines, thereby maximizing the rate at which processor-to-memory requests are intercepted, ultimately reducing latency.

When a processor requests data (by doing loads), the line containing the data is copied from main memory into the cache. When a processor modifies data (by doing stores), the copy in the cache will become more up-to-date than the copy in memory. HP 9000 J/K-class products resolve this stale data problem by using the snoopy cache coherency scheme defined in the Runway bus protocol. Each processor monitors all Runway transactions to determine whether the virtual index requested matches a line currently stored in its cache. This is called "snooping the bus." A Runway processor must own a cache line exclusively or *privately* before it can complete a store. Once the store is complete, the cache line is considered *dirty* relative to the stale memory copy. To maximize Runway bus efficiency, processors are not required to write this stale data back to memory immediately. Instead, the write-back operation occurs when the cache line location is required for use by the owning processor for another memory access. If, following the store but before the write-back, another processor issues a read of this cache line, the owning processor will snoop this read request and respond with a cache-to-cache copy of the updated cache line data. This data is then stored in the requesting processor's cache and main memory.

Since the I/O system must also read (output) and modify (input) memory data via DMA transactions, data consistency for the I/O system must be ensured as well. For example, if the I/O system is reading data from memory (for outbound DMA) that is currently dirty in a processor's cache, it must be prevented from obtaining a stale, out-of-date copy from memory. Likewise, if the I/O system is writing data to memory (for inbound DMA), it must ensure that the processor caches acquire this update. The optimum solution not only maintains consistency by performing necessary input/output operations while preventing the transfer of any stale copies of data, but also minimizes any interference with CPU cycles, which relate directly to performance.

Cache coherence refers to this consistency of memory objects between processors, memory modules, and I/O devices. HP 9000 systems without coherent I/O hardware must rely on software to maintain cache coherency. At the hardware level, the I/O device's view of memory is different from the processor's because requested data might reside in a processor's cache. Typically, processor caches are virtually indexed while I/O devices use physical addresses to access memory. Hence there is no way for I/O devices to participate in the processor's coherency protocol without additional hardware support in the I/O adapter.

Some architectures have prevented stale data problems by implementing physically indexed caches, so that it is the physical index, not the virtual index, that is snooped on the bus. Thus, the I/O system is not required to perform a physical-to-virtual address translation to participate in the snoopy coherence protocol. On the HP 9000 J/K-class products, we chose to implement virtually indexed caches, since this minimizes cache lookup time by eliminating a virtual-to-physical address translation before the cache access.

Other architectures have avoided the output stale data problem by implementing write-through caches in which all processor stores are immediately updated in both the cache and the memory. The problem with this approach is its high use of processor-to-memory bus bandwidth. Likewise, to resolve the input stale data problem, many architectures allow cache lines to be marked as uncacheable, meaning that they can never reside in cache, so main memory will always have correct data. The problem with this approach is that input data must first be stored into this uncacheable area and then copied into a cacheable area for the processor to use it. This process of copying the data again consumes processor-to-memory cycles for nonuseful work.

Previous implementations of HP's PA-RISC processors circumvent these problems by making caches visible to software. On outbound DMA, the software I/O device drivers execute flush data cache instructions immediately before output operations. These instructions are broadcast to all processors and require them to flush their caches by writing the specified dirty cache lines back to main memory. After the DMA buffer has been flushed to main memory, the outbound operation can proceed and is guaranteed to have the most up-to-date data. On inbound DMA, the software I/O device drivers execute broadcast purge data cache instructions just before the input operations to remove the DMA buffer from all processor caches in the machine. The PA-RISC architecture's flush data cache and purge data cache instruction overhead is small compared to the

performance impact incurred by these other schemes, and the I/O hardware remains ignorant of the complexities associated with coherency.

I/O Adapter Requirements. The HP 9000 J/K-class products and the generation of processors they are designed to support place greater demands on the I/O hardware system, ultimately requiring the implementation of cache coherent I/O hardware in the I/O bus adapter, which is the bus converter between the Runway processor-memory bus and the HP-HSC I/O bus. The first of these demands for I/O hardware cache coherence came from the design of the PA 7200 and PA 8000 processors and their respective implementations of cache prefetching and speculative execution. The introduction of these features would have required software I/O device drivers to purge inbound buffers twice, once before the DMA and once after the DMA completion, thus doubling the performance penalty. Because aggressive prefetches into the DMA region could have accessed stale data after the purge but before the DMA, the second purge would have been necessary after DMA completion to cleanse stale data prefetch buffers in the processor. By designing address translation capabilities into the I/O adapter, we enable it to participate in the Runway snoopy protocol. By generating virtual indexes, the I/O adapter enables processors to compare and detect collisions with current cache addresses and to purge prefetched data aggressively before it becomes stale.

Another demand on the I/O adapter pushing it in the direction of I/O cache coherence came from the HP 9000 J/K-class memory controller. It was decided that the new memory controller would not implement writes of less than four words. (These types of writes would have required read-modify-write operations in the DRAM array, which have long cycle times and, if executed frequently, degrade overall main memory performance.) Because one-word writes occur in the I/O system, for registers, semaphores, or short DMA writes it was necessary that the I/O adapter implement a one-line-deep cache to buffer cache lines, so that these one-word writes could be executed by performing a coherent read private transaction on the Runway bus, obtaining the most recent copy of the cache line, modifying it locally in cache, and finally writing the modified line back to main memory. For the I/O adapter to support a cache on the Runway bus, it has to have the ability to compare processor-generated virtual address transactions with the address contained in its cache to ensure that the processors always receive the most up-to-date data.

To simplify the design, the I/O adapter implements a subset of the Runway bus coherency algorithm. If a processor requests a cache line currently held privately by the I/O adapter, the I/O adapter stalls the coherency response, finishes the read-modify-write sequence, writes the cache line back to memory, and then responds with COH_OK, meaning that the I/O adapter does not have a copy of this cache line. This was much simpler to implement than the processor algorithm, which on a conflict responds with COH_CPY, meaning that the processor has this cache line and will issue a cache-to-cache copy after modifying the cache line. Since the I/O adapter only has a one-line cache and short DMA register and semaphore writes are infrequent, it was felt that the simpler algorithm would not be a performance issue.

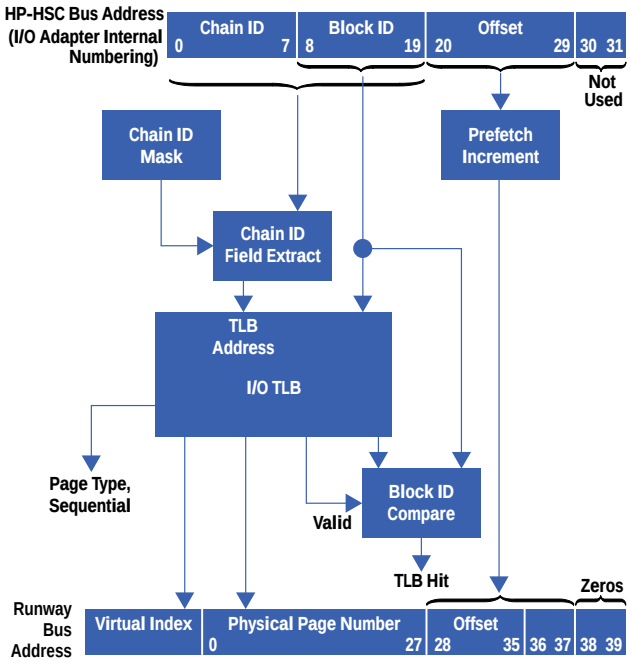
A final requirement for the I/O adapter is that it handle 32-bit I/O virtual addresses on the HP-HSC bus and a larger 40-bit physical address on the Runway bus to support the new processors. Thus a mapping function is required to convert all HP-HSC DMA transactions. This is done via a lookup table in memory, set up by the operating system, called the I/O page directory. With minor additional effort, I/O page directory entries were defined to provide the I/O adapter with not only the 40-bit physical address, but also the software virtual index. This provides all the information necessary for the I/O adapter to be a coherent client on the Runway bus. The I/O adapter team exploited the mapping process by implementing an on-chip TLB (translation lookaside buffer), which caches the most recent translations, to speed the address conversion, and by storing additional attribute bits into each page directory entry to provide clues about the DMA's page destination, thereby allowing further optimization for each HP-HSC-to-Runway transaction.

I/O TLB Access. The mechanism selected for accessing the I/O TLB both minimizes the potential for thrashing and is flexible enough to work with both large and small I/O systems. (Thrashing occurs when two DMA streams use the same TLB RAM location, each DMA transaction alternately casting the other out of the TLB, resulting in tremendous overhead and lower performance.) Ideally, each DMA stream should use a different TLB RAM location, so that only one TLB miss read is done per page of DMA.

We implemented a scheme by which the upper 20 bits of the I/O virtual address are available to be divided into a *chain ID* and a *block ID* (Fig. 2). The lower 12 bits of the address must be left alone because of the 4K-byte page size defined by the architecture. The upper address bits (chain ID) of the I/O virtual address are used to access the TLB RAM, and the remainder of the I/O virtual address (block ID) is used to verify a TLB hit (as a tag). This allows software to assign each DMA stream to a different chain ID and each 4K-byte block of the DMA to a different block ID, thus minimizing thrashing between DMA streams.

A second feature of this scheme is that it helps limit the overhead of the I/O page directory. Recall that the I/O page directory contains all active address translations and must be memory-resident. I/O page directory size is equal to the size of one entry times 2^k, where k is the number of chain ID bits plus the number of block ID bits. The division between the chain ID and the block ID is programmable, as is the total number of bits (k), so software can reduce the memory overhead of the I/O page directory for systems with smaller I/O subsystems if we guarantee that the leading address bits are zero for these smaller systems.

Fig. 2. *TLB translation scheme. The upper address bits (chain ID) of the I/O virtual address are used to access the TLB RAM, and the remainder of the I/O virtual address (block ID) is used to verify a TLB hit (as a tag).*



If the translation is not currently loaded in the I/O adapter's I/O TLB, the I/O adapter reads the translation data from the I/O page directory and then proceeds with the DMA. Servicing the TLB miss does not require processor intervention, although the I/O page directory entry must be valid before initiating the DMA.

Attribute Bits. Each mapping of a page of memory has attribute bits (or clues) that allow some control over how the DMA is performed. The page attribute bits control the enabling and disabling of prefetch for reads, the enabling and disabling of atomic mode, and the selection of fast DMA or safe DMA.

Prefetch enable allows the I/O adapter to prefetch on DMA reads, thus improving outbound DMA performance. Because the I/O adapter does not maintain coherency on prefetch data, software must only enable prefetching on pages where there will be no conflicts with processor-held cache data. Prefetch enable has no effect on inbound DMA writes.

Atomic or locked mode allows a DMA transfer to own all of memory. While an atomic mode transfer is in progress, processors cannot access main memory. This feature was added to support PC buses that allow locking (ISA, EISA). The HP-HSC bus also supports this functionality. In almost all cases, atomic mode is disabled, because it has tremendous performance effects on the rest of the system.

The fast/safe bit only has an effect on half-cache-line DMA writes. However, many I/O devices issue this type of 16-byte write transaction. In safe mode, the write is done as a read-modify-write transaction in the I/O adapter cache, which is relatively low in performance. In fast mode, the write is issued as a WRITE16_PURGE transaction which is interpreted by the processors as a purge cache line transaction and a write half cache line transaction to memory. The fast/safe DMA attribute is used in the following way. In the middle of a long inbound DMA, fast mode is used: the processor's caches are purged while DMA data is moved into memory. This is acceptable because the processor should not be modifying any cache lines since the DMA data would overwrite the cache data anyway. However, at the beginning or end of a DMA transfer, where the cache line might be split between the DMA sequence and some other data unrelated to the DMA sequence, the DMA transaction needs to preserve this other data, which might be held private-dirty by a processor. In such cases, the safe mode is used. This feature allows the vast majority of 16-byte DMA writes to be done as WRITE16_ PURGEs, which have much better performance than read-modify-writes internal to the I/O adapter cache. This is the only half-cache-line transaction the memory subsystem supports. All other memory transactions operate on full cache lines.

HP-UX Implementation

Cache coherent I/O affects HP-UX I/O device drivers. Although the specific algorithm is different for each software I/O device driver that sets up DMA transactions, the basic algorithm is the same. For outbound DMA on a system without coherent I/O hardware, the driver must perform the following tasks:

- Flush the data caches to make sure memory is consistent with the processor caches.
- Convert the processor virtual address to a physical address.
- Modify and flush any control structures shared between the driver and the I/O device.

- Initiate the DMA transfer by programming the device to move the data from the given physical address, using a device-specific mechanism.

For inbound DMA the algorithm is similar:

- Purge the data cache to prevent stale cache data from being written over DMA data.
- Convert the processor virtual address to a physical address.
- Modify and flush any control structures shared between the driver and the I/O device.
- Initiate the DMA transfer by programming the device to move the data to the given physical address, using a device-specific algorithm.

When the DMA completes, the device notifies the host processor via an interrupt, the driver is invoked to perform any post-DMA cleanup, and high-level software is notified that the transfer has completed. For inbound DMA the cleanup may include purging the data buffer again in case the processor has prefetched the old data.

To support coherent I/O hardware, changes to this basic algorithm could not be avoided. Since coherent I/O hardware translates 32-bit I/O virtual addresses into processor physical addresses that may be wider than 32 bits, I/O devices must be programmed for DMA to I/O virtual addresses instead of physical addresses. Also, since coherency is maintained by the coherent I/O adapter, no cache flushes or purges are necessary and should be avoided for performance reasons. To allow drivers to function properly regardless of whether the system has coherent I/O hardware, HP-UX services were defined to handle the differences transparently. There are three main services of interest to drivers: map() is used to convert a virtual address range into one or more I/O virtual addresses, unmap() is used to release the resources obtained via map() once the transfer is complete, and dma_sync() is used to synchronize the processor caches with the DMA buffers, replacing explicit cache flush and purge services. These services are discussed in more detail below.

Drivers had to be modified where one of the following assumptions existed:

- *Devices use processor physical addresses to access memory.* This assumption is still true for noncoherent systems, but on HP 9000 J/K-class systems I/O virtual addresses must be used. The map() service transparently returns a physical address on noncoherent systems and an I/O virtual address on coherent systems.
- *Cache management must be performed by software.* This is still true for noncoherent systems, but on coherent systems flushes and purges should be avoided for performance reasons. The dma_sync() service performs the appropriate cache synchronization functions on noncoherent systems but does not flush or purge on coherent systems.
- *The driver does not have to keep track of any DMA resources.* Drivers now have to remember what I/O virtual addresses were allocated to them so they can call unmap() when the DMA transfer is complete.

To accommodate these necessary modifications, the software model for DMA setup has been changed to:

- Synchronize the caches using dma_sync().
- Convert the processor virtual address to an I/O virtual address using the map() service.
- Initiate the DMA transfer via a device-specific mechanism.
- Call the unmap() service to release DMA resources when the DMA transfer is complete.

On noncoherent systems, this has the same effect as before, except that the driver doesn't know whether or not the cache was actually flushed or whether the I/O virtual address is a physical address.

For drivers that rely entirely on existing driver services to set up DMA buffers (like most EISA drivers), no changes were needed since the underlying services were modified to support coherent I/O.

Driver Services: map and unmap. The map() service and its variants are the only way to obtain an I/O virtual address for a given memory object. Drivers cannot assume that a buffer can be mapped in a single call to map() unless the buffer is aligned on a cache line boundary and does not cross any page boundaries. Since it is possible that multiple I/O virtual addresses are needed to map a DMA buffer completely, map() should be called within a loop as shown in the following C code fragment:

```
/*  Function to map an outbound DMA buffer
 *  Parameters:
 *          isc: driver control structure
 *          space_id: the space id for the buffer
 *          virt_addr: the virtual offset for the buffer
 *          buffer_length: the length (in bytes) of the buffer
 *          iovecs (output): an array of address/length I/O
 *              virtual address pairs
 *
 *  Output:
 *          Returns the number of address/length I/O virtual
 *            address pairs (–1 if error)
 *
```

```
 */
int my_buffer_mapper(isc,space_id,virt_addr,buffer_length,iovecs)
struct isc_table_type  *isc;
int vec_cnt;
struct iovec  *iovecs;
{
        int vec_cnt = 0;
        struct iovec hostvec;
        int retval;

        /*  Flush cache (on noncoherent systems)  */
        dma_sync(space_id,virt_addr,buffer_length,IO_SYNC_FORCPU);

        /*  Set up input for map()  */
        hostvec–>iov_base =  virt_addr;
        hostvec–>iov_len   =  buffer_length;

        do {
            /*  Map the buffer  */
            retval  =  wsio_map(isc,NULL,0,space_id,virt_addr,
                              &hostvec,iovecs);

            if (retval  >=  0) {
                  /*  Mapping was successful point to the
                   *  next I/O virtual address vector.  Note:
                   *  hostvec was modified by map() to point to
                   *  unmapped portion of the buffer.
                   */
                  vec_cnt++;
                  iovecs++;
            }
        } while (retval  >  0);
        /*  Check for any errors  */
        if (retval < 0) {
            while (vec_cnt) {
                      wsio_unmap(isc, iovec[vec_cnt].iov_base);
                      vec_cnt—;
            }
        vec_cnt—;
        }
        return(vec_cnt);
}
```

In this case the map() variant wsio_map() is used to map a buffer. When a mapping is successful, the driver can expect that the virtual host range structure has been modified to point to the unmapped portion of the DMA buffer. The wsio_map() service just converts the isc parameter into the appropriate token expected by map() to find the control structure for the correct I/O page directory. The calling convention for map() is:

map(token,map_cb,hints,range_type,host_range,io_range),

where token is an opaque value that allows map() to find the bookkeeping structures for the correct I/O page directory. The map_cb parameter is an optional parameter that allows map() to store some state information across invocations. It is used to optimize the default I/O virtual address allocation scheme (see below). The hints parameter allows drivers to specify page attributes to be set in the I/O page directory or to inhibit special handling of unaligned DMA buffers. The host_range contains the virtual address and length of the buffer to be mapped. As a side effect, the host_range is modified by map() to point to the unmapped portion of the buffer (or the end of the buffer if the entire range was mapped). The io_range is set up by map() to indicate the I/O virtual address and the length of the buffer that was just mapped. The range_type is usually the space ID for the virtual address, but may indicate that the buffer is a physical address.

All I/O virtual addresses allocated via map() must be deallocated via unmap() when they are no longer needed, either because there was an error or because the DMA completed. The calling convention for unmap() is:

unmap(token,io_range).

The map() service uses the following algorithm to map memory objects into I/O virtual space:

- Allocate an I/O virtual address for the mapping.
- Initialize the I/O page directory entry with the appropriate page attributes. The page directory entry will be brought into the I/O translation lookaside buffer when there is a miss.
- Update the caller's range structures and return the number of bytes left to map.

The first two steps are discussed separately below.

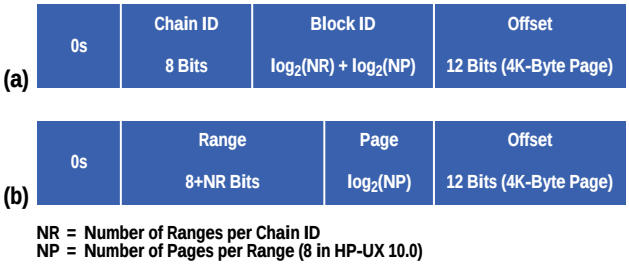
I/O Virtual Address Allocation Policies. As mentioned above, if several DMA streams share a chain ID, there is a risk that performance will suffer significantly because of thrashing. Two allocation schemes that appeared to eliminate thrashing are:

- Allocate a unique chain ID to every DMA stream.
- Allocate a unique chain ID to every HP-HSC guest.

In the I/O adapter there are a total of 256 I/O translation lookaside buffer entries, and therefore there are 256 chain IDs to allocate. The first allocation scheme is unrealistic because many more than 256 DMA streams can be active from the operating system's point of view. For instance, a single networking card can have over 100 buffers available for inbound data at a given time, so with only three networking cards the entire I/O TLB would be allocated. Thrashing isn't really a problem unless individual transactions are interleaved with the same chain ID, so on the surface it may appear that the second allocation scheme would do the trick (since most devices interleave different DMA streams at fairly coarse granularity like 512 or 1K bytes). Unfortunately, the second scheme has a problem in that some devices (like SCSI adapters) can have many large DMA buffers, so all current outstanding DMA streams cannot be mapped into a single chain ID. One of the goals of the design was to minimize the impact on drivers, and many drivers had been designed with the assumption that there were no resource allocation problems associated with setting up a DMA buffer. Therefore, it was unacceptable to fail a mapping request because the driver's chain ID contained no more free pages. The bookkeeping involved in managing the fine details of the individual pages and handling overflow cases while guaranteeing that mapping requests would not fail caused us to seek a solution that would minimize (rather than eliminate) the potential for thrashing, while also minimizing the bookkeeping and overhead of managing the chain ID resource.

What we finally came up with was two allocation schemes: a default I/O virtual address allocator which is well-suited to mass storage workloads (disk, tape, etc.) and an alternate allocation scheme for networking-like workloads. It was observed early on that there are some basic differences in how some devices behave with regard to DMA buffer management. Networking drivers tend to have many buffers allocated for inbound DMA, but devices tend to access them sequentially. Therefore, networking devices fit the model very well for the second allocation scheme listed at the beginning of this section, except that it is likely that multiple chain IDs will be necessary for each device because of the number of buffers that must be mapped at a given time. Mass storage devices, however, may have many DMA buffers posted to the device, and no assumptions can be made about the order in which the buffers will be used. This behavior was dubbed *nonsequential*. It would have resulted in excessive overhead in managing the individual pages of a given chain ID if the second scheme listed at the beginning of this section had been implemented. To satisfy the requirements for both sequential and nonsequential devices, it was decided to manage *virtual* chain IDs called *ranges* instead of chain IDs. This allows the operating system to manage the resource independent of the physical size of the I/O translation lookaside buffer. Thrashing is minimized by always allocating free ranges in order, so that thrashing cannot occur unless 256 ranges are already allocated. Therefore, software has a slightly different view of the I/O virtual address than the hardware, as shown in Fig. 3.

Fig. 3. *(a) Hardware's (I/O TLB's) view of I/O virtual addresses. (b) Software's view of I/O virtual addresses.*



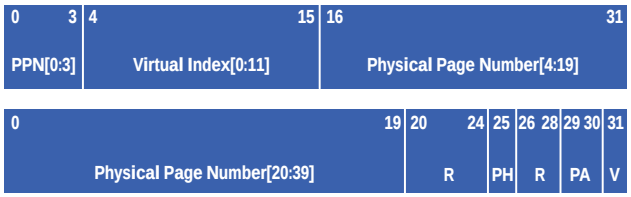
With this definition of an I/O virtual address, software is not restricted to 256 resources, but instead can configure the number of resources by adjusting the number of ranges per chain ID. For the HP-UX 10.0 implementation, there are eight pages per range so that up to 32K bytes can be mapped with a single range. The main allocator keeps a bitmap of all ranges, but does not manage individual pages.

A mass storage driver will have one of these ranges allocated whenever it maps a 32K-byte or smaller buffer (assuming the buffer is aligned on a page). For large buffers (>32K bytes), several ranges will be allocated. When the DMA transfer is complete, the driver unmaps the buffer and the associated ranges are returned to the pool of free resources.

All drivers use the default allocator unless they notify the system that an alternate allocator is needed. A driver service called set_attributes allows the driver to specify that it controls a sequential device and therefore should use a different allocation scheme. In the sequential allocation scheme used by networking drivers, the driver is given permanent ownership of ranges and the individual pages are managed (similar to the second scheme above). When a networking driver attempts to map a page and the ranges it owns are all in use, the services use the default allocation scheme to gain ownership of another range. Unlike the default scheme, these ranges are never returned to the free pool. When a driver unmaps the DMA buffer, the pages are marked as available resources only for that driver.

When unmap() is called to unmap a DMA buffer, the appropriate allocation scheme is invoked to release the resource. If the buffer was allocated via the default allocation scheme then unmap() purges the I/O TLB entry using the I/O adapter's direct write command to invalidate the entry. The range is then marked as available. If the sequential allocation scheme is used then the I/O TLB is purged using the I/O adapter's purge TLB command each time unmap() is called.

Fig. 4. *An I/O page directory entry.*



PPN[0:3] = Physical Page Number[0:3]
R = Reserved Field (All Bits Are Set to Zero in Reserved Fields)
PH = Prefetch Hint Bit (Enable/Disable Prefetch)
PA = Page Attribute Bits (Disable/Enable Atomic; Fast/Safe DMA)
V = Valid Indicator

I/O Page Directory Entry Initialization. Once map() has allocated the appropriate I/O virtual address as described above, it will initialize the corresponding entry in the I/O page directory. Fig. 4 shows the format of an I/O page directory entry. To fill in the page directory entry, map() needs to know the physical address, the virtual index in the processor cache, whether the driver will allow the device to prefetch, and the page type attributes. The physical address and virtual index are both obtained from the range_type and host_range parameters by using the LPA and LCI instructions, respectively. The LCI (load coherence index) instruction was defined specifically for coherent I/O services to determine the virtual index of a memory object. The page type attributes are passed to map() via the hints parameter. Hints that the driver can specify are:

- IO_SAFE. Causes the safe page attribute to be set.
- IO_LOCK. Causes the lock (atomic) page attribute to be set.
- IO_NO_SEQ. Causes the prefetch enable page attribute to be cleared.

Refer to the section "*Attribute Bits*" in this article for details of how the I/O adapter behaves for each of these driver hints. Once the I/O page directory has been initialized, the buffer can be used for DMA.

Other hints are:

- IO_IGN_ALIGNMENT. Normally the safe bit will be set automatically by map() for buffers that are not cache line aligned or that are smaller than a cache line. This flag forces map() to ignore cache line alignment.
- IO_CONTIGUOUS. This flag tells map() that the whole buffer must be mapped in a single call. If map() cannot map the buffer contiguously then an error is returned (this hint implies IO_IGN_ALIGNMENT).

Driver Impact. The finite size of the I/O page directory used for address translations posed some interesting challenges for drivers.

Drivers must now release DMA resources upon DMA completion. This requires more state information than drivers had previously kept. Additionally, some drivers had previously set up many (thousands) of memory objects, which I/O adapters need to access. Mapping each of these objects into the I/O page directory individually could easily consume thousands of entries. Finally, the default I/O page directory allocation policies would allocate several entries to a driver at a time, even if the driver only requires a single translation.

In the case where drivers map hundreds or thousands of small objects, the solution requires the driver code to be modified to allocate and map large regions of memory and then break it down into smaller objects. For example, if a driver individually allocates and maps 128 32-byte objects, this would require at least 128 I/O page directory entries. However, if the driver allocates one page (4096 bytes) of data, maps the whole page, and then breaks it down into 128 32-byte objects, only one I/O page directory entry is required.

Another solution is to map only objects for transactions that are soon to be started. Some drivers have statically allocated and mapped many structures, consuming large numbers of I/O page directory entries, even though only a few DMA transactions were active at a time. Dynamically mapping and unmapping objects on the fly requires extra CPU bandwidth and driver state information, but can substantially reduce I/O page directory utilization.

Networking-Specific Applications

The benefits of the selected hardware I/O cache coherence scheme become evident when examining networking applications.

High-speed data communication links place increased demands on system resources, including CPU, bus, and memory, which must carry and process the data. Processing the data that these links carry and the applications for which they will be used requires that resource utilization, measured both on a per-byte and on a per-packet basis, be reduced. Additionally, the end-to-end latency (the time it takes a message sent by an application on one system to be received by an application on another system) must be reduced from hundreds of microseconds to tens of microseconds.

Cache coherent I/O, scatter-gather I/O, and copy-on-write I/O all offer reduced resource consumption or reduced latency or both. They do this by reducing data movement and processor stalls and by simplifying both the hardware and software necessary to manage complex DMA operations.

Cache coherent I/O reduces the processor, bus, and memory bandwidth required for each unit of data by eliminating the need for the processors to manage the cache and by reducing the number of times data must cross the memory bus. The processor cycles saved also help to reduce per-packet latency.

The I/O adapter's address translation facility can be used to implement scatter-gather I/O for I/O devices that cannot efficiently manage physically noncontiguous buffers. Previously, drivers needed to allocate large, physically contiguous blocks of RAM for such devices. For outbound I/O, the driver would copy the outbound data into such a buffer. The mapping facility allows the driver to set up virtually contiguous mappings for physically scattered, page-sized buffers. The I/O device's view of the buffer is then contiguous. This is done by allocating the largest range that the I/O mapping services allow (32K bytes, currently), then using the remap() facility to set up a translation for each physical page in a DMA buffer. Using this facility reduces the processing and bus bandwidth necessary, and the associated latencies, for moving noncontiguous data. Requiring only a single address/length pair, this facility can also be used to reduce the processing necessary to set up DMAs for, and the latencies imposed by, existing scatter-gather I/O mechanisms that require an address/length pair for each physical page.

Cache coherent I/O can be used to achieve true copy-on-write functionality. Previously, even for copy-on-write data, the data had to be flushed from the data cache to physical memory, where the I/O device could access the data. This flush is essentially a copy. Cache coherent I/O, by allowing the I/O device to access data directly from the CPU data caches, eliminates processing time and latency imposed by this extra copy. The hardware can now support taking data straight from a user's data buffer to the I/O device.

To take advantage of the optimal page attributes where possible (e.g., IO_FAST for inbound DMA buffers) while ensuring correct behavior for devices that require suboptimal memory accesses such as I/O semaphore or locked (atomic) memory transactions, the mapping facility can be used to alias multiple I/O virtual addresses to the same physical addresses. Some software DMA programming models place DMA control information immediately adjacent to the DMA buffer. Frequently, this control information must be accessed by the I/O device using either read-modify-write or locking behavior. By mapping the same page twice, once as IO_SAFE and again as IO_FAST, and providing the I/O device with both I/O virtual addresses, the device can access memory using optimal IO_FAST DMA for bulk data transfer and IO_SAFE DMA for updating the control structures.

Finally, through careful programming, it is possible to take advantage of IO_FAST coherent I/O, and to allow the driver to maintain coherence for the occasional noncoherent update. For example, it is possible for the driver to flush a data structure explicitly from its cache, which will later be partially updated through a write-purge transaction from the adapter. This has the advantage of allowing the adapter to use its optimum form of DMA, while allowing the driver to determine when coherency must be explicitly maintained.

Performance Results

To collect performance results, the SPEC† SFS†† (LADDIS) 1.1 benchmark was run on the following configuration:

- 4-way HP 9000 Model K410
- 1G-byte RAM
- 4 FW-SCSI interfaces
- 56 file systems
- 4 FDDI networks
- HP-UX 10.01 operating system
- 132 NFS daemons
- 8 NFS clients
- 15 load-generating processes per client.

† SPEC stands for Systems Performance Evaluation Cooperative, an industry-standard benchmarking consortium.

†† The SPEC SFS benchmark measures a system's distributed file system (NFS) performance.

The noncoherent system achieved 4255 NFS operations per second with an average response time of 36.1 ms/operation. The coherent system achieved 4651 NFS operations per second with an average response time of 32.4 ms/operation.

The noncoherent system was limited by the response time. It's likely that with some fine-tuning the noncoherent system could achieve slightly more throughput.

To compare the machine behavior with and without coherent I/O, CPU and I/O adapter measurements were taken during several SFS runs in the configuration described above. The requested SFS load was 4000 NFS operations per second. This load level was chosen to load the system without hitting any known bottlenecks.

Comparing the number of instructions executed per NFS operation, the coherent system showed a 4% increase over the noncoherent system, increasing to 40,100 instructions

from 38,500. This increase was because of the overhead of the mapping calls. If we assume an average of 11 map/unmap pairs per NFS operation, then each pair costs about 145 instructions more than the alternative broadcast flush/purge data cache calls.

The degradation in path length was offset by a 17% improvement in CPI (cycles per instruction). CPI was measured at 2.01 on the coherent system and 2.42 on the noncoherent system.

The overall result was a 13% improvement in CPU instruction issue cycles per NFS operation. The coherent system used 80,700 CPU cycles per operation, while the noncoherent system needed 93,300 cycles.

To determine the efficiency of the software algorithms that manage the I/O TLB and to evaluate the sizing of the TLB, the number of I/O TLB misses was measured during these SFS runs. Under an SFS load of 4000 NFS operations per second, the disk drives missed 1.30 times per NFS operation, or 0.64% of all accesses.

Acknowledgments

The authors would like to acknowledge Monroe Bridges, Robert Brooks, Bill Bryg, and Jerry Huck for their contributions to the cache coherent I/O definition.

Bibliography

1. W.R. Bryg, J.C. Huck, R.B. Lee, T.C. Miller, and M.J. Mahon, "Hewlett-Packard Precision Architecture: The Processor," *Hewlett-Packard Journal*, Vol. 37, no. 8, August 1986, pp. 4-21.
2. J.L. Hennessy and D.A. Patterson, *Computer Architecture, a Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.
3. *HP-UX Driver Development Guide*, Preliminary Draft, HP 9000 Series 700 Computers, Hewlett-Packard Company, January 1995.

▶ Go to *Article 7*
▶ Go to Table of Contents
▶ Go to HPJournal Home Page