A Machine-Independent DMA Framework for NetBSD

Jason R. Thorpe[1]
Numerical Aerospace Simulation Facility
NASA Ames Research Center

Abstract

One of the challenges in implementing a portable kernel is finding good abstractions for semantically-similar operations which often have very machine-dependent implementations. This is especially important on modern machines which share common architectural features, e.g. the PCI bus.

This paper describes why a machine-independent DMA mapping abstraction is needed, the design considerations for such an abstraction, and the implementation of this abstraction in the NetBSD/alpha and NetBSD/i386 kernels.

1. Introduction

NetBSD is a portable, modern UNIX-like operating system which currently runs on eighteen platforms covering nine processor architectures. Some of these platforms, including the Alpha and i386[2], share the PCI bus as a common architectural feature. In order to share device drivers for PCI devices between different platforms, abstractions that hide the details of bus access must be invented. The details that must be hidden can be broken down into two classes: CPU access to devices on the bus (*bus_space*) and device access to host memory (*bus_dma*). Here we will discuss the latter; *bus_space* is a complicated topic in and of itself, and is beyond the scope of this paper.

Within the scope of DMA, there are two broad classes of details that must be hidden from the core device driver. The first class, host details, deals with issues such as the physical mapping of system memory (and the DMA mechanisms employed as a result of such mapping) and cache semantics. The second class, bus details, deals with issues related to features or limitations specific to the bus to which a device is attached, such as DMA bursting and address line limitations.

[1]Jason R. Thorpe is an employee of MRJ Technology Solutions, Inc. This work is funded by NASA contract NAS2-14303.

[2]The term "i386" is used here to refer to all of the 386-class and higher processors, including the i486, Pentium, Pentium Pro, and Pentium II.

1.1. Host platform details

In the example platforms listed above, there are at least three different mechanisms used to perform DMA. The first is used by the i386 platform. This mechanism can be described as "what you see is what you get": the address that the device uses to perform the DMA transfer is the same address that the host CPU uses to access the memory location in question.

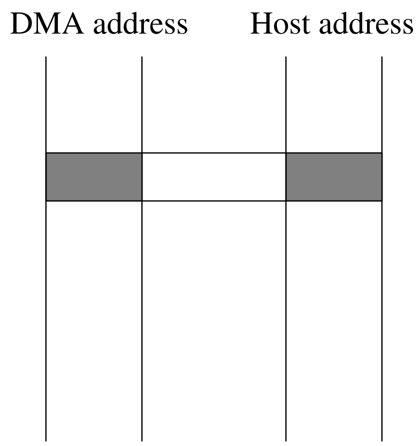

Figure 1 - WYSIWYG DMA

The second mechanism, employed by the Alpha, is very similar to the first; the address the host CPU uses to access the memory location in question is offset from some base address at which host memory is direct-mapped on the device bus for the purpose of DMA.

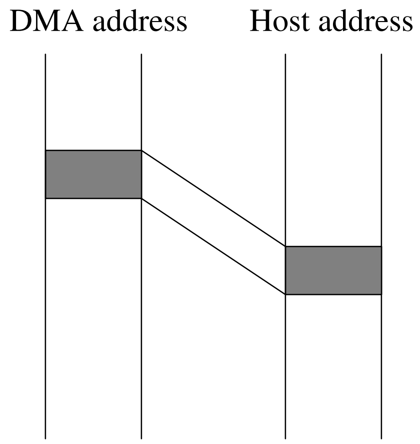

Figure 2 - direct-mapped DMA

The third mechanism, scatter-gather-mapped DMA, employs an MMU which performs translation of DMA addresses to host memory physical addresses. This mechanism is also used by the Alpha, because Alpha platforms implement a physical address space sometimes significantly larger than the 32-bit address space supported by most currently-available PCI devices.

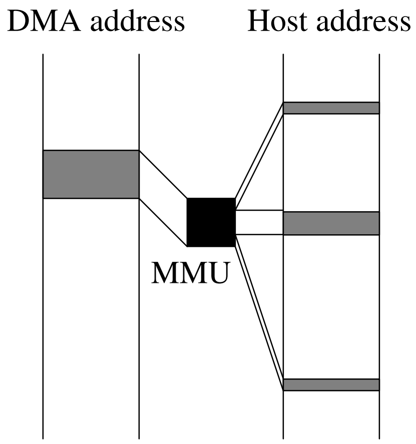

Figure 3 - scatter-gather-mapped DMA

The second and third DMA mechanisms above are combined on the Alpha through the use of *DMA windows*. The ASIC which implements the PCI bus on a particular platform has at least two of these DMA windows. Each window may be configured for direct-mapped or scatter-gather-mapped DMA. Windows are chosen based on the type of DMA transfer being performed, the bus type, and the physical address range of the host memory being accessed.

These concepts apply to platforms other than those listed above and busses other than PCI. Similar issues exist with the TurboChannel bus used on DECstations and early Alpha systems, and with the Q-bus used on some DEC MIPS and VAX-based servers.

The semantics of the host system's cache are also important to devices which wish to perform DMA. Some systems are capable of cache-coherent DMA. On such systems, the cache is often write-through (i.e. stores are written both to the cache and to host memory), or the cache has special snooping logic that can detect access to a memory location for which there is a dirty cache line (which causes the cache to be flushed automatically). Other systems are not capable of cache-coherent DMA. On these systems, software must explicitly flush any data caches before memory-to-device DMA transfers, as well as invalidate soon-to-be-stale cache lines before device-to-memory DMA.

1.2. Bus details

In addition to hiding the platform-specific DMA details for a single bus, it is desirable to share as much device driver code as possible for a device which may attach to multiple busses. A good example is the BusLogic family of SCSI adapters. This family of devices comes in ISA, EISA, VESA local bus, and PCI flavors. While there are some bus-specific details, such as probing and interrupt initialization, the vast majority of the code that drives this family of devices is identical for each flavor.

The BusLogic family of SCSI adapters are examples of what are termed *bus masters*. That is to say, the device itself performs all bus handshaking and host memory access during a DMA transfer. No third party is involved in the transfer. Such devices, when performing a DMA transfer, present the DMA address on the bus address lines, execute the bus's fetch or store operation, increment the address, and so forth until the transfer is complete. Because the device is using the bus address lines, the range of host physical addresses the device can access is limited by the number of such lines. On the PCI bus, which has at least 32 address lines, the device may be able to access the entire physical address space of a 32-bit architecture, such as the i386. ISA, however, only has 24 address lines. This means that the device can directly access only 16MB of physical address space.

A common solution to the limited-address-lines problem is a technique known as *DMA bouncing*. This technique involves a second memory area, located in the physical address range accessible by the device, known as a *bounce buffer*. In a memory-to-device transfer, the data is copied by the CPU to the bounce buffer, and the DMA operation is started. Conversely, in a device-to-memory transfer, the DMA operation is started, and the CPU then copies the data from the bounce buffer once the DMA operation has completed.

While simple to implement, DMA bouncing is not the most elegant way to solve the limited-address-line problem. On the Alpha, for example, scatter-gather-mapped DMA may be used to translate the out-of-range memory physical addresses to in-range DMA addresses that the device may use. This solution tends to offer better performance due to eliminated data copies, and is less expensive in terms of memory usage.

Returning to the BusLogic SCSI example, it is undesirable to place intimate knowledge of direct-mapping, scatter-gather-mapping, and DMA bouncing in the core device driver. Clearly, an abstraction that hides these details and presents a consistent interface, regardless of the DMA mechanism being used, is needed.

2. Design considerations

Hiding host and bus details is actually very straightforward. Handling WYSIWYG and direct-mapped DMA mechanisms is trivial. Handling scatter-gather-mapped DMA is also very easy, with the help of state kept in machine-dependent code layers. The presence and semantics of caches are also easy to handle with a set of four "synchronization" operations, and once caches are handled, DMA bouncing is conceptually trivial if viewed as a non-DMA-coherent cache. Unfortunately, while these operations are quite easy to do individually, traditional kernels do not provide a sufficiently abstract interface to the operations. This means that device drivers in these traditional kernels must handle each case explicitly.

In addition to the interface to these operations, a comprehensive DMA framework must also consider data buffer structures and DMA-safe memory handling.

2.1. Data buffer structures

The BSD kernel has essentially three different structures used to represent data buffers. The first is a simple linear buffer in virtual space, for example the data areas used to implement the file system buffer cache, and miscellaneous buffers allocated by the general purpose kernel memory allocator. The second is the *mbuf chain*. Mbufs are typically used by code which implements inter-process communication and networking. Their structure, small buffers chained together, reduces memory fragmentation and allows packet headers to be prepended easily. The third is the *uio* structure. This structure describes software scatter-gather to the kernel address space or to the address space of a specific process. It is most commonly used by the *read(2)* and *write(2)* system calls. While it would be possible for the device driver to treat the two more complex buffer structures as sets of multiple simple linear buffers, this is undesirable in terms of source code maintenance; the code to handle these data buffer structures can be complex, especially in terms of error handling.

In addition to the obvious need to DMA to and from memory mapped into kernel address space, it is common in modern operating systems to implement an optimized I/O interface for user processes which provides a method for devices to DMA directly to or from memory regions mapped into a process's address space. While this facility is partially provided for character device I/O by double-mapping the user buffer into kernel address space, the interface is not sufficiently general, and consumes kernel resources. This is somewhat related to the *uio* structure, in that the *uio* is capable of addressing buffers in a process's address space. However it may be desirable to use an alternate data format, such as a linear buffer, in some applications. In order to implement this, the DMA mapping framework must have access to processes' virtual memory structures.

It may also be desirable to DMA to or from buffers not mapped into any address space. The obvious example is frame grabbers. These devices, which capture video images, often require large, physically contiguous memory regions to store the captured image data. On some architectures, mapping of virtual address space is expensive. An application may wish to give a large buffer to the device, allow the device to continuously update the buffer, and then only map small regions of the buffer at any given time. Since the entire buffer need not be mapped into virtual address space, the DMA framework should provide an interface for using raw, unmapped buffers in DMA transfers.

2.2. DMA-safe memory handling

A comprehensive DMA framework must also provide several memory handling facilities. The most obvious of these is a method of allocating (and freeing) DMA-safe memory. The term "DMA-safe" is a way of describing a set of attributes the memory will have. First, DMA-safe memory must be addressable within the constraints of the bus. It must also be allocated in such a way as to not exceed the number of physical segments[3] specified by the caller.

In order for the kernel to access the DMA-safe memory, a method must exist to map this memory into kernel virtual address space. This is a fairly straightforward operation, with one exception. On some platforms which do not have cache-coherent DMA, cache flushes are very expensive. However, it is sometimes possible to mark virtual mappings of memory as cache-inhibited, or access physical memory though a cache-inhibited direct-mapped address segment. In order to accommodate these situations, a hint may be provided to the memory mapping function which specifies that the user of this memory wishes to avoid expensive data cache flushes.

To facilitate optimized I/O to process address spaces, it is necessary to provide processes a way of mapping a DMA-safe memory area. The most convenient way to do this is via a device driver's *mmap()* entry point. Thus, a DMA mapping framework must have a way to communicate with the VM system's *device pager*[4].

All of these requirements must be considered in the design of a complete DMA framework. When possible, the framework may merge semantically similar operations or concepts, but it must address all of these issues. The next section describes the interface provided by such a framework.

3. The *bus_dma* interface

What follows is a description of *bus_dma*, the DMA portion of the machine-independent bus access interface in NetBSD, commonly referred to as *bus.h*[5]. The DMA portion of the interface is comprised of three DMA-specific data types and thirteen function calls. The *bus_dma* interface also shares two data types with the *bus_space* interface.

The *bus_dma* functional interface is split into two categories: mapping calls and memory handling calls. The function calls themselves may be implemented as *cpp(1)* macros.

[3]This is somewhat misleading. The actual constraint is on the number of DMA segments the memory may map to. However, this usually corresponds directly to the number of physical memory segments which make up the allocated memory.

[4]The *device pager* provides support for memory mapping devices into a process's address space.

[5]The name is derived from the name of the include file that exports the interface.

3.1. Data types

The first of the two data types shared with the *bus_space* interface is the *bus_addr_t* type, which represents device bus addresses to be used for CPU access or DMA, and must be large enough to specify the largest possible bus address on the system. The second is the *bus_size_t* type, which represents sizes of bus address ranges.

The implementation of DMA on a given host/bus combination is described by the *bus_dma_tag_t*. This opaque type is passed to a bus's autoconfiguration machinery by machine-dependent code. The bus layer in turn passes it down to the device drivers. This tag is the first argument to every function in the interface.

Individual DMA segments are described by the *bus_dma_segment_t*. This type is a structure with two publicly accessible members. The first member, *ds_addr*, is a *bus_addr_t* containing the address of a DMA segment. The second, *ds_len*, is a *bus_size_t* containing the length of the segment.

The third, and probably most important, data type is the *bus_dmamap_t*. This type is a pointer to a structure which describes an individual DMA mapping. The structure has three public members. The first member, *dm_mapsize* is a *bus_size_t* describing the length of the mapping, when valid. A *dm_mapsize* of 0 indicates that the mapping is invalid. The second member, *dm_nsegs*, is an *int* which contains the number of DMA segments that comprise the mapping. The third public member, *dm_segs*, is an array or a pointer to an array of *bus_dma_segment_t* structures.

In addition to data types, the *bus_dma* interface also defines a set of flags which are passed to some of the interface's functions. Two of these flags, **BUS_DMA_WAITOK** and **BUS_DMA_NOWAIT**, indicate to the function that waiting for resources to become available is or is not allowed[6]. There are also four placeholder flags, **BUS_DMA_BUS1** through **BUS_DMA_BUS4**. These flags are reserved for the individual bus layers, which may need to define special semantics specific to that bus. An example of this is the ability of VESA local bus devices to utilize 32-bit DMA addresses; while the kernel considers such devices to be logically connected to the ISA bus, they are not limited to the addressing constraints of other ISA devices. The placeholder flags allow such special cases to be handled on a bus-by-bus basis.

3.2. Mapping functions

There are eight functions in the *bus_dma* interface that operate on DMA maps. These can be sub-categorized into functions that create and destroy maps, functions that load and unload mappings, and functions that synchronize maps.

The first two functions fall into the create/destroy sub-category. The *bus_dmamap_create()* function creates a DMA map and initializes it according to the parameters provided. The parameters include the maximum DMA transfer size the DMA map will map, the maximum number of DMA segments, the maximum size of any given segment, and any DMA boundary limitations. In addition to the standard flags, *bus_dmamap_create()* also takes the flag **BUS_DMA_ALLOCNOW**. This flag indicates that all resources necessary to map the maximum size transfer should be allocated when the map is created, and is useful in case the driver must load the DMA map at a time where blocking is not allowed, such as in interrupt context. The *bus_dmamap_destroy()* function destroys a DMA map, and frees any resources that may be assigned to it.

The next five functions fall into the load/unload sub-category. The two basic functions are *bus_dmamap_load()* and *bus_dmamap_unload()*. The former maps a DMA transfer to or from a linear buffer. This linear buffer may be mapped into either kernel or a process's virtual address space. The latter unloads the mappings previously loaded into the DMA map. If the **BUS_DMA_ALLOCNOW** flag was specified when the map was created, *bus_dmamap_load()* will not block or fail on resource allocation. Similarly, when the map is unloaded, the mapping resources will not be freed.

In addition to linear buffers handled by the basic *bus_dmamap_load()*, there are three alternate data buffer structures handled by the interface. The *bus_dmamap_load_mbuf()* function operates on mbuf chains. The individual data buffers are assumed to be in kernel virtual address space. The *bus_dmamap_load_uio()* function operates on *uio* structures, from which it extracts information about the address space in which the data resides. Finally, the *bus_dmamap_load_raw()* function operates on raw memory, which is not mapped into any virtual address space. All DMA maps loaded with these functions are unloaded with the *bus_dmamap_unload()* function.

Finally, the map synchronization sub-category includes one function: *bus_dmamap_sync()*. This function performs the four DMA synchronization operations necessary to handle caches and DMA bouncing. The four operations are:

BUS_DMASYNC_PREREAD
BUS_DMASYNC_POSTREAD
BUS_DMASYNC_PREWRITE
BUS_DMASYNC_POSTWRITE

The direction is expressed from the perspective of the host's memory. In other words, a device-to-memory transfer is a read, and a memory-to-device transfer is a write. The synchronization operations are expressed as flags, so it is possible to combine **READ** and **WRITE** operations in a single call. This is especially useful for synchronizing mappings of device control descriptors. Mixing of **PRE** and **POST** operations is not allowed.

In addition to the map and operation arguments, *bus_dmamap_sync()* also takes offset and length arguments. This is done in order to support partial syncs. In the case where a control descriptor is DMA'd to a device, it may be undesirable to synchronize the entire mapping, as doing so may be inefficient or even destructive to other control descriptors. Synchronizing the entire mapping is supported by passing an offset of 0 and the length specified by the map's *dm_mapsize*.

3.3. Memory handling functions

There are two sub-categories of functions that handle DMA-safe memory in the *bus_dma* interface: memory allocation and memory mapping.

The first function in the memory allocation sub-category, *bus_dmamem_alloc()*, allocates memory which has the specified attributes. The attributes that may be specified are: the size of the memory region to allocate, the alignment of each segment in the allocation, any boundary limitations, and the maximum number of DMA segments that may make up the allocation. The function fills in a provided array of *bus_dma_segment_t*s and indicates the number of valid segments in the array. Memory allocated by this interface is raw memory[7]; it is not mapped into any virtual address space. Once it is no longer in use, it may be freed with

[6]Waiting (also called "blocking") is allowed only if the kernel is running in a process context, as opposed to the interrupt context used when handling device interrupts.

the *bus_dmamem_free()* function.

In order for the kernel or a user process to access the memory, it must be mapped either into the kernel address space or the process's address space. These operations are performed by the memory mapping subcategory of DMA-safe memory handling functions. The *bus_dmamem_map()* function maps the specified DMA-safe raw memory into the kernel address space. The address of the mapping is returned by filling in a pointer passed by reference. Memory mapped in this manner may be unmapped by calling *bus_dmamem_unmap()*.

DMA-safe raw memory may be mapped into a process's address space via a device driver's *mmap()* entry point. In order to do this, the VM system's device pager repeatedly calls the driver, once for each page that is to be mapped. The driver translates the user-specified mmap offset into a DMA memory offset, and calls the *bus_dmamem_mmap()* function to translate the memory offset into an opaque value to be interpreted by the *pmap module*[8]. The device pager invokes the pmap module to translate the mmap cookie into a physical page address which is then mapped into the process's address space.

There are currently no methods for the virtual memory system to specify that an mmap'd area is being unmapped, or for the device driver to specify to the virtual memory system that a mmap'd region must be forcibly unmapped (for example, if a hot-swapable device has been removed from the system). This is widely regarded as a bug, and may be addressed in a future version of the NetBSD virtual memory system. If a change to this effect is made, the *bus_dma* interface will have to be adjusted accordingly.

4. Implementation of *bus_dma* in NetBSD/alpha and NetBSD/i386

This section is a description of the *bus_dma* implementation in two NetBSD ports, NetBSD/alpha and NetBSD/i386. It is presented as a side-by-side comparison in order to give the reader a better feel for the types of details that are being abstracted by the interface.

[7]This implies that *bus_dmamap_load_raw()* is an appropriate interface for mapping a DMA transfer to or from memory allocated by this interface.

[8]The pmap module is the machine-dependent layer of the NetBSD virtual memory system.

4.1. Platform requirements

NetBSD/alpha currently supports six implementations of the PCI bus, each of which implement DMA differently. In order to understand the design approach for NetBSD/alpha's fairly complex *bus_dma* implementation, it is necessary to understand the differences between the bus adapters. While some of these adapters have similar descriptions and features, the software interface to each one is quite different. (In addition to PCI, NetBSD/alpha also supports two TurboChannel DMA implementations on the DEC 3000 models. For simplicity's sake, we will limit the discussion to the PCI and related busses.)

The first PCI implementation to be supported by NetBSD/alpha was the DECchip 21071/21072 (APECS)[1]. It is designed to be used with the DECchip 21064 (EV4) and 21064A (EV45) processors. Systems in which this PCI host bus adapter is found include the AlphaStation 200, AlphaStation 400, and AlphaPC 64 systems, as well as some AlphaVME systems. The APECS supports up to two DMA windows, which may be configured for direct-mapped or scatter-gather-mapped operation, and uses host RAM for scatter-gather page tables.

The second PCI implementation to be supported by NetBSD/alpha was the built-in I/O controller found on the DECchip 21066[2] and DECchip 21068 family of Low Cost Alpha (LCA) processors. This processor family was used in the AXPpci33 and Multia AXP systems, as well as some AlphaVME systems. The LCA supports up to two DMA windows, which may be configured for direct-mapped or scatter-gather-mapped operation, and uses host RAM for scatter-gather page tables.

The third PCI implementation to be supported by NetBSD/alpha was the DECchip 21171 (ALCOR)[3], 21172 (ALCOR2), and 21174 (Pyxis)[9]. These PCI host bus adapters are found in systems based on the DECchip 21164 (EV5), 21164A (EV56), and 21164PC (PCA56) processors, including the AlphaStation 500, AlphaStation 600, and AlphaPC 164, and Digital Personal Workstation. The ALCOR, ALCOR2, and Pyxis support up to four DMA windows, which may be configured for direct-mapped or scatter-gather-mapped operation, and uses host RAM for scatter-gather page tables.

[9]While these chipsets are somewhat different from one another, the software interface is similar enough that they share a common device driver in the NetBSD/alpha kernel.

The fourth PCI implementation to be supported by NetBSD/alpha was the Digital DWLPA/DWLPB[4]. This is a TurboLaser-to-PCI[10] bridge found on AlphaServer 8200 and 8400 systems. The bridge is connected to the TurboLaser system bus via a KFTIA (internal) or KFTHA (external) I/O adapter. The former supports one built-in and one external DWLPx. The latter supports up to four external DWLPxs. Multiple I/O adapters may be present on the TurboLaser system bus. Each DWLPx supports up to four primary PCI busses and has three DMA windows which may be configured for direct-mapped or scatter-gather-mapped DMA. These three windows are shared by all PCI busses attached to the DWLPx. The DWLPx does not use host RAM for scatter-gather page tables. Instead, the DWLPx uses on-board SRAM, which must be shared by all PCI busses attached to the DWLPx. This is because the store-and-forward architecture of these systems would cause latency on DMA page table access to be too high. The DWLPA has 32K of page table SRAM and the DWLPB has 128K. Since the DWLPx can snoop access to the page table SRAM, no explicit scatter-gather TLB invalidation is necessary on this PCI implementation.

The fifth PCI implementation to be supported by NetBSD/alpha was the A12C PCI bus on the Avalon A12 Scalable Parallel Processor[5]. This PCI bus is a secondary I/O bus[11], has only a single PCI slot in mezzanine form-factor, and is used solely for Ethernet I/O. This PCI bus is not able to directly access host RAM. Instead, devices DMA to and from a 128K SRAM buffer. This is, in essence, a hardware implementation of DMA bouncing. This is not considered a limitation of the architecture given the target application of the A12 system (parallel computation applications which communicate via MPI[12] over the crossbar).

The sixth PCI implementation to be supported by NetBSD/alpha was the MCPCIA MCBUS-to-PCI bridge found on the AlphaServer 4100 (Rawhide) systems. The Rawhide architecture is made up of a "horse" (the central backplane) and two "saddles" (primary PCI bus adapters on either side of the backplane). The saddles may also contain EISA bus adapters. Each MCPCIA has four DMA windows which may be configured for direct-mapped or scatter-gather-mapped operation, and uses host RAM for scatter-gather page tables.

In sharp contrast to the Alpha, the i386 platform has a very simple PCI implementation; the PCI bus is capable of addressing the entire 32-bit physical address space of the PC architecture, and, in general, all PCI host bus adapters are software compatible. The i386 platform also has WYSIWYG DMA, so no window translations are necessary. The i386 platform, however, must contend with DMA bouncing on the ISA bus, due to ISA's 24-bit address limitation and lack of scatter-gather-mapped DMA.

4.2. Data structures

The DMA tags used by NetBSD/alpha and NetBSD/i386 are very similar. Both contain thirteen function pointers for the thirteen functional methods in the *bus_dma* interface. The NetBSD/alpha DMA tag, however, also has a function pointer used to obtain the DMA tag for children of the primary I/O bus and an opaque cookie to be interpreted by the low-level implementation of these methods.

The opaque cookie used by NetBSD/alpha's DMA tag is a pointer to the chipset's statically-allocated state information. This state information includes one or more *alpha_sgmap* structures. The *alpha_sgmap* contains all of the state information for a single DMA window to perform scatter-gather-mapped DMA, including pointers to the scatter-gather page table, the *extent map*[13] that manages the page table, and the DMA window base.

The DMA map structure contains all of the parameters used to create the map. (This is a fairly standard practice among all current implementations of the *bus_dma* interface.) In addition to the creation parameters, the two implementations contain additional state variables specific to their particular DMA quirks. For example, the NetBSD/alpha DMA map contains several state variables related to scatter-gather-mapped DMA. The i386 port's DMA map, on the other hand, contains a pointer to a map-specific cookie. This cookie holds state information for ISA DMA bouncing. This state is stored in a separate cookie because DMA

[10] "TurboLaser" is the name of the system bus on the AlphaServer 8200 and 8400 systems.

[11] The primary I/O bus on the A12 is a crossbar, which is used to communicate with other nodes in the parallel processor.

[12] MPI, or the Message Passing Interface, is a standardized API for passing data and control within a parallel program.

[13] An *extent map* is a data structure which manages an arbitrary number range, providing several resource allocation primitives. NetBSD has a general-purpose extent map manager which is used by several kernel subsystems.

bouncing is far less common on the i386 then scatter-gather-mapped DMA is on the Alpha, since the Alpha must also do scatter-gather-mapped DMA for PCI if the system has a large amount of physical memory.

In both the NetBSD/alpha and NetBSD/i386 *bus_dma* implementations, the DMA segment structure contains only the public members defined by the interface.

4.3. Code structure

Both the NetBSD/alpha and NetBSD/i386 *bus_dma* implementations use a simple inheritance scheme for code reuse. This is achieved by allowing the chipset- or bus-specific code layers (i.e. the "master" layers) to assemble the DMA tag. When the tag is assembled, the master layer inserts its own methods in the function pointer slots where special handling at that layer is required. For those methods which do not require special handling, the slots are initialized with pointers to common code.

The Alpha *bus_dma* code is broken down into four basic categories: chipset-specific code, code that implements common direct-mapped operations, code that implements common scatter-gather-mapped operations, and code that implements operations common to both direct-mapped and scatter-gather-mapped DMA. Some of the common functions are not called directly via the tag's function switch. These functions are helper functions, and are for use only by chipset front-ends. An example of such a helper is the set of common direct-mapped DMA load functions. These functions take all of the same arguments as the interface-defined methods, plus an extra argument: the DMA window's base DMA address.

The i386 *bus_dma* implementation, on the other hand, is broken down into three basic categories: common implementations of *bus_dma* methods, common helper functions, and ISA DMA front-ends[14]. All of the common interface methods may be called directly from the DMA tag's function switch. Both the PCI and EISA DMA tags utilize this feature; they provide no bus-specific DMA methods. The ISA DMA front-ends provide support for DMA bouncing if the system has more than 16MB of physical memory. If the system has 16MB of physical memory or less, no DMA bouncing is required, and the ISA DMA front-ends simply redirect the *bus_dma* function calls to the common implementation.

[14]ISA is currently the only bus supported by NetBSD/i386 with special DMA requirements. This may change in future versions of the system.

4.4. Autoconfiguration

The NetBSD kernel's autoconfiguration system employs a depth-first traversal of the nodes (devices) in the device tree. This process is started by machine-dependent code telling the machine-independent autoconfiguration framework that it has "found" the root "bus". In the two platforms described here, this root bus, called *mainbus*, is a virtual device; it does not directly correspond to any physical bus in the system. The device driver for *mainbus* is implemented in machine-dependent code. This driver's responsibility is to configure the primary I/O bus or busses.

In NetBSD/alpha, the chipset which implements the primary I/O bus is considered to be the primary I/O bus by the *mainbus* layer. Platform-specific code specifies the name of the chipset, and the *mainbus* driver configures it by "finding" it. When the chipset's device driver is attached, it initializes its DMA windows and data structures. Once this is complete, it "finds" the primary PCI bus or busses logically attached to the chipset, and passes the DMA tag for these busses down to the PCI bus device driver. This driver in turn finds and configures each device on the PCI bus, and so on.

In the event that the PCI bus driver encounters a PCI-to-PCI bridge (PPB), the DMA tag is passed unchanged to the PPB device driver, which in turn passes it to the secondary PCI bus instance attached to the other side of the bridge. However, intervention by machine-dependent code is required if the PCI bus driver encounters a bridge to a different bus type, such as EISA or ISA; this bus may require a different DMA tag. For this reason, all PCI-to-<other bus> bridge (PCxB) drivers are implemented in machine-dependent code. While the PCxB drivers could be implemented in machine-independent code using machine-dependent hooks to obtain DMA tags, this is not done as the secondary bus may require special machine-dependent interrupt setup and routing. Once all of the call-backs to handle the machine-dependent bus transition details were implemented, the amount of code that would be shared would hardly be worth the effort.

When a device driver is associated with a particular hardware device that the bus driver has found, it is given several pieces of information needed to initialize and communicate with the device. One of these pieces of information is the DMA tag. If the driver wishes to perform DMA, it must remember this tag, which, as noted previously, is used in every call to the *bus_dma* interface.

While the procedure for configuring busses and devices is essentially identical to the NetBSD/alpha case, NetBSD/i386 configures the primary I/O busses quite differently. The PC platform was designed from the ground up around the ISA bus. EISA and PCI are, in many ways, very similar to ISA from a device driver's perspective. All three have the concept of I/O-mapped[15] and memory-mapped space. The hardware and firmware in PCs typically map these busses in such a way that initialization of the bus's adapter by operating system software is not necessary. For this reason, it is possible to consider PCI, EISA, and ISA to all be primary I/O busses, from the autoconfiguration perspective.

The NetBSD/i386 *mainbus* driver configures the primary I/O busses in order of descending priority: PCI first, then EISA, and finally, ISA. The *mainbus* driver has direct access to each bus's DMA tags, and passes them down to the I/O bus directly. In the case of EISA and ISA, the *mainbus* layer only attempts to configure these busses if they were not found during the PCI bus configuration phase; NetBSD/i386, as a matter of correctness, identifies PCI-to-EISA (PCEB) and PCI-to-ISA (PCIB) bridges, and assigns autoconfiguration nodes in the device tree to them. The EISA and ISA busses are logically attached to these nodes, in a way very similar to that of NetBSD/alpha. The bridge drivers also have direct access to the bus's DMA tags, and pass them down to the I/O bus accordingly.

4.5. Example of underlying operation

This subsection describes the operation of the machine-dependent code which implements the *bus_dma* interface as used by a device driver for a hypothetical DES encryption card. While this is not the original application of *bus_dma*, it provides an example which is much easier to understand; the application for which the interface was developed is a high-performance hierarchical mass storage system, the details of which are overwhelming.

Not all of the details of a NetBSD device driver will be described here, but rather only those details which are important within the scope of DMA.

For the purpose of our example, the card comes in both PCI and ISA models. Since we describe two platforms, there are four permutations of actual examples. They will be tagged with the following indicators:

[15] I/O-mapped space is accessed with special instructions on Intel processors.

[Alpha/ISA]
[Alpha/PCI]
[i386/ISA]
[i386/PCI]

We will assume that the **[i386/ISA]** platform has more than 16MB of RAM, so transfers might have to be bounced if DMA-safe memory is not used explicitly. We will also assume that the direct-mapped DMA window on the **[Alpha/PCI]** platform is capable of addressing all of system RAM.

Please note that in the description of map synchronization, only cases which require special handing will be described. In both the **[Alpha/ISA]** and **[Alpha/PCI]** cases, all synchronizations cause the CPU's write buffer to be drained using the Alpha's *mb*[6] instruction. All synchronizations in the **[i386/PCI]** case are no-ops, as are synchronizations of DMA-safe memory in the **[i386/ISA]** case.

4.5.1. Hardware overview

The card is a bus master, and operates by reading a fixed-length command block via DMA. There are three commands: **SET KEY**, **ENCRYPT**, and **DECRYPT**. Commands are initiated by filling in the command block, and writing the DMA address of the command block to the card's *dmaAddr* register. The command block contains 6 32-bit words: *cbCommand*, *cbStatus*, *cbInAddr*, *cbInCount*, *cbOutAddr*, and *cbOutCount*. The *cbInAddr* and *cbOutAddr* members are the DMA addresses of software scatter-gather lists used by the card's DMA engine. The *cbInCount* and *cbOutCount* members are the number of scatter-gather entries in their respective lists. Each scatter-gather entry contains a DMA address and a length, both 32-bit words.

When the card processes a request, it reads the command block via DMA. It then examines the command block to determine which action to take. In the case of all three supported commands, it reads the input scatter-gather list at DMA address *cbInAddr* for length *cbInCount * 8*. It then switches the input to the appropriate processing engine. In the case of the **SET KEY** command, the scatter-gather list is used to DMA the DES key into SRAM located on the card. For all other commands, the input is directed at the pipelined DES engine, switched into either encrypt or decrypt mode. The DES engine then reads the output scatter-gather list specified by *cbOutAddr* for *cbOutCount * 8* bytes. Once the DES engine has all of the DMA addresses, it then begins the cycle of input-process-output until all data has been consumed. Once any command is finished, a status word is written to *cbStatus*, and an interrupt is delivered to the host. The driver software must read this word to determine if the command completed successfully.

4.5.2. Device driver overview

The device driver for this DES card provides *open()*, *close()*, and *ioctl()* entry points. The driver uses DMA to the user address space for high performance. When a user issues a request via the ioctl corresponding to the requested operation, the driver places it on a work queue. The *ioctl()* system call returns immediately, allowing the application to run or block via *sigsuspend()*. If the card is currently idle, the driver immediately issues the command to the card. When the job is finished, the card interrupts, and the driver notifies the user that the request has completed via the **SIGIO** signal. If there are more jobs on the work queue, the next job is removed from the queue and started, until there are no more jobs.

4.5.3. Driver initialization

When the driver instance is created (attached), it must create and initialize the data structures necessary for operation. This driver uses multiple DMA maps: one for the control structures (control block and scatter-gather lists), and many for data submitted by user requests. The data maps are kept in the driver job queue entries, which are created when jobs are submitted.

Next the driver must allocate DMA-safe memory for the control structures. The driver will allocate three pages of memory via *bus_dmamem_alloc*(). For simplicity, the driver will request a single memory segment. For all platforms and busses in this example, this operation simply calls a function in the virtual memory system that allocates memory with the requested constraints. In the **[i386/ISA]** case, the ISA layer inserts itself into the call graph to specify a range of 0 - 16MB. All other cases simply specify the entire present physical memory range.

A small piece of this memory will be used for the command block. The rest of the memory will be divided evenly between the two scatter-gather lists. This memory is then mapped into kernel virtual address space using *bus_dmamem_map()* with the **BUS_DMA_COHERENT** flag, and the kernel pointers to the three structures are initialized. When the memory is mapped on the i386, the **BUS_DMA_COHERENT** flag causes the cache-inhibit bits to be set in the PTEs. No special handing of this flag is required on the Alpha. However, in the Alpha case, since there is only a single segment, the memory is mapped via the Alpha's direct-mapped kernel segment; no use of kernel virtual address space is required.

Finally, the driver loads the control structure DMA map by passing the kernel virtual address of the memory to *bus_dmamap_load()*. To make it easier to start transactions, the driver caches the DMA addresses of the various control structures (by adding their offsets to the memory's DMA address). In all cases, the underlying load function steps through each page in the virtual address range, extracting the physical address from the pmap module and compacting the segments where possible. Since the memory was allocated as a single segment, it maps to a single DMA segment.

4.5.4. Example transaction

Let's suppose that the user has already set the key, and now wishes to use it to encrypt a data buffer. The calling program packages up the request, providing a pointer to the input buffer, output buffer, and status word, all in user space, and issues the "encrypt buffer" ioctl.

Upon entry into the kernel, the driver locks the user's buffer to prevent the data from being paged out while the DMA is in progress. A job queue entry is allocated, and two DMA maps are created for the job queue entry, one for the input buffer and one for the output buffer. In all cases, this allocates the standard DMA map structure. In the **[i386/ISA]** case, an ISA DMA cookie for each map is also allocated.

Once the queue entry has been allocated, it must be initialized. The first step in this process is to load the DMA maps for the input and output buffers. Since this process is essentially identical for input and output, only the actions for the input buffer's map are described here.

On **[Alpha/PCI]** and **[i386/PCI]**, the underlying code traverses the user's buffer, extracting the physical addresses for each page. For **[Alpha/PCI]**, the DMA window base is added to this address. The address and length of the segment are placed into the map's DMA segment list. Segments are concatenated when possible.

On **[Alpha/ISA]**, a very similar process occurs. However, rather than placing the physical addresses into the map's segment list, some scatter-gather-mapped DMA address space is allocated and the addresses placed into the corresponding page table entries. Once this process is complete, a single DMA segment is placed in the map's segment list, indicating the beginning of the scatter-gather-mapped area.

The **[i386/ISA]** case also traverses the user's buffer, but twice. In the first pass, the buffer is checked to ensure that it does not have any pages above the 16MB threshold. If it does not, then the procedure is identical to the **[i386/PCI]** case. However, for the sake of example, the buffer has pages outside the threshold so the transfer must be bounced. At this point, a bounce buffer is allocated. Since we are still in the process's context, this allocation may block. A pointer to the bounce buffer is stored in the ISA DMA cookie, and the physical address of the bounce buffer is placed in the map's segment list.

The next order of business is to enqueue or begin the transfer. To keep the example simple, we will assume that no other transfers are pending. The first step in this process is to initialize the control block with the cached DMA addresses of the card's scatter-gather lists. These lists are also initialized with the contents of the DMA maps' segment list. Before we tell the card to begin transferring data, we must synchronize the DMA maps.

The first map to be synchronized is the input buffer map. This is a **PREWRITE** operation. In the **[i386/ISA]** case, the user's buffer is copied from the user's address space into the bounce buffer[16]. The next map to be synchronized is the output buffer map. This is a **PREREAD** operation. Finally, the control map is synchronized. Since the status will be read back from the control block after the transaction is complete, this synchronization is a **PREREAD|PREWRITE**.

At this point the DMA transaction may occur. The card is started by writing the cached DMA address of the control block into the card's *dmaAddr* register. The driver returns to user space, and the process waits for the signal indicating that the transaction has completed.

Once the transaction has completed, the card interrupts the host. The interrupt handler is now responsible for finishing the DMA sequence and notifying the requesting process that the operation is complete.

[16]This is not currently implemented, as it required substantial changes to the virtual memory system. This is because the *copyin()* and *copyout()* functions only operate on the current process's context, which may not be available at the time of the bounce. Those changes to the virtual memory system have now been made, so support for bouncing to and from user space will appear in a future release of NetBSD. Support for bouncing from kernel space is currently supported, however.

The first task to perform is to synchronize the input buffer map. This is a **POSTWRITE**. Next we synchronize the output buffer map. This is a **POSTREAD**. In the **[i386/ISA]** case, the contents of the output bounce buffer are copied to the user's buffer[17]. Finally, we synchronize the control map. This is a **POSTREAD|POSTWRITE**.

Now that the DMA maps have been synchronized, they must be unloaded. In the **[Alpha/PCI]** and **[i386/PCI]** cases, there are no resources to be freed; the mapping is simply marked invalid. In the **[Alpha/ISA]** case, the scatter-gather-mapped DMA resources are released. In the **[i386/ISA]** case, the bounce buffer is freed.

Since the user's buffer is no longer in use, it is unlocked by the device driver. Now the process may be signaled that I/O has completed. The last task to perform is to destroy the input and output buffer DMA maps and the job queue entry.

5. Conclusions

The *bus_dma* interface was introduced into the NetBSD kernel at development version 1.2G, just before the release cycle for NetBSD 1.3 began. When the code was committed to the NetBSD master sources, several drivers, mostly for SCSI controllers, were converted to the interface at the same time. (All of these drivers had been previously converted to use the *bus_space* interface.) Not only did these drivers provide an example of the correct use of *bus_dma*, but they provided functionality that had not previously existed in the NetBSD kernel: support for bus mastering ISA devices in PCs with more than 16MB of RAM.

The first real test of the interface on the Alpha platform came by installing a bus mastering ISA device (an Adaptec 1542 SCSI controller) in an AXPpci33 computer. After addressing a small bug in the Alpha implementation of *bus_dmamap_load*(), the device worked flawlessly.

When converting device drivers to use the new interface, developers discovered that a fair amount of mostly-similar code could be removed from each driver converted. The code in question was the loop that built the software scatter-gather list. In some cases, the drivers performed noticeably better, due to the fact that the implementation of this loop within *bus_dmamap_load()* is more efficient and supports

[17]The same caveat applies here as to the **[i386/ISA]** **PREWRITE** case for the input map.

segment concatenation.

Most of the machine-independent drivers that use DMA have been converted to the new interface, and more platforms have implemented the necessary back-ends. The results have been very encouraging. Nearly every device/platform combination that has been tested has worked without additional modifications to the device driver. The few exceptions to this have generally been to handle differences in host and device byte-order, and are not directly related to DMA.

The *bus_dma* interface has also paved the way for additional machine-independent bus autoconfiguration frameworks, such as for VME. Eventually, this will help support PCI-to-VME bridges, and allow Sun, Motorola, and Intel systems to share common VME device drivers.

We have found the *bus_dma* interface to be a major architectural benefit in the NetBSD kernel, greatly simplifying the process of porting the kernel to new platforms, and making portable device driver development considerably easier. In short, the abstraction has delivered what it was designed to deliver: a means of supporting a wide range of platforms with maximum code reuse.

6. References

[1] Digital Equipment Corporation, *DECchip 21071 and DECchip 21072 Core Logic Chipsets Data Sheet,* DEC order number EC-QAEMA-TE, November 1994.

[2] Digital Equipment Corporation, *DECchip 21066 Alpha AXP Microprocessor Data Sheet,* DEC order number EC-N0617-72, May 1994.

[3] Digital Equipment Corporation, *DECchip 21171 Core Logic Chipset Technical Reference Manual,* DEC order number EC-QE18B-TE, September 1995.

[4] Digital Equipment Corporation, *DWLPA and DWLPB PCI Adapter Technical Manual,* DEC order number EK-DWLPX-TM, July 1996.

[5] H. Ross Harvey, *Avalon A12 Parallel Supercomputer Theory of Operation,* Avalon Computer Systems, Inc., October 1997.

[6] Richard L. Sites and Richard T. Witek, *Alpha AXP Architecture Reference Manual, Second Edition,* Digital Press, 1995.

7. Obtaining NetBSD

More information about NetBSD, including information on where to obtain sources and binaries for the NetBSD operating system itself, may be found at **http://www.netbsd.org/**.

Updates to this paper may appear periodically, and can be found at **http://www.netbsd.org/Documentation/research/**.

8. Acknowledgments

I would like to thank the following individuals for their very constructive input and insight during the *bus_dma* design phase: Chris Demetriou, Charles Hannum, Ross Harvey, Matthew Jacob, Jonathan Stone, and Matt Thomas.

I would also like to extend special thanks to Chris Demetriou, Lonhyn Jasinskyj, Kevin Lahey, Yvonne Malloy, David McNab, and Harry Waddell for the time they spent reviewing and helping me to polish this paper.

9. About the author

Jason R. Thorpe is a Network Systems Engineer at the Numerical Aerospace Simulation Facility at NASA's Ames Research Center. His professional interests include design and implementation of portable operating systems, high-speed computer networks, and network protocols. In addition to his work on the NetBSD operating system in support of network and mass storage system development projects at the NAS facility, he is an active participant in the Internet Engineering Task Force. He has been a contributor to the NetBSD Project since mid-1993, and has run nearly every port at one time or another. He currently maintains NetBSD's hp300 port, and is a member of the NetBSD Core Group.

The author may be reached at: Numerical Aerospace Simulation Facility, Mail Stop 258-5, NASA Ames Research Center, Moffett Field, CA 94035, or via electronic mail at *thorpej@nas.nasa.gov*.