Protected, User-level DMA for the SHRIMP Network Interface

Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, and Kai Li
Department of Computer Science, Princeton University, Princeton, NJ 08544

Abstract

Traditional DMA requires the operating system to perform many tasks to initiate a transfer, with overhead on the order of hundreds or thousands of CPU instructions. This paper describes a mechanism, called User-level Direct Memory Access (UDMA), for initiating DMA transfers of input/output data, with full protection, at a cost of only two user-level memory references. The UDMA mechanism uses existing virtual memory translation hardware to perform permission checking and address translation without kernel involvement. The implementation of the UDMA mechanism is simple, requiring a small extension to the traditional DMA controller and minimal operating system kernel support. The mechanism can be used with a wide variety of I/O devices including network interfaces, data storage devices such as disks and tape drives, and memory-mapped devices such as graphics frame-buffers. As an illustration, we describe how we used UDMA in building network interface hardware for the SHRIMP multicomputer.

1 Introduction

Direct Memory Access (DMA) is a common method for routing data directly between memory and an input/output device without requiring intervention by the CPU to control each datum transferred. Normally, a DMA transaction can be initiated only through the operating system kernel, which provides protection, memory buffer management, and related address translation. The overhead of this kernel-initiated DMA transaction is hundreds, possibly thousands of CPU instructions [2].

The high overhead of traditional DMA devices requires coarse grained transfers of large data blocks in order to achieve the available raw DMA channel bandwidths. This is particularly true for high-bandwidth devices such as network interfaces and HIPPI [1] devices. For example, the overhead of sending a piece of data over a 100 MByte/sec HIPPI channel on the Paragon multicomputer is more than 350 microseconds [13]. With a data block size of 1 Kbyte, the transfer rate achieved is only 2.7 MByte/sec, which is less than 2% of the raw hardware bandwidth. Achieving a transfer rate of 80 MBytes/sec requires the data block size to be larger than 64 KBytes. The overhead is the dominating factor which limits the utilization of DMA devices for fine grained data transfers.

This paper describes a protected, user-level DMA mechanism (UDMA) developed at Princeton University as part of the SHRIMP project [4]. The UDMA mechanism uses virtual memory mapping to allow user processes to start DMA operations via a pair of ordinary load and store instructions. UDMA uses the existing virtual memory mechanisms – address translation and permission checking – to provide the same degree of protection as the traditional DMA operations. A UDMA transfer

- can be started with two user-level memory references,
- does not require a system call,
- does not require DMA memory pages to be "pinned," and
- can move data between an I/O device and any location in memory.

A UDMA device can be used concurrently by an arbitrary number of untrusting processes without compromising protection. Finally, UDMA puts no constraints on the scheduling of the processes that use it.

In contrast, a traditional DMA transfer costs hundreds, possibly thousands of instructions, including a system call and the cost of pinning and unpinning the affected pages (or copying pages into special prepinned I/O buffers). In this paper we show that the UDMA mechanism is simple and requires little operating system kernel support.

The UDMA technique is used in the design of the network interface hardware for the SHRIMP multicomputer. The use of UMDA in SHRIMP allows very fast, flexible communication, which is managed at user level.

In Proceedings of the 2nd International Symposium on High-Performance Computer Architecture, February, 1996.

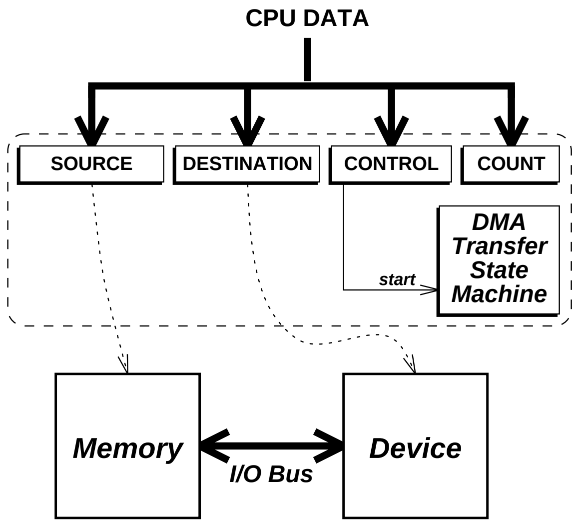

Figure 1: *Traditional DMA hardware configured for a memory to device transfer. Transfer in both directions is possible.*

Although we have utilized the UDMA technique in the design of a network interface, we reiterate that it is applicable to a wide variety of high-speed I/O devices including graphics frame-buffers, audio and video devices, and disks.

2 Traditional DMA

Direct Memory Access was first implemented on the IBM SAGE computer in 1955 [8] and has always been a common approach in I/O interface controller designs for data transfer between main memory and I/O devices.

Figure 1 shows a typical DMA controller which is configured to perform DMA from memory to a device over an I/O bus. The device typically consists of a single port to or from which the data is transferred. The DMA mechanism typically consists of a state machine and several registers including source and destination address registers, a counter, and a control register. To transfer data from memory to the I/O device, the CPU puts the physical memory address into the source register, the device address into the destination register, sets the counter to the number of bytes to be transferred, and triggers the control register to start transferring the first datum. After the first datum is transferred, the state machine increments the source register, decrements the counter, and starts transferring the second datum. These transfers continue until the counter reaches zero. Note that the entire data transfer process requires no CPU intervention.

Although this mechanism is simple from a hardware point of view, it imposes several high-cost requirements on the operating system kernel. First, it requires the use of a system call to initiate the DMA operation, primarily to verify the user's permission to access the device, and to ensure mutual exclusion among users sharing the device. User processes must pay the overhead of a system call to initiate a DMA operation.

A second requirement is virtual-to-physical memory address translation. Because the DMA controller uses physical memory addresses, the virtual memory addresses of the user programs must be translated to physical addresses before being loaded into the address registers. Virtual-to-physical address translation is performed by the operating system kernel.

Finally, the physical memory pages used for DMA data transfers must be pinned to prevent the virtual memory system from paging them out while DMA data transfers are in progress. Since the cost of pinning memory pages is high, most of today's systems reserve a certain number of pinned physical memory pages for each DMA device as I/O buffers. This method may require copying data between memory in user address space and the reserved, pinned DMA memory buffers.

Altogether, a typical DMA transfer requires the following steps:

1. A user process makes a system call, asking the kernel to do an I/O operation. The user process names a region in its virtual memory to serve as the source or destination of the transfer.

2. The kernel translates the virtual addresses to physical addresses, verifies the user process's permission to perform the requested transfer, pins the physical pages into memory, builds a DMA descriptor specifying the pages to transfer, and then starts the DMA device.

3. The DMA device performs the requested data transfer, and then notifies the kernel by changing a status register or causing an interrupt.

4. The kernel detects that the transfer has finished, unpins the physical pages, and reschedules the user level process.

Starting a DMA transaction usually takes hundreds or thousands of CPU instructions. Therefore, DMA is beneficial only for infrequent operations which transfer a large amount of data, restricting its usefulness.

3 The UDMA Approach

In order to reduce the cost of DMA, we must remove the kernel tasks from the critical data transfer path. The challenge is to find inexpensive methods for permission checking, address translation, and prevention of DMA page remapping. Our solution is a mechanism that allows programs to use two ordinary user-level memory instructions to initiate a DMA transaction, yet provides the same degree of protection as the traditional method. We call such a DMA mechanism *User-Level DMA* (or UDMA).

A user program initiates a UDMA transfer by issuing two ordinary user-level memory references:

```
STORE nbytes TO destAddr
LOAD status FROM srcAddr
```

The `STORE` instruction specifies the destination base address of the DMA transaction, `destAddr`, and the number of bytes to transfer, `nbytes`. The `LOAD` instruction specifies the source base address of the DMA transfer, `srcAddr`, and initiates the transfer if the DMA engine is not busy. The `LOAD` returns a status code, `status`, to indicate whether the initiation was successful or not.

It is imperative that the order of the two memory references be maintained, with the `STORE` proceeding the `LOAD`. Although many current processors optimize memory bus usage by reordering references, all provide some mechanism that software can use to ensure program order execution for memory-mapped I/O.

Central to the UDMA mechanism is the use of virtual memory mapping. The UDMA mechanism uses the Memory Management Unit (MMU) of the CPU to do permission checking and virtual-to-physical address translation. In order to utilize these features, a combination of hardware and operating system extensions is required.

The hardware support can be viewed as simple extensions to the traditional DMA hardware shown in Figure 1. The extensions include a state machine to interpret the two instruction (`STORE`, `LOAD`) initiation sequence, and simple physical address translation hardware.

The operating system extensions include some changes in the virtual memory software to create and manage proxy mappings, and a single tiny change in the context-switch code. The close cooperation between hardware and system software allows us to completely eliminate the kernel's role in initiating UDMA transfers.

The following three sections explain the virtual memory mapping, hardware support and operating system support in detail.

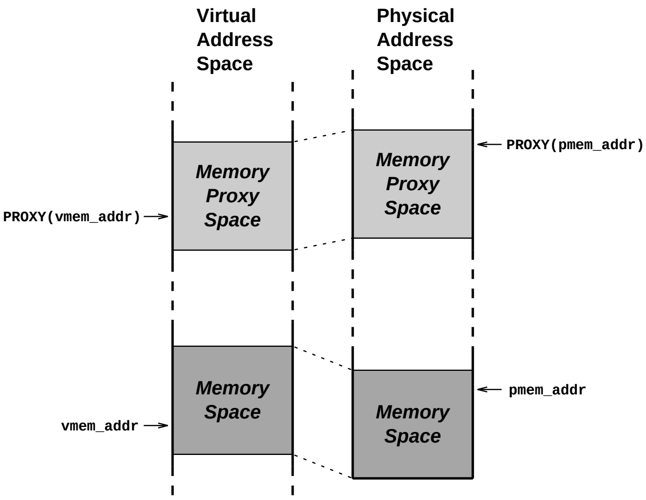

Figure 2: *Memory and memory proxy space mappings.*

4 Memory Mapping for UDMA

UDMA takes advantage of the existing virtual memory translation hardware in the CPU to perform protection checking and address translation. The key element of the virtual memory mapping for the UDMA mechanism is a concept called *proxy space*.

Proxy Space

A proxy space is a region of address space reserved for user-level communication with a UDMA device. There are proxy spaces in both the virtual and physical address spaces, and they are related as shown in Figure 2. This figure shows the system memory space and its associated memory proxy space in both the physical and virtual address spaces. A proxy space is uncachable and it is not backed by any real physical memory, so it cannot store data.

The key concept is that *there is a one-to-one association between memory proxy addresses and real memory addresses*. This association can be represented by an address translation function, `PROXY`:

```
proxy address = PROXY(real address)
```

`PROXY` applied to a virtual memory address, `vmem_addr`, returns the associated virtual memory proxy address, `vmem_prox`. Likewise, `PROXY` applied to a physical memory address, `pmem_addr`, returns the associated physical memory proxy address, `pmem_prox`.

Virtual memory proxy pages are mapped to physical memory proxy pages just as standard virtual memory pages are mapped to standard physical memory pages. Mapping a virtual proxy page to a physical proxy page is equivalent to granting the owner of the virtual page restricted permission to access the UDMA mechanism. For example, an address in the memory proxy space can be referenced by a user-level process to specify a source or destination for DMA transfers in the memory space.

In addition to the memory proxy space described so far, there is the similar concept of *device proxy space*, which is used to refer to regions inside the I/O device. Unlike memory proxy space, which is associated with real memory, there is no "real" device space associated with device proxy space. Instead, there is a fixed, one-to-one correspondence between addresses in device proxy space and possible DMA sources and destinations within the device. To name an address in the device as a source or destination for DMA, the user process uses the unique address in device proxy space corresponding to the desired address in the device.

The precise interpretation of "addresses in device proxy space" is device specific. For example, if the device is a graphics frame-buffer, a device address might specify a pixel. If the device is a network interface, a device address might name a destination network address. If the device is a disk, a device address might name a block. Furthermore, unlike traditional DMA, the UDMA mechanism can increment the device address along with the memory address as the transfer progresses.

Proxy Mapping for UDMA

Like ordinary memory, proxy space exists in both virtual and physical manifestations. User processes deal with virtual addresses, and the hardware deals with physical addresses. The operating system kernel sets up the associated virtual memory page table entries to create the protection and mapping from virtual proxy addresses to physical proxy addresses. The ordinary virtual memory translation hardware (the MMU) performs the actual translation and protection checking. Translated physical proxy space addresses are recognized by the UDMA hardware.

Figure 3 shows a typical memory configuration for a system which supports one device accepting UDMA transfers. The physical address space contains three regions: real memory space, memory proxy space, and device proxy space. Accesses to each region can be recognized by pattern-matching some number of high-order address bits, depending on the size and location of the regions.

Each of the three regions in the physical space has a corresponding region in the virtual space which can be mapped to it. Mapping a virtual memory proxy page enables the owner of the page to perform UDMA transfers to or from the associated memory page *only*. Therefore, protection of physical memory proxy pages (and their associated physical memory pages) is provided by the existing virtual memory system. A process must obtain a proxy memory page mapping for every real memory page it uses as a source or destination for UDMA transfers.

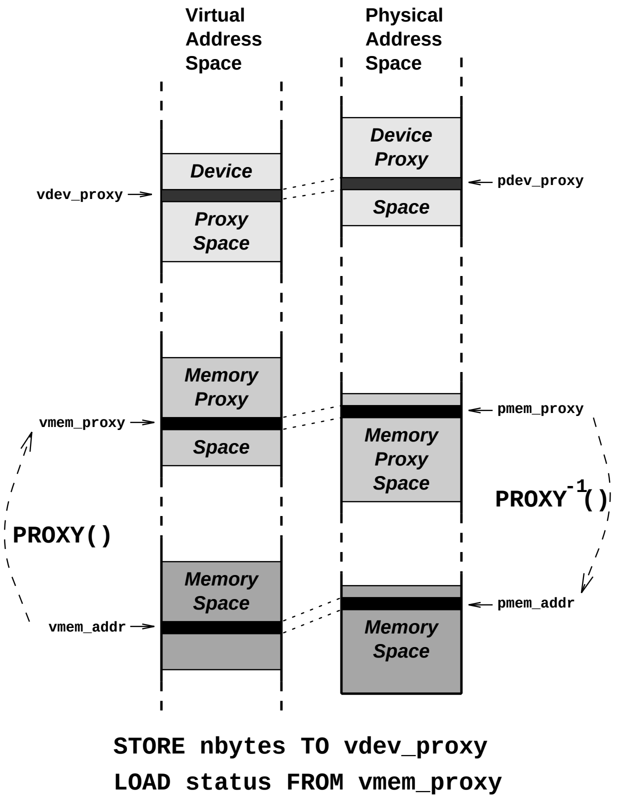

Figure 3: *Address translation for a memory-to-device transfer.*

Likewise, mapping a virtual device proxy page enables the owner of the page to perform some sort of device-specific UDMA transfer to or from the device. Again, the virtual memory system can be used to protect portions of the device, depending on the meaning of the device proxy addresses.

Figure 3 shows a transfer from the physical memory base address `pmem_addr` to the physical device base address `pdev_proxy`. The process performing the transfer must have mappings for `vmem_addr`, `vmem_proxy`, and `vdev_proxy`. After computing `vmem_proxy = PROXY(vmem_addr)`, the process issues the two instructions to initiate the transfer. The UDMA hardware computes `pmem_addr =` $\texttt{PROXY}^{-1}$`(pmem_proxy)` and initiates a DMA transfer of nbytes from the base address pmem_addr to the device. Note that the translation from pdev_proxy to a device address is device-specific.

Because virtual memory protection is provided on a per-page basis, a basic UDMA transfer cannot cross a page boundary in either the source or destination spaces. We extend the basic scheme in Section 7 to include multi-page transfers.

The device proxy mapping from virtual memory address space to physical address space is straightforward. An operating system call is responsible for creating the mapping. The system call decides whether to grant permission to a user process's request and whether the permission is read-only. The system call will set appropriate mapping in the virtual memory translation page table entries and return appropriate status to the user process.

The memory proxy mapping is similarly created, but the virtual memory system must maintain the mapping based on the virtual-to-physical memory mapping of its corresponding real physical memory. The virtual memory system guarantees that a virtual-to-physical memory proxy space mapping is valid only if the virtual-to-physical mapping of its corresponding real memory is valid. This invariant is maintained during virtual memory page swapping, and is described in detail in Section 6.

5 UDMA Hardware

The purpose of the UDMA hardware is to provide the minimum necessary support for the UDMA mechanism while reusing existing DMA technology. The UDMA hardware extends standard DMA hardware to provide translation from physical proxy addresses to real addresses, to interpret the transfer initiation instruction sequence, and to guarantee atomicity for context switches in operating systems. Figure 4 shows how the additional hardware is situated between the standard DMA engine and the CPU, and should be compared to Figure 1. Notice that the UDMA hardware utilizes both the CPU address and data in order to communicate very efficiently.

Address translation from physical proxy addresses to real addresses consists of applying the function $\mathtt{PROXY}^{-1}$ to the CPU address and loading that value into either the source or destinations address register of the standard DMA engine. For simplicity of address translation, the real memory space and the proxy memory space can be laid out at the same offset in each half of the physical address space. Then the PROXY and $\mathtt{PROXY}^{-1}$ functions amount to nothing more than

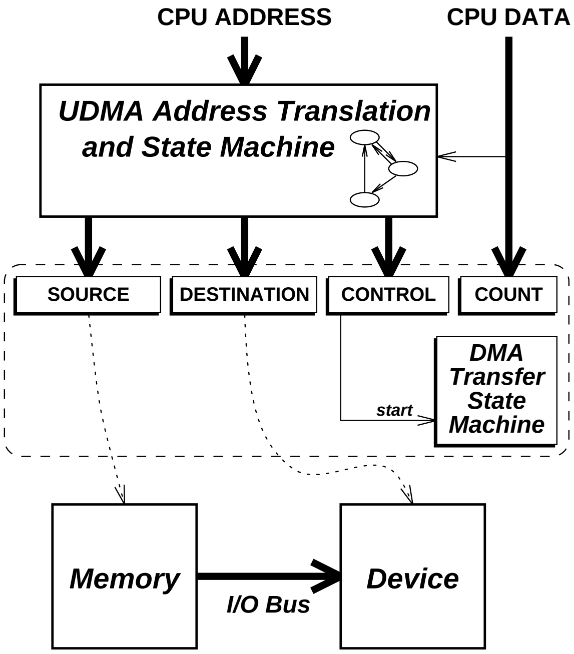

Figure 4: *UDMA hardware configured for a memory to device transfer.*

flipping the high order address bit. A somewhat more general scheme is to lay out the memory proxy space at some fixed offset from the real memory space, and add or subtract that offset for translation.

UDMA State Machine

The STORE, LOAD transfer initiation instruction sequence is interpreted by a simple state machine within the UDMA hardware as shown in Figure 5. (In the diagram, if no transition is depicted for a given event in a given state, then that event does not cause a state transition.)

The state machine manages the interaction between proxy-space accesses and the standard DMA engine. The machine has three states: *Idle*, *Dest-Loaded*, and *Transferring*. It recognizes three transition events: Store, Load, and Inval. Store events represent STOREs of positive values to proxy space. Load events represent LOADs from proxy space. Inval events represent STOREs of negative values (passing a negative, and hence invalid, value of nbytes to proxy space).

To understand the state transitions, consider first the most common cases. When idle, the state machine is in the *Idle* state. It stays there until a STORE to proxy space is performed, causing a Store event. When this occurs, the referenced proxy address is translated to a

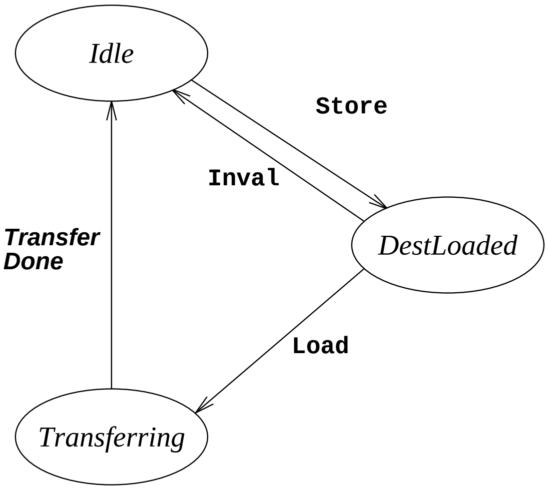

Figure 5: *State transitions in the UDMA hardware state machine.*

real address and put in the `SOURCE` register, the value stored by the CPU is put in the `COUNT` register, and the hardware enters the *DestLoaded* state.

The next relevant event is a `LOAD` from proxy space, causing a `Load` event. When this occurs, the referenced proxy address is translated to a real address and put into the `DESTINATION` register, and the hardware enters the *Transferring* state. This causes the UDMA state machine to write a value to the control register to start the standard DMA transfer.

When the transfer finishes, the UDMA state machine moves from the *Transferring* state back into the *Idle* state, allowing user processes to initiate further transfers. Although this design does not include a mechanism for software to terminate a transfer and force a transition from the *Transferring* state to the *Idle* state, it is not hard to imagine adding one. This could be useful for dealing with memory system errors that the DMA hardware cannot handle transparently.

Several other, less common transitions are also possible. In the *DestLoaded* state, a `Store` event does not change the state, but overwrites the `DESTINATION` and `COUNT` registers. An `Inval` event moves the machine into the *Idle* state and is used to terminate an incomplete transfer initiation sequence.

A fourth transition event not shown on the state transition diagram is the `BadLoad` event, which causes a transition from the *DestLoaded* to the *Idle* state. This represents a load from a proxy address in the same proxy region (memory or device) as the value in the `DESTINATION` register, and corresponds to a user process asking for a memory-to-memory or device-to-device transfer, which the (basic) UDMA device does not support.

Status Returned by Proxy LOADs

A `LOAD` instruction can be performed at any time to any proxy address in order to check the status of the UDMA engine. The `LOAD` will only initiate a transfer under the conditions described above. Every `LOAD` returns the following information to the user process:

- INITIATION FLAG (1 bit): zero if the access causes a transition from the *DestLoaded* state to the *Transferring* state (i.e. if the access started a DMA transfer); one otherwise.

- TRANSFERRING FLAG (1 bit): one if the device is in the *Transferring* state; zero otherwise.

- INVALID FLAG (1 bit): one if the device is in the *Idle* state; zero otherwise.

- MATCH FLAG (1 bit): one if the machine is in the *Transferring* state and the address referenced is equal to the base (starting) address of the transfer in progress; zero otherwise.

- WRONG-SPACE FLAG (1 bit): one if the access is a *BadLoad* as defined above; zero otherwise.

- REMAINING-BYTES (variable size, based on page size): the number of bytes remaining to transfer if in the *DestLoaded* or *Transferring* state; zero otherwise.

- DEVICE-SPECIFIC ERRORS (variable size): used to report error conditions specific to the I/O device. For example, if the device requires accesses to be aligned on 4-byte boundaries, an error bit would be set if the requested transfer was not properly aligned.

The `LOAD` instruction that attempts to start a transfer will return a zero initiation flag value if the transfer was successfully initiated. If not, the user process can check the individual bits of the return value to figure out what went wrong. If the transferring flag or the invalid flag is set, the user process may want to re-try its two-instruction transfer initiation sequence. If other error bits are set, a real error has occurred.

To check for completion of a successfully initiated transfer, the user process should repeat the `LOAD` instruction that it used to start the transfer. If this `LOAD` instruction returns with the match flag set, then the transfer has not completed; otherwise it has.

6 Operating System Support

The UDMA mechanism requires support from the operating system kernel to guarantee the atomicity of DMA transfer initiations, to create virtual memory mapping for UDMA, and to maintain memory proxy mapping during virtual memory paging. The operating system maintains four invariants:

I1 If a `LOAD` instruction initiates a UDMA transfer, then the destination address and byte count must have been `STORE`d by the same process.

I2 If there is a mapping from `PROXY(vmem_addr)` to `PROXY(pmem_addr)`, then there must be a virtual memory mapping from `vmem_addr` to `pmem_addr`.

I3 If `PROXY(vmem_addr)` is writable, then `vmem_addr` must be dirty.

I4 If `pmem_addr` is in the hardware `SOURCE` or `DESTINATION` register, then `pmem_addr` will not be remapped.

These invariants are explained in detail in the following subsections.

Maintaining *I1*: Atomicity

The operating system must guarantee invariant *I1* to support atomicity of the two-instruction transfer initiation sequence. Because the UDMA mechanism requires a program to use two user-level references to initiate a transfer, and because multiple processes may share a UDMA device, there exists a danger of incorrect initiation if a context switch takes place between the two references.

To avoid this danger, the operating system must invalidate any partially initiated UDMA transfer on every context switch. This can be done by causing a hardware `Inval` event (i.e. by storing a negative `nbytes` value to any valid proxy address), causing the UDMA hardware state machine to return to the `Idle` state. The context-switch code does this with a single `STORE` instruction.

When the interrupted user process resumes, it will execute the `LOAD` instruction of its transfer-initiation sequence, which will return a failure code signifying that the hardware is in the *Idle* state or *Transferring* for another process. The user process can deduce what happened and re-try its operation.

Note that the UDMA device is stateless with respect to a context switch. Once started, a UDMA transfer continues regardless of whether the process that started it is de-scheduled. The UDMA device does not know which user process is running, or which user process started any particular transfer.

Maintaining *I2*: Mapping Consistency

The virtual memory manager in the operating system must cooperate with the UDMA device to create virtual memory mappings for memory proxy and device proxy spaces, and must guarantee invariant *I2* to ensure that a virtual-to-physical memory proxy space mapping is valid only if the virtual-to-physical mapping of its corresponding real memory is valid.

In order for a process to perform DMA to or from `vmem_page`, the operating system must create a virtual-to-physical mapping for the corresponding proxy page, `PROXY(vmem_page)`. Each such mapping maps `PROXY(vmem_page)` to a physical memory proxy page, `PROXY(pmem_page)`. These mappings are created on demand. If the user process accesses a virtual memory proxy page that has not been set up yet, a normal page-fault occurs. The kernel responds to this page-fault by trying to create the required mapping. Three cases can occur, based upon the state of `vmem_page`:

- `vmem_page` *is currently in core and accessible.* In this case, the kernel simply creates a virtual-to-physical mapping from `PROXY(vmem_page)` to `PROXY(pmem_page)`.
- `vmem_page` *is valid but is not currently in core.* The kernel first pages in `vmem_page`, and then behaves as in the previous case.
- `vmem_page` *is not accessible for the process.* The kernel treats this like an illegal access to `vmem_page`, which will normally cause a core dump.

The kernel must also ensure that *I2* continues to hold when pages are remapped. The simplest way to do this is by invalidating the proxy mapping from `PROXY(vmem_page)` to `PROXY(pmem_page)` whenever the mapping from `vmem_page` to `pmem_page` is changed in any way.

Note that if `vmem_page` is read-only for the application program, then `PROXY(vmem_page)` should be read-only also. In other words, a read-only page can be used as the source of a transfer but not as the destination.

Maintaining *I3*: Content Consistency

The virtual memory manager of the operating system must guarantee invariant *I3* to maintain consistency between the physical memory and backing store.

Traditionally, the operating system maintains a dirty bit in each page table entry. The dirty bit is set if the version of a page on backing store is out of date, i.e. if the page has been changed since it was last written to backing store. The operating system may "clean" a dirty page by writing its contents to backing store and simultaneously clearing the page's dirty bit. A page is never replaced while it is dirty; if the operating system wants to replace a dirty page, the page must first be cleaned.

A page must be marked as dirty if it has been written to by incoming DMA, so that the newly-arrived data will survive page replacement. In traditional DMA, the kernel knows about all DMA transfers, so it can mark the appropriate pages as dirty. However, in UDMA, device-to-memory transfers can occur without kernel involvement. Therefore, we need another way of updating the dirty bits.

This problem is solved by maintaining invariant *I3*. Incoming transfers can only change a page if it is already dirty, so writes done by incoming UDMAs will eventually find their way to backing store.

As part of starting a UDMA transfer that will change page `vmem_page`, the user process must execute a STORE instruction to `PROXY(vmem_page)`. *I3* says that this STORE will cause an access fault unless `vmem_page` is already dirty. If the access fault occurs, the kernel enables writes to `PROXY(vmem_page)` so the user's transfer can take place; the kernel also marks `vmem_page` as dirty to maintain *I3*.

If the kernel cleans `vmem_page`, this causes `vmem_page`'s dirty bit to be cleared. To maintain *I3*, the kernel also write-protects `PROXY(vmem_page)`.

Race conditions must be avoided when the operating system cleans a dirty page. The operating system must make sure not to clear the dirty bit if a DMA transfer to the page is in progress while the page is being cleaned. If this occurs, the page should remain dirty.

There is another way to maintain content consistency without using using *I3*. The alternative method is to maintain dirty bits on all of the proxy pages, and to change the kernel so that it considers `vmem_page` dirty if either `vmem_page` or `PROXY(vmem_page)` is dirty. This approach is conceptually simpler, but requires more changes to the paging code.

Maintaining *I4*: Register Consistency

The operating system cannot remap any physical page that is involved in a pending transfer, because doing so would cause data to be transferred to or from an incorrect virtual address. Since transfers are started without kernel involvement, the kernel does not get a chance to "pin" the pages into physical memory.

Invariant *I4* makes sure that pages involved in a transfer are never remapped. To maintain *I4*, the kernel much check before remapping a page to make sure that that page's address is not in the hardware's `SOURCE` or `DESTINATION` registers. (The kernel reads the two registers to perform the check.) If the page is indeed in one of the registers, the kernel must either find another page to remap, or wait until the transfer finishes. If the hardware is in the *DestLoaded* state, the kernel may also cause an `Inval` event in order to clear the `DESTINATION` register.

Although this scheme has the same effect as page pinning, it is much faster. Pinning requires changing the page table on every DMA, while our mechanism requires no kernel action in the common case. The inconvenience imposed by this mechanism is small, since the kernel usually has several pages to choose from when looking for a page to remap. In addition, remapped pages are usually those which have not been accessed for a long time, and such pages are unlikely to be used for DMA.

For more complex designs, the hardware might allow the kernel to do queries about the state of particular pages. For example, the hardware could provide a readable "reference-count register" for each page, and the kernel could query the register before remapping that page.

7 Supporting Multi-Page Transfers with Queueing

The mechanism described so far can only support transfers within a single page. That is, no transfer may cross a page boundary in either the source space or the destination space. Larger transfers must be expressed as a sequence of small transfers.

While this is simple and general, it can be inefficient. We would like to extend our basic mechanism to allow large, multi-page transfers. The most straightforward way to do this is by queueing requests in hardware, which works as long as invariants *I1* through *I4* are maintained.

Queueing allows a user-level process to start multi-page transfers with only two instructions per page in the best case. If the source and destination addresses are not aligned to the same offset on their respective pages, two transfers per page are needed. To wait for completion, the user process need only wait for the completion of the last transfer. A transfer request is refused only when the queue is full; otherwise the hardware accepts it and performs the transfer when it reaches the head of the queue.

Queueing has two additional advantages. First, it makes it easy to do gather-scatter transfers. Second, it allows unrelated transfers, perhaps initiated by separate processes, to be outstanding at the same time.

The disadvantage of queueing is that it makes it more difficult to check whether a particular page is involved in any pending transfers. There are two ways to address this problem: either the hardware can keep a counter for each physical memory page of how often that page appears in the UDMA engine's queue, or the hardware can support an associative query that searches the hardware queue for a page. In either case, the cost of the lookup is far less than that of pinning a page.

Implementing hardware for multiple priority queues is straightforward, but not necessarily desirable because the UDMA device is shared, and a selfish user could starve others. Implementing just two queues, with the higher priority queue reserved for the system, would certainly be useful.

8 Implementation in SHRIMP

The first UDMA device — the SHRIMP network interface [5] — is now working in our laboratory. Each node in the SHRIMP multicomputer is an Intel Pentium Xpress PC system [12] and the interconnect is an Intel Paragon routing backplane. The custom designed SHRIMP network interface is the key system component which connects each Xpress PC system to a router on the backplane. At the time of this writing, we have a four-processor prototype running.

The network interface supports efficient, protected, user-level message passing based on the UDMA mechanism. A user process sends a packet to another machine with a simple UDMA transfer of the data from memory to the network interface device. The network interface automatically builds a packet containing the data and sends it to the remote node. The destination of the packet is determined by the address in the network interface device proxy space, where every page can be configured to name some physical page on a remote node.

Since we have already described the central UDMA mechanisms, readers should already understand how UDMA works. We will focus in this section on how the general UDMA idea was specialized for SHRIMP.

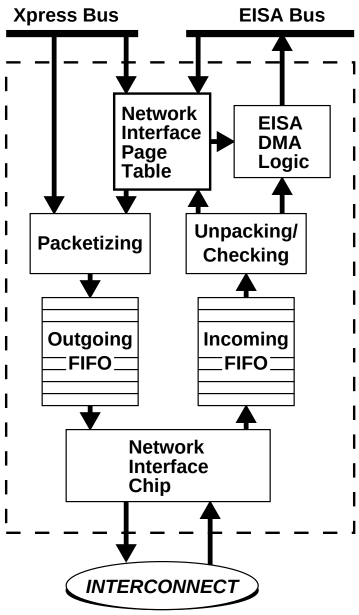

Figure 6: *SHRIMP network interface architecture.*

The SHRIMP Network Interface

Figure 6 shows the basic architecture of the SHRIMP network interface. The block labeled "EISA DMA Logic" contains a UDMA device which is used to transfer outgoing message data aligned on 4-byte boundaries from memory to the network interface. This device does not support multi-page transfers.

All potential message destinations are stored in the Network Interface Page Table (NIPT), each entry of which specifies a remote node and a physical memory page on that node. In the context of SHRIMP, a UDMA transfer of data from memory to the network interface is called "deliberate update". In this case, proxy device addresses refer to entries in the NIPT. A proxy destination address can be thought of as a proxy page number and an offset on that page. The page number is used to index into the NIPT directly and obtain the desired remote physical page, and the offset is combined with that page to form a remote physical memory address.

Proxy Mapping

Figure 7 shows how accesses to device proxy space are interpreted by the SHRIMP hardware. The application issues an access to a virtual proxy address, which is translated by the MMU into a physical proxy address. Proxy addresses are in the PC's I/O memory space, in a region serviced by the SHRIMP network interface board; accesses to proxy addresses thus cause

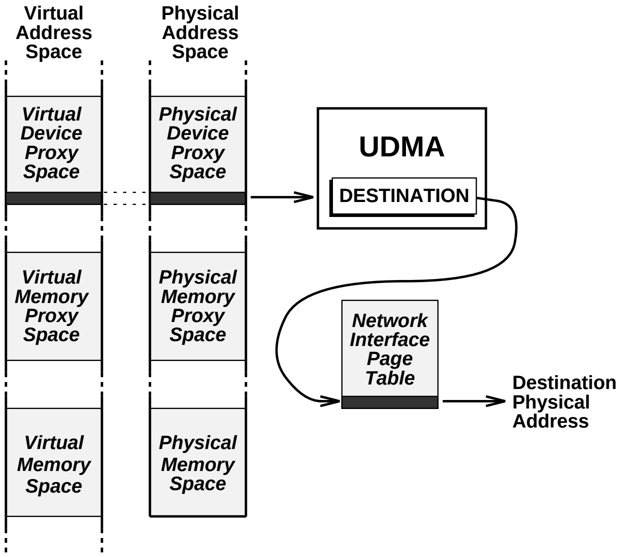

Figure 7: *How the SHRIMP network interface interprets references to proxy space.*

I/O bus cycles to the network interface board.

When the network interface board gets an I/O bus cycle to a physical device proxy address, it stores the address in the `DESTINATION` register and uses it to create a packet header. The address is separated into a page number and an offset. The rightmost 15 bits of the page number are used to index directly into the Network Interface Page Table to obtain a destination node ID and a destination page number. The destination page number is concatenated with the offset to form the destination physical address. Since the NIPT is indexed with 15 bits, it can hold 32K different destination pages.

Once the destination node ID and destination address are known, the hardware constructs a packet header. The packet data is transferred directly from memory by the UDMA engine using a base address specified by a physical memory proxy space `LOAD`. The SHRIMP hardware assembles the header and data into a packet, and launches the packet into the network. At the receiving node, packet data is transferred directly to physical memory by the EISA DMA Logic.

Operating System Support

The SHRIMP nodes run a slightly modified version of the Linux operating system. The modifications are mostly as discussed in section 6. In particular, the modified Linux maintains invariants *I1*, *I2*, and *I4*. (*I3* is not necessary because SHRIMP uses UDMA only for memory-to-device transfers, and *I3* is concerned only with device-to-memory transfers.)

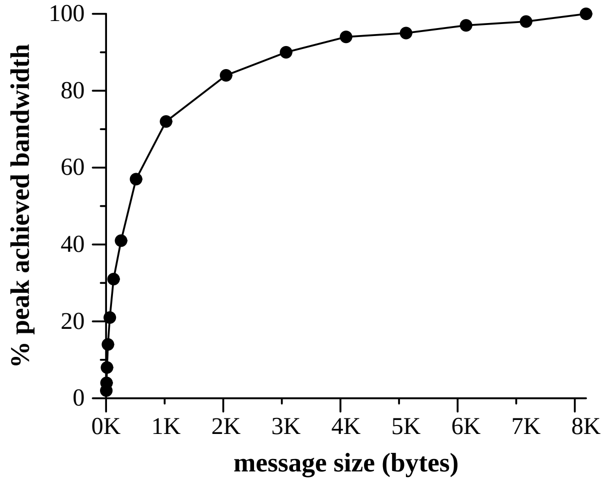

Figure 8: *Bandwidth of deliberate update UDMA transfers as a percentage of the maximum measured bandwidth on the SHRIMP network interface*

UDMA Hardware Performance

We have measured the performance of the UDMA device implemented in the SHRIMP network interface. The time for a user process to initiate a DMA transfer is about 2.8 microseconds, which includes the time to perform the two-instruction initiation sequence and check data alignment with regard to page boundaries. The check is required because the implementation optimistically initiates transfers without regard for page boundaries, since they are enforced by the hardware. An additional transfer may be required if a page boundary is crossed.

Figure 8 shows the bandwidth of deliberate update UDMA transfers as a percentage of the maximum measured bandwidth for various message sizes, as measured on the real SHRIMP system. The maximum is sustained for messages exceeding 8 Kbytes in size. The rapid rise in this curve highlights the low cost of initiating UDMA transfers.

The bandwidth exceeds 50% of the maximum measured at a message size of only 512 bytes. The largest single UDMA transfer is a page of 4 Kbytes, which achieves 94% of the maximum bandwidth. The slight dip in the curve after that point reflects the cost of initiating and starting a second UDMA transfer.

Summary

The SHRIMP network interface board is the first working UDMA device. It demonstrates that the general UDMA design can be specialized to a particular device in a straightforward way.

9 Related Work

Most of the interest in user-level data transfer has focused on the design of network interfaces since the high speed of current networks makes software overhead the limiting factor in message-passing performance. Until recently, this interest was primarily restricted to network interfaces for multicomputers.

An increasingly common multicomputer approach to the problem of user-level transfer initiation is the addition of a separate processor to every node for message passing [16, 10]. Recent examples are the Stanford FLASH [14], Intel Paragon [11], and Meiko CS-2 [9]. The basic idea is for the "compute" processor to communicate with the "message" processor through either mailboxes in shared memory or closely-coupled datapaths. The compute and message processors can then work in parallel, to overlap communication and computation. In addition, the message processor can poll the network device, eliminating interrupt overhead. This approach, however, does not eliminate the overhead of the software protocol on the message processor, which is still hundreds of CPU instructions. In addition, the node is complex and expensive to build.

Another approach to protected, user-level communication is the idea of memory-mapped network interface FIFOs [15, 6]. In this scheme, the controller has no DMA capability. Instead, the host processor communicates with the network interface by reading or writing special memory locations that correspond to the FIFOs. The special memory locations exist in physical memory and are protected by the virtual memory system. This approach results in good latency for short messages. However, for longer messages the DMA-based controller is preferable because it makes use of the bus burst mode, which is much faster than processor-generated single word transactions.

Our method for making the two-instruction transfer-initiation sequence appear to be atomic is related to Bershad's restartable atomic sequences [3]. Our approach is simpler to implement, since we have the kernel take a simple "recovery" action on every context switch, rather than first checking to see whether the application was in the middle of the two-instruction sequence. Our approach requires the application to explicitly check for failure and retry the operation; this does not hurt our performance since we require the application to check for other errors in any case.

Several systems have used address-mapping mechanisms similar to our memory proxy space. Our original SHRIMP design[5] used memory proxy space to specify the source address of transfers. However, there was no distinction between memory proxy space and device proxy space: the same memory address played both roles. The result was that a fixed "mapping" was required between source page and destination page, making the system less flexible than our current design, which allows source and destination addresses to be specified independently. In addition, our previous design did not generalize to handle device-to-memory transfers, and did not cleanly solve the problems of consistency between virtual memory and the DMA device. Our current design retains the automatic update transfer strategy described in [5] which still relies upon fixed mappings between source and destination pages.

A similar address-mapping technique was used in the CM-5 [17] vector unit design. A program running on the main processor of a CM-5 computing node communicated command arguments to its four vector co-processors by accessing a special region of memory not unlike our memory proxy space. Their use of address mapping was specialized to the vector unit, while UDMA is a more general technique. Also, CMOST, the standard CM-5 operating system, did not support virtual memory or fast context switching.

The AP1000 multicomputer uses special memory-mapped regions to issue "line send" commands to the network interface hardware. This mechanism allows a user-level program to specify the source address of a transfer. Unlike in UDMA, however, the AP1000 mechanism does not allow the size or destination address of the transfer to be controlled — the size is hardwired to a fixed value, and the destination address is a fixed circular buffer in the receiver's address. In addition, the transfer is not a DMA, since the CPU stalls while the transfer is occurring.

The Flash system [7] uses a technique similar to ours for communicating requests from user processes to communication hardware. Flash uses the equivalent of our memory proxy addresses (which they call "shadow addresses") to allow user programs to specify memory addresses to Flash's communication hardware. The Flash scheme is more general than ours, but requires considerably more hardware support — Flash has a fully programmable microprocessor in its network interface. The Flash paper presents three alternative methods of maintaining consistency between virtual memory mappings and DMA requests. All three methods are more complicated and harder to implement than ours.

10 Conclusions

UDMA allows user processes to initiate DMA transfers to or from an I/O device at a cost of only two user-level memory references, without any loss of security. A single instruction suffices to check for completion of a transfer. This extremely low overhead allows the use of DMA for common, fine-grain operations.

The UDMA mechanism does not require much additional hardware, because it takes advantage of both hardware and software in the existing virtual memory system. Special proxy regions of memory serve to communicate user commands to the UDMA hardware, with ordinary page mapping mechanisms providing the necessary protection.

We have built a network interface board for the SHRIMP multicomputer that uses UDMA to send messages directly from user memory to remote nodes.

Acknowledgements

This project is sponsored in part by ARPA under grant N00014-91-J-4039 and N00014-95-1-1144, by NSF under grant MIP-9420653, and by Intel Scalable Systems Division. Edward Felten is supported by an NSF National Young Investigator Award.

References

[1] American National Standard for information systems. *High-Performance Parallel Interface - Mechanical, Electrical, and Signalling Protocol Specification (HIPPI-PH)*, 1991. Draft number X3.183-199x.

[2] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 108–120, 1991.

[3] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, 1992.

[4] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Two virtual memory mapped network interface designs. In *Proceedings of Hot Interconnects II Symposium*, pages 134–142, August 1994.

[5] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. A virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 142–153, April 1994.

[6] FORE Systems. *TCA-100 TURBOchannel ATM Computer Interface, User's Manual*, 1992.

[7] John Heinlein, Kourosh Gharachorloo, Scott Dresser, and Anoop Gupta. Integration of message passing and shared memory in the stanford FLASH multiprocessor. In *Proceedings of 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1994.

[8] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1990.

[9] Mark Homewood and Moray McLaren. Meiko CS-2 interconnect elan – elite design. In *Proceedings of Hot Interconnects '93 Symposium*, August 1993.

[10] Jiun-Ming Hsu and Prithviraj Banerjee. A message passing coprocessor for distributed memory multicomputers. In *Proceedings of Supercomputing '90*, pages 720–729, November 1990.

[11] Intel Corporation. *Paragon XP/S Product Overview*, 1991.

[12] Intel Corporation. *Express Platforms Technical Product Summary: System Overview*, April 1993.

[13] Vineet Kumar. A host interface architecture for HIPPI. In *Proceedings of Scalable High Performance Computing Conference '94*, pages 142–149, May 1994.

[14] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford FLASH multiprocessor. In *Proceedings of 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[15] C.E. Leiserson, Z.S. Abuhamdeh, D.C. Douglas, C.R. Feynman, M.N. Ganmukhi, J.V. Hill, D. Hillis, B.C. Kuszmaul, M.A. St. Pierre, D.S. Wells, M.C. Wong, S. Yang, and R. Zak. The network architecture of the connection machine CM-5. In *Proceedings of 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 272–285, June 1992.

[16] R.S. Nikhil, G.M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of 19th International Symposium on Computer Architecture*, pages 156–167, May 1992.

[17] Thinking Machines Corporation. *CM-5 Technical Summary*, 1991.