Issues in the Design and Implementation of User-Level DMA

Evangelos P. Markatos Manolis G.H. Katevenis
George Kalokerinos Gregory Maglis
George Milolidakis Thanos Oikonomou

Institute of Computer Science (ICS)
Foundation for Research & Technology – Hellas (FORTH)
P.O.Box 1385, Science and Technology Park of Crete,
Heraklion, Crete, GR-711-10 GREECE
markatos@ics.forth.gr
Technical Report 182, ICS-FORTH
URL: http://www.ics.forth.gr/proj/arch-vlsi/telegraphos.html

1 Introduction

The goal of several current supercomputing projects is to demonstrate supercomputer performance at workstation cost. A supercomputer is being created by interconnecting a set of high-performance workstations via a high-speed SCI interconnect. Host workstations are connected to the network over Dolphin's PCI-SCI interface [5] that plugs in the PCI I/O bus of the workstation.

This interface has been carefully designed so as to achieve supercomputing-like communication performance over a network of workstations (NOW). To achieve very low message-passing latency, the interface implements a remote-write operation. The remote-write operation (also called direct-deposit) is initiated by a `store` assembly instruction to a non-local memory location. Using the remote write primitive, a processor may write a message directly to its destination memory using regular `store` instructions to non-local memory locations. For example, suppose that a processor has a single-word message stored in variable `source`, and wants to send it to a remote processor that will store it in variable `destination`, the sending processor can send the message by executing a single assignment instruction:

```
destination := source ;
```

Most compilers will translate the above assignment statement into a two-instruction sequence:

LOAD Register1 **FROM** source_address ;
STORE Register1 **TO** destination_address ; // *remote write*

The **LOAD** instruction fetches the single-word message into a processor's register, and the **STORE** instruction sends the message directly to its destination. This **STORE** instruction is also called a remote-write operation. Although remote-write operations achieve very low latency for sending *short* messages, they are expensive when sending large messages as sequences of remote write operations. [1] To overcome this problem, the PCI-SCI interface (along with similar high-speed interfaces) provide a DMA operation. The DMA transfers a large chunk of data from the host computer's main memory into the network interface, and from there into the SCI network without keeping the host processor busy during the transfer. The host processor is only needed to initialize the DMA transfer, and to be notified of its completion. During the DMA transfer, the processor is free to execute other useful work. Besides freeing the processor, DMA transfers impose low memory bus traffic, since they transfer data directly from the

[1] To draw an analogy from real life, lets consider the fax. The fastest way to send a *short* document is probably to fax it to the recipient. When faxing a document, the sender essentially deposits the information *directly* at the receiver's office (much like a remote-write or direct-deposit operation). However, sending long documents (e.g. several books) using fax, is not a good idea as it results in slow and expensive data transfer.

main memory to the network interfaces. On the contrary, each remote-write operation, transfers the data from the memory to a processor register (loading from the source address), and then from the register to the remote memory (storing to the destination address), thereby transferring each word twice over the memory bus, and polluting the processor's cache during these load and store operations.

For all the mentioned reasons, large messages are transferred from main memory to the network interface using DMA transfers. DMA management has been traditionally done by the Operating System kernel. The Operating System is the only trusted entity that is allowed to access DMA registers. User applications are not allowed to initiate DMA operations by themselves. There are two reasons for the necessity of the Operating System involvement in starting a DMA operation in traditional systems:

- *Atomicity*: To start a DMA operation, the software should pass several arguments to a DMA engine. At least three arguments are needed: the source address, the destination address, and the size of the DMA transfer. All these arguments should be given to the DMA engine atomically, otherwise, two processes that want to initiate two DMA operations at about the same time may overwrite each other's arguments, in their attempt to grab the DMA engine. To resolve such race conditions, in traditional systems, processes invoke the operating system which runs uninterrupted, starts the DMA operation of the first process, and when finished, starts the DMA of the second process.

- *Protection from programming errors and malicious users:* Most DMA engines accept only physical (PCI or SCI) addresses as the source and destination address of a DMA operation. Ordinary users should not be allowed to pass physical addresses to a DMA engine, since they may pass physical addresses, that they are not allowed to access. Thus, an ignorant or malicious user may start a DMA operation from/to memory addresses that (s)he normally has no access to. As a result, (s)he may read private data, destroy the operating system, or crash the computer. The only trustworthy entity to determine which user is allowed to access which physical addresses is the operating system.

In the previous decades, since the overhead of the operating system involvement in the initiation of a DMA was small compared to the DMA data transfer itself, no attempt was made to allow user applications to start DMA operations. However, in contemporary fast local area networks, starting a DMA operation from inside the operating system kernel may take more than the network transfer operation itself! For this reason, several researchers have started to address the problem of letting user applications initiate a DMA. Pioneering work in the SHRIMP [1] and FLASH [4] projects have pinpointed the importance of user-level DMA operations and have proposed initial solutions to user-level DMA. Unfortunately, these approaches to user-level DMA require modifications to the operating system kernel. To function correctly, both mentioned approaches modify the operating system context switch handler, in order to enforce atomicity of user-level DMA operations, and avoid race conditions. The SHRIMP approach requires that the context switch handler aborts all half-started DMA operations [2] (so that no race condition may happen), while the FLASH approach requires that the context switch handler informs the DMA engine about the identity of the running process at context switch time, (so that the DMA engine has enough information to avoid race conditions). Although a few lines of code to the context switch handler seem a trivial change, they may turn out to be a major obstacle to the success of user-level DMA for the following reasons:

- Modifications of the operating system kernel may not be possible because the source code of the operating system may be confidential or sold under a license only. In either case, ordinary users may not be able, or willing to acquire operating system sources. Even if the changes to the context switch handler are distributed as an operating system patch, they may generate even more problems:
  - Distributing changes (for user-level DMA) to existing operating system, as patches, sets a bad example. If all peripheral device vendors start distributing patches to existing operating systems, different patches will eventually conflict with each other, leading to erroneous code.
  - Patches are difficult to maintain. They force the vendor of the DMA device to produce a new patch for each new version of the operating system. Moreover, end-users are required to install these patches each time their operating system is being upgraded.

- The context switch handler is usually on the critical path of the performance of the operating system. If each manufacturer of each device adds a few lines of code to the context switch handler, the Operating System performance would be significantly lower.

In this paper we propose several solutions to the user-level DMA problem that require no modifications to the operating system kernel. Two of them are novel, and the other two are elaborations of our older designs. Our methods allow user applications to securely and atomically start DMA operations from user-level without needing to change the operating system kernel.

2 User-Level DMA - Early Work

2.1 The Problem

A DMA operation has (at least) three arguments: `DMA (vsource, vdestination, size)`. Its function is to transfer `size` number of bytes from virtual address `vsource`, to virtual address `vdestination`. To simplify their operation, DMA engines usually operate only on physical addresses. Sometimes they are able to operate on virtual addresses, but this functionality makes both hardware and software more complicated: the DMA hardware would need to include translation tables to translate virtual to physical addresses, while the operating system software would need to keep these tables up to date. Thus, most DMA engines require physical addresses as arguments. Hence, the problem with starting a DMA operation from user-level is twofold:

- `Protection`: Users should not be allowed to pass physical addresses directly to the hardware without any protection checking. Otherwise, an erroneous or malicious application may start DMA operations to physical addresses on which it has no access rights, compromising security, or damaging memory contents.
- `Atomicity:` since two addresses are needed for each DMA operation, these two addresses should be passed atomically to the DMA engine. Otherwise, a random interleaving of processes that want to start DMA's may result in a mix up of arguments and may start data transfers from (to) wrong addresses.

2.2 Kernel-Level Initiation of DMA

The traditional solution to the above problem is that the user application wanting to make a data transfer calls the operating system with the necessary arguments: `vsource`, `vdestination` and `size`. The operating system runs (with interrupts disabled), checks size, translates the virtual addresses `vsource` and `vdestination` to their corresponding physical addresses `psource` and `pdestination`, writes the arguments `psource`, `pdestination`, and `size` to the DMA engine registers, and starts a DMA transfer. The pseudo-code necessary to start a DMA operation from inside the operating system is shown in figure 1.

DMA(vsource, vdestination, size)
    psource = virtual_to_physical(vsource) */* translate virtual address*/*
    pdestination = virtual_to_physical(vdestination)
    check_size() ; */* check protection in whole transfer range */*
    STORE psource **TO** DMA_SOURCE */* set DMA registers*/*
    STORE pdestination **TO** DMA_DESTINATION
    STORE size **TO** DMA_SIZE */* start DMA */*
    LOAD status **FROM** DMA_STATUS */* succeeded? */*

Figure 1: Typical Initiation of kernel-level DMA

Note, that *all* mentioned instructions are executed uninterrupted (in kernel mode), and thus the atomicity of DMA initiation is guaranteed. Translation from virtual addresses to physical addresses is being done inside the operating system in software, by the `virtual_to_physical` function. During address translation, the access rights of the user are checked to make sure that the user process who requested the DMA operation has read access to the page that contains address `vsource`, and write access to the page that contains address `vdestination`.

Although the overhead of the operating system involvement in starting a DMA operation has been considered low (especially for DMA operations that transfer data between main memory and disk), this is not the case for network transfers anymore. It is well known that Operating Systems do not get faster as fast as hardware does [8, 9]. Operating System overhead (measured in processor cycles) continues to increase with time. Large sets of registers that need to be saved/restored, lack of data locality, and slow I/O peripherals are some of the reasons why operating systems do not get faster as fast as processors do. Recent performance results suggest that the overhead of an empty system call of commercial UNIX-like operating systems ranges between 1,000 and 5,000 processor cycles [7]. At the same time, we witness a impressive improvement in network throughput. For example, the sustained data transfer over the mentioned SCI network is more 400 Mbits/sec. Gigabit LANs have already started to appear in the market. Thus, the operating system overhead keeps getting an ever-increasing percentage of the DMA transfer time, while the time for the data transfer per se, continues to decrease. Soon, the operating system overhead will dominate the DMA transfer, making the necessity of user-level DMA more important than ever.

2.3 Passing Physical Addresses

Before we describe the various user-level DMA mechanisms, we will discuss the notion of *shadow addressing*, that is common to all user-level DMA solutions, and has been proposed (under various names) in [2, 4]. The method of shadow addressing is used to securely translate virtual to physical addresses and pass them to the DMA engine from user-level processes. For each virtual address `vaddr` that is mapped in the physical address `paddr`, there is also a shadow address `shadow(vaddr)`, which is mapped in the shadow physical address `shadow(paddr)`.[2] The shadow function is simple and known to the DMA engine. One simple shadow function is to concatenate each address with an extra shadow bit. When the shadow bit is set, then the address is a shadow one. For example, 0x0FFFFFFFF is a regular 33-bit address, while 0x1FFFFFFFF is its shadow address.

An access to a shadow address is always interpreted by the DMA engine as a special argument passing operation. For example, suppose that virtual address `vaddr` is mapped to physical address `paddr`, and that the virtual address `shadow(vaddr)` is mapped into `shadow(paddr)`. Normally, a load (store) operation to virtual address `vaddr` by a user application is translated by the TLB (page-table) into a load (store) operation to physical address `paddr` and is performed by the appropriate memory controller. [3] Similarly, a load (store) operation to virtual address `shadow(vaddr)` is translated by the TLB into a load (store) operation to physical address `shadow(paddr)`. When, however, this operation reaches the DMA engine it will be treated as an argument passing operation, and neither a load nor a store operation will be performed to physical address `shadow(paddr)`. Thus, when the user application wants to pass to the DMA engine the physical address `paddr`, it makes an access to virtual address `shadow(vaddr)`. Eventually, the access mode along with the physical address `shadow(paddr)` reach the DMA engine. The DMA engine recognizes the shadow address and takes the physical address `paddr` by applying function $\texttt{shadow}^{-1}$ to physical address `shadow(paddr)`. [4]

2.4 The first SHRIMP solution

Blumrich *et al.* described one of the first user-level DMA solutions [1], developed in conjunction with the SHRIMP prototype. In SHRIMP, each page that is used for communication, is "mapped out" to another page in a different workstation. In this DMA mode of operation, if a local page is used as the `source` argument in a DMA operation, the `destination` argument will always be its mapped-out

[2] The Operating System is responsible for creating both mappings at memory allocation (initialization) time.

[3] Dolphin's PCI/SCI interface already provides an addressing mode that is reminiscent of shadow addressing. Specifically, in the 36-bit address mapping, the highest bit of each SCI address (called the lock bit) has a special meaning: if it is set, a **fetch_and_add** atomic operation is performed on the supplied address. Thus, a `load` operation from this address will trigger a **fetch_and_add** atomic operation, instead of a regular load operation.

[4] All shadow addresses should be within the physical address range of the DMA engine, and distinct from the normal physical addresses used by that engine.

page. To start a DMA operation, an access to a shadow address is performed. This access, passes to the DMA engine, the source address, the destination address (the "mapped out" page), the size of the DMA, and returns the success/failure of the DMA initiation. All this information is passed by using a `compare-and-exchange` atomic instruction. The address argument of the instruction is the `source` address, the data argument of the instruction is the `size` of the transfer, and the return value is used to determine if the DMA operation has started correctly. [5]

This solution, although correct, is of limited functionality. A DMA operation can happen only between a page and its mapped out counterpart, which is very restrictive in practice. To achieve arbitrary user-level DMA transfers, the mapping between a page and its "mapped-out" page would have to change, which would result in significant operating system overhead.

2.5 The second SHRIMP solution

In their subsequent work, Blumrich *et al.* [2] developed a more general user-level DMA method. Their solution is based on the following idea: the two physical addresses needed to start a DMA will be given to the network interface by accessing the two shadow addresses. Their solution to user-level DMA is shown in figure 2.

```
DMA(vsource, vdestination, size)
    /* pass physical address shadow(pdestination) to the
    ** DMA engine, and the size of the transfer */
    STORE size TO shadow(vdestination)
    /* pass physical address shadow(psource) to the
    ** DMAengine and read if the operation was successful */
    LOAD return_status FROM shadow(vsource)
```

Figure 2: **SHRIMP solution to user-level DMA.** This elegant solution to user-level DMA manages to pass all the arguments needed for the DMA to the network interface with only few instructions. Kernel modification is necessary to ensure atomic initiation of user-level DMA.

The `STORE` instruction is used to pass to the SHRIMP interface the physical address that corresponds to virtual address `vdestination`, and the `size` of the DMA transfer. The `LOAD` operation is used to pass to the SHRIMP interface the physical address that corresponds to virtual address `vsource`, and to return the status of the DMA initiation.

This solution has the following limitation: If the user process is interrupted *after* the `STORE` operation, but *before* the `LOAD` operation, then its arguments to the DMA operation may get mixed with arguments of other processes that want to start a user-level DMA and eventually a wrong DMA operation may be started. To alleviate this problem, Blumrich *et al.* suggested that "the operating system must invalidate any partial initiated user-level DMA transfer on every context switch", which implies that the context switch code (and the operating system kernel) must be changed to provide the support for the mentioned invalidations. Although, context switch code may be modified for the purposes of a research prototype, this severely limits the portability of user-level DMA in the market, for the reasons described in the introductory section.

2.6 The FLASH solution

Heinlein *et al.* [4] implemented user-level DMA within the context of the FLASH multiprocessor. A user-level DMA is initiated using a sequence of uncached accesses to shadow addresses (much like the SHRIMP approaches). To provide atomicity in user-level DMA, the context switch handler informs the DMA engine about which process is currently running. Thus, the DMA engine knows which process runs, and makes sure that DMA arguments belonging to different processes do not get mixed. This solution, just like the previous one, needs to modify the context switch handler, to inform the DMA engine of the identity of the running process at each context switch.

[5]Recall, no `destination` argument needs to be passed, because the destination for the DMA transfer, is the mapped-out page of the source address.

2.7 The PAL Code approach

We have seen so far that the mechanism of shadow addressing is a fast and reliable method to pass physical addresses to a DMA engine from user-space. What is difficult however, is to achieve atomicity of a user-level DMA operation, that is, to pass *both* physical addresses to hardware, without any danger of mixing physical addresses of different processes, and create race conditions. All previous approaches solve the problem of atomicity, by changing the context switch code to either abort semi-initiated DMA operations (e.g. in SHRIMP), or explicitly tell the DMA engine that a context switch has happened (e.g. in FLASH). If only there were a way to execute two assembly instructions *uninterrupted* from user-space, then the atomicity problem would have been solved. Unfortunately, traditional systems do not allow user-level processes to execute uninterrupted code because malicious users may monopolize the computing system. A recent processor, however, the DEC Alpha processor, provides a special mode of execution, the PAL mode, which allows uninterrupted execution [10]. PAL code is organized in 16-instruction long PAL calls. A PAL call is executed uninterrupted. To ensure protection, only super-users are allowed to write and install PAL functions. However, once a PAL function is installed, any ordinary user is allowed to invoke it.

User-level DMA atomicity can be achieved by turning the two instructions needed to start a DMA operation into a PAL call. Thus, the pseudo-code to start a DMA operation would look as follows:

```
DMA(vsource, vdestination, size)
    call_pal user_level_dma(vsource,
             vdestination, size)
```

and the `user_level_dma` PAL call would be implemented as:

DMA(vsource, vdestination, size)
    STORE size **TO** shadow(vdestination)
    LOAD return_status **FROM** shadow(vsource)

This solution achieves user-level DMA without any changes to the operating system kernel code. We believe that systems equipped with the Alpha processor should use this method of user-level DMA. The PAL code solution to user-level DMA has been incorporated into the Telegraphos I network interface [6].

3 User-level DMA without Kernel Modifications

3.1 User-level DMA based on keys

/ the KEY allows the process to write arguments into CONTEXT_ID */*
global KEY, CONTEXT_ID ;
/ The reg. context CONTEXT_ID is mapped into address REGISTER_CONTEXT */*
global address REGISTER_CONTEXT ;
DMA(vsource, vdestination, size)
    / pass the destination argument */*
    STORE KEY#CONTEXT_ID **TO** shadow(vdestination)
    STORE KEY#CONTEXT_ID **TO** shadow(vsource) */* pass the source argument */*
    STORE size **TO** REGISTER_CONTEXT ; */* pass the size argument */*
    LOAD return_status **FROM** REGISTER_CONTEXT ; */* did it succeed? */*

Figure 3: The key-based approach to user-level DMA

Although the PAL code approach to user-level DMA is simple, it requires the host processor to be the Alpha processor. In this section we will describe a method to provide atomic user-level DMA without the need to execute uninterrupted code. The idea behind our approach is the following:

> The DMA engine is equipped with several (say 4 to 8) register contexts. Each context has a `source` register, a `destination` register, and a `size` register, with the obvious meanings.

Each context is mapped into memory address space so that the processor can access it. Distinct contexts are mapped into distinct memory pages so that each process gets access rights for only a single context. Each process that is allowed to start user-level DMA operations is allowed to write into one such context (the operating system divides the sets fairly and efficiently among competing processes). These registers are being used to keep arguments to the DMA operations for each process. Thus, if a process gets interrupted while starting a DMA operation, its arguments can not be mixed with another process's arguments, since each process has its own set of context registers to write its arguments into.

The idea sounds simple: each process has its own space in the DMA engine so that DMA arguments from two different processes do not get mixed as a result of context switch.

Unfortunately, a user-level application can not use regular load and store operations to access these registers and load them with the arguments of a DMA operation. Recall, that in user-level DMA, argument passing is done using shadow addressing - the address of the load/store operation is a shadow address, and is used as an argument to the DMA operation. Thus, a process that would like to pass a physical address to a register context, will pass the context identification as *data argument* of the store operation, since the *address argument* of the store operation has already been reserved to pass the shadow address. For example, to write the physical address that corresponds to virtual address `vaddress`, into a register context, a user-level process would execute the following instruction:

```
STORE context_id TO shadow(vaddress)
```

The above instruction is interpreted by the DMA engine as follows: Extract the `paddress` from the `shadow(paddress)`, and put it in register context `context_id`. Effectively, to start a DMA, a process makes a sequence of uncached store operations like the above one. Unfortunately, in this way, *any* user process will be allowed to write an address argument into *any* register context. To prohibit this erroneous behavior, along with the context identification, a key is passed in the data argument of the store operation. The key is given to the user process by the operating system. Possession of the key implies that the user process is allowed to write to this register context. Thus, a physical address is passed to a DMA engine as follows:

```
STORE key#context_id TO shadow(vaddress)
```

The above instruction is interpreted by the DMA engine as follows: Use the physical address that corresponds to `shadow(paddress)`, and store it as an argument in the register context `context_id`, only if the provided `key` matches the key stored by the operating system in the DMA engine, in memory locations un-readable by user processes.

Using the above instruction the address arguments of the DMA operations are securely passed to the DMA engine. However, one more argument needs to be passed: The `size` of the DMA transfer. This is passed using a regular `store` operation to the address that corresponds to the register context. Any store operation to any register within a context is being performed to the `size` register only, i.e. the user can not read/write the `source`, and `destination` registers of a register context using regular `load/store` operations, otherwise, (s)he would be able to start DMA from/to illegal addresses. Thus, although the register context is mapped in an process' address space, the process can only modify the `size` register of the context. A read operation from a register context returns the number of bytes that need to be transferred yet (-1 means failure, 0 means completed DMA operation).

A user-level DMA operation is initiated as shown in figure 3.

The first two `STORE` operations pass the physical addresses that correspond to the arguments of the DMA operation. The third operation, stores the size of the DMA transfer to the register context of the current user process. Finally the last operation initiates the DMA operation and reads the status result back.

The reader will notice that both address arguments are passed using `store` instructions, while in previous solutions, the source address argument was passed using a `load` instruction. This restriction implies that only processes that have both read and write access to the `source` address will be able to do user-level DMA operations from it. We believe that this is not a significant limitation. Most parallel and distributed applications that send data using DMA, have both read and write access to these data.

Another limitation of this method seems to be its probabilistic nature: a lucky user may "guess" a key and may start illegal DMA transfers. We believe that this is highly unlikely: In 64-bit architectures, there will be close to 60 bits available for the key field, which makes the probability of guessing correctly practically zero. It would be easier for a malicious process to crack the password of another user, rather than to guess a DMA key.

3.2 Extended Shadow Addressing

DMA(vsource, vdestination, size)
    / pass physical address shadow(pdestination) to the*
    *** DMA engine, and the size of the transfer */*
    STORE size **TO** shadow(vdestination)
    / pass physical address shadow(psource) to the*
    *** DMAengine and read if the operation was successful */*
    LOAD return_status **FROM** shadow(vsource)

Figure 4: User-level DMA based on extended shadow addressing

Although the previous solution achieves user-level DMA without operating system kernel modifications, it can theoretically be broken by a lucky user who manages to guess another user's key. To avoid this problem, we have developed a user-level DMA solution that makes the identification of the process part of the shadow address. That is, some bits (e.g. the highest ones) of the physical address that will be passed as an argument to the DMA engine correspond to the process identification. [6] These bits are set by the operating system when it creates the mappings from shadow virtual addresses to shadow physical addresses. Part of the shadow physical address is now the `CONTEXT_ID`. We envision the `CONTEXT_ID` to be 1-2 bits long. Thus, 2-4 processes will be able to start user-level DMA operations from the same processor. If more processes would like to start DMA operations, the rest will have to go through the kernel. We believe that allowing 1-2 bits of the physical address for the `CONTEXT_ID` is enough for most practical cases.

A typical shadow address looks like:

shadow bit:1	context id:1	address:62

A user-level DMA operation will be initiated exactly as the in the second SHRIMP solution - only the shadow addresses will be different - see figure 4.

By checking the `CONTEXT_ID`, the DMA engine knows which process the shadow address belongs to. Effectively, this user-level DMA solution is similar, in principle, to the FLASH solution: in both cases the DMA engine knows which process issues which shadow access. The difference is that this information in our solution is embedded in the shadow address, while in the FLASH solution the operating system kernel is modified to pass the information to the DMA engine. If the DMA engine has several register contexts, it may save these addresses it receives in the appropriate contexts and start the DMA operations when all arguments are available. If the DMA engine has no register contexts, then when it receives pairs of `STORE`, and `LOAD` instructions, it checks for the `CONTEXT_ID` values of the two physical addresses. If they are different, the DMA operation is not started and an error code is returned by the last `LOAD` instruction.

3.3 Repeated passing of arguments

Our final solution achieves user-level DMA without the need of extra bits in the physical address. It is based on an idea proposed by Charek Dubnicki [3]: If a process passes at least one shadow address more than once, then the DMA engine may be able to determine if a user process was interrupted by checking the two successive accesses to the same shadow address. Dubnicki's solution uses a three-instruction sequence as follows:

[6] To be consistent with the previous description we will use the `CONTEXT_ID` as the process identification.

DMA(vsource, vdestination, size)
1: **LOAD** status1 **FROM** shadow(vsource)
2: **STORE** size **TO** shadow(vdestination)
3: **LOAD** status2 **FROM** shadow(vsource)

The DMA engine initiates a DMA transfer only if it sees a sequence of the form `LOAD`, `STORE`, and `LOAD`, and the address arguments of the first and third instruction in the sequence are the same. [7] If a process is interrupted while trying to start a DMA, then the DMA engine will probably receive a non-valid sequence of shadow addresses, and no DMA operation will be initiated.

```
  LEGITIMATE  PROCESS           MALICIOUS PROCESS
1:LOAD status1 from shadow(A)
2:                              STORE foo TO shadow(foo)
3:                              LOAD status2 FROM shadow(foo) <- DMA is not started
4:                              LOAD status1 FROM shadow (C)
5:STORE size to shadow (B)
6:                              LOAD status2 FROM shadow(C)   <- DMA is  started
7:LOAD ... from shadow (A)                                    <- too late to do anything
```

Figure 5: Possible interleaving in the 3-instruction *Repeated passing of argument DMA* approach. A malicious user is able to start a DMA and transfer its own data (`C`), into another process's address space (`B`).

Although seemingly correct, the above solution may lead to erroneous data transfers, if abused by malicious users. Assume, for example, that we have a legitimate process that wants to start a DMA, and a malicious process that wants to interfere. A possible interleaving of shadow accesses is shown in figure 5. In this interleaving, the instructions `1:` to `3:` reach the DMA engine, but no DMA operation is started. However, then next three instructions (`4:` to `6:`) appear as a valid DMA sequence, and thus a DMA operation is started transferring data from address `C` to `B`, while the legitimate process wanted to transfer data from `A` to `B`.

Straightforward 4-instruction sequence extensions to the above solution seem to remedy this situation. For example, assume the following obvious 4-instruction extension:

DMA(vsource, vdestination, size)
1: **STORE** size **TO** shadow(vdestination)
2: **LOAD** return_status1 **FROM** shadow(vsource)
3: **STORE** size **TO** shadow(vdestination)
4: **LOAD** return_status2 **FROM** shadow(vsource)

If a malicious does not have any access to addresses `vsource`, and `vdestination`, then the above sequence seems to operate correctly. If however, the data contained in `vsource` are such that they can be read by any process in the system, then a malicious user may achieve the interleaving shown in figure 6. Suppose that the legitimate process wants to transfer data from `A` to `B`, and the malicious process has read-only access to `A`. In the interleaving shown in figure 6, the malicious process initiates the DMA transfer (in `4:`), but the DMA engine tells the legitimate process that the DMA transfer was *not* initiated (in `5:`). Such a behavior will probably lead several applications to erroneous behavior.

To remedy the above limitation, we have developed a 5-instruction sequence that achieves user-level DMA. The `shadow(vsource)` address is passed twice to the DMA engine, while the `shadow(vdestination)` address is passed three times, as shown in figure 7. The DMA engine is prepared to receive 5-instruction sequences to shadow address space. The sequence should be of the form `STORE, LOAD, STORE, LOAD, LOAD`. If it sees anything out of this order, the DMA engine resets itself, waiting for the 5-instruction

[7] Some hardware devices (e.g. write buffers) may attempt to collapse successive read/write operations to the same address. In these cases appropriate memory barrier commands should be used to ensure that all issued instructions will reach the DMA engine.

```
  LEGITIMATE PROCESS           MALICIOUS PROCESS
1: STORE size TO  shadow(B)
2: LOAD rs FROM shadow(A)
3: STORE size TO  shadow(B)
4:                             LOAD rs FROM shadow (A) <- DMA is started
5: LOAD rs FROM shadow (A)                             <- DMA is rejected
```

Figure 6: Possible interleaving in the 4-instruction *Repeated passing of argument DMA* approach. The malicious process starts the DMA (in `4:`) but misinforms the legitimate process that the DMA did not start (in `5:`).

DMA(vsource, vdestination, size)
1: **STORE** size **TO** shadow(vdestination)
2: **LOAD** return_status **FROM** shadow(vsource)
If (return_status != DMA_OK1) goto 1:
3: **STORE** size **TO** shadow(vdestination)
4: **LOAD** return_status **FROM** shadow(vsource)
If (return_status != DMA_OK2) goto 1:
5: **LOAD** return_status **FROM** shadow(vdestination)
If (return_status != DMA_OK3) goto 1:

Figure 7: User-level DMA by repeated passing of arguments.

sequences. A DMA operation is started only if the DMA engine receives a sequence of the type `STORE, LOAD, STORE, LOAD, LOAD`, and the address arguments of instructions `1,3` and `5` are the same, and the address arguments of instructions `2` and `4` are the same as well. [8]

4 Prototype

To verify the correctness and performance of our algorithms, we implemented a prototype board for user-level DMA according to the "repeated passing of arguments" method. The board was plugged in the TurboChannel I/O bus of a DEC Alpha 3000/300 workstation. All the required logic for the user-level DMA was implemented within a single FPGA.

4.1 Finite State Machine and Datapath Description

The design utilizes a four-state synchronous FSM (*S0*, *S1*, *S2*, *S3* and *S4*) shown in figure 8. The FSM jumps from one state to another when a Turbo Channel transaction takes place and remains in a state as long as there are no new transactions. A new transaction is implied by sampling signal FIRSTSEL high. There are two Turbo Channel instructions received, `load` (signal RW_=1) and `store` (signal RW_=0).

There are virtually three 12-bit registers used in this construct, named **DEST**, **SOURCE** and **STATUS**. **DEST** holds the destination address of the intended DMA, **SOURCE** holds the source address [9] of the intended DMA and **STATUS** holds the response of the DMA machine to Turbo Channel load instructions. The values of **STATUS** are: OK1=001h, OK2=002h, OK3=003h and FAIL=000h.

[8] A DMA operation is correctly initiated as long as processes that want to use user-level DMA do not share any data. In the case where competing processes share data, they need to synchronize (e.g. using locks) before using the described user-level DMA algorithm, otherwise they may mix up their arguments: if the DMA engine sees the same address twice, it does not know if it is part of a legitimate user-level DMA sequence, or if it is part of two user-level DMA sequences started by two different processes that share the mentioned address.

[9] Although the Turbo Channel support 27-bit addresses, our FPGA manipulates only the 12 highest bits of each address in order to reduce the space requirements of our FPGA.

The FSM resets at *S0* and waits for a `store` instruction to go to *S1*. On receiving a `load` it responds with FAIL, while remaining in *S0*. When a `store` arrives the bus address is saved to **DEST** and state alters to *S1*.

If a `load` instruction comes now, the bus address is stored to **SOURCE**, OK1 is returned and the FSM goes to *S2*. Otherwise the FSM returns to *S0*.

Once in *S2* to move to the next state (*S3*) a `store` instruction with address equal to **DEST** is expected. Otherwise the FSM resets to *S0* and if the instruction is `load`, FAIL is put to the bus as response.

Now, to jump to state *S4* the FSM is waiting for a `load` instruction with address equal to **SOURCE**. In that case OK2 is returned. If the address is not equal to **SOURCE** or the instruction is a `store` the FSM returns to *S0* (responding with FAIL to `load`).

S4 is the last state and always returns to *S0*. If the instruction is a `load` then if the bus address is equal to **DEST**, OK3 is returned and the DMA starts, else FAIL is returned. If the instruction is a `store` then the FSM just resets and no DMA starts.

In conclusion, the FSM goes through all five states only if the sequence of the instructions received is `store-load-store-load-load`. In addition, virtual register **STATUS** contains the value OK3 (which indicates that the instruction sequence succeeded and the DMA will start) only if the addresses of the 1st, 3rd and 5th instructions are equal as well as the addresses of the 2nd and 4th instructions.

The datapath of a Turbo-Channel slave FPGA, implementing the repeated passing of arguments algorithm is shown in figure 9. The signal **CLK** is the global clock of the FPGA (actually the Turbo-Channel's clock). **RW_** is the Turbo-Channel operation signal. On a read (load) operation RW_ is 1 and on a write (store) it is 0. Signal **SEL_** is driven low as soon as an address is put on the bus lines and stays low during the whole transaction process. The signal **RDY_** is driven low by the FPGA and stays down for a clock cycle. RDY_ is used to inform about the end of a transaction and when it is driven high the signal SEL_ follows. The FPGA uses only 12 of the 32 Turbo-Channel address/data lines, named **Address** in the diagram. On these lines the processor puts the load/store address and in the case of a load the FPGA answers with the data.

The **FSEL** signal samples the first time during a transaction the signal SEL_ is driven low. In other words FSEL marks the beginning of a transaction (it stays asserted for only one clock cycle). When FSEL is asserted the bus address is stored in a register. If the signal **SRC_LD** is asserted then the address will be stored to the register **SOURCE**. If **DST_LD** is asserted then the address is copied to the **DEST** register. This will be the case on the first two of the five transactions of the DMA starting sequence. On any other case the current address and the contents of the SOURCE or DEST registers are compared and the result of this comparison (signal **EQUAL**) goes to the FSM. During this cycle the signal **FIRSTSEL** is also generated and passed to the FSM.

On the next cycle the RDY_ signal is generated, along with **STAT_SEL** and **STATUS**. STATUS is the value returned on a load and STAT_SEL is asserted if the instruction received is a load, to enable the FPGA to drive the bus lines. The FSM also generates the signal SRC_LD and DST_LD to save the current address to the SOURCE and DEST registers as mentioned before. The **DS_SEL** signal that decides upon the second operand of the comparison on the next transaction is also generated during this cycle.

4.2 Assembly Code

In this section we present the assembly pseudo-code of the "repeated passing of arguments" algorithm, in order to demonstrate some low level implementation details. The assembly pseudo-code can be found in figure 10. Note that, between any two accesses to the shadow address space (that resides within the TurboChannel address range) we use the `mb` (memory barrier) instruction, which makes sure that all memory references prior to it complete before any further instructions are issued. Since the Alpha processor [10] (like many other processors) allows out of order execution, the sequence that shadow accesses reach the DMA boards may not be the same as the sequence they were issued.

The interested reader will also note the use of the `.set volatile` compiler directive, that is used to bypass default compiler optimizations. The "repeated passing" DMA initiation method makes repeated accesses to the same shadow addresses. To an optimizing compiler/assembler, without knowledge that the concerned addresses do not correspond to real memory, this seems a waste of instructions, and our assembler substituted instruction `5` by a register to register transfer, since it assumed that the value

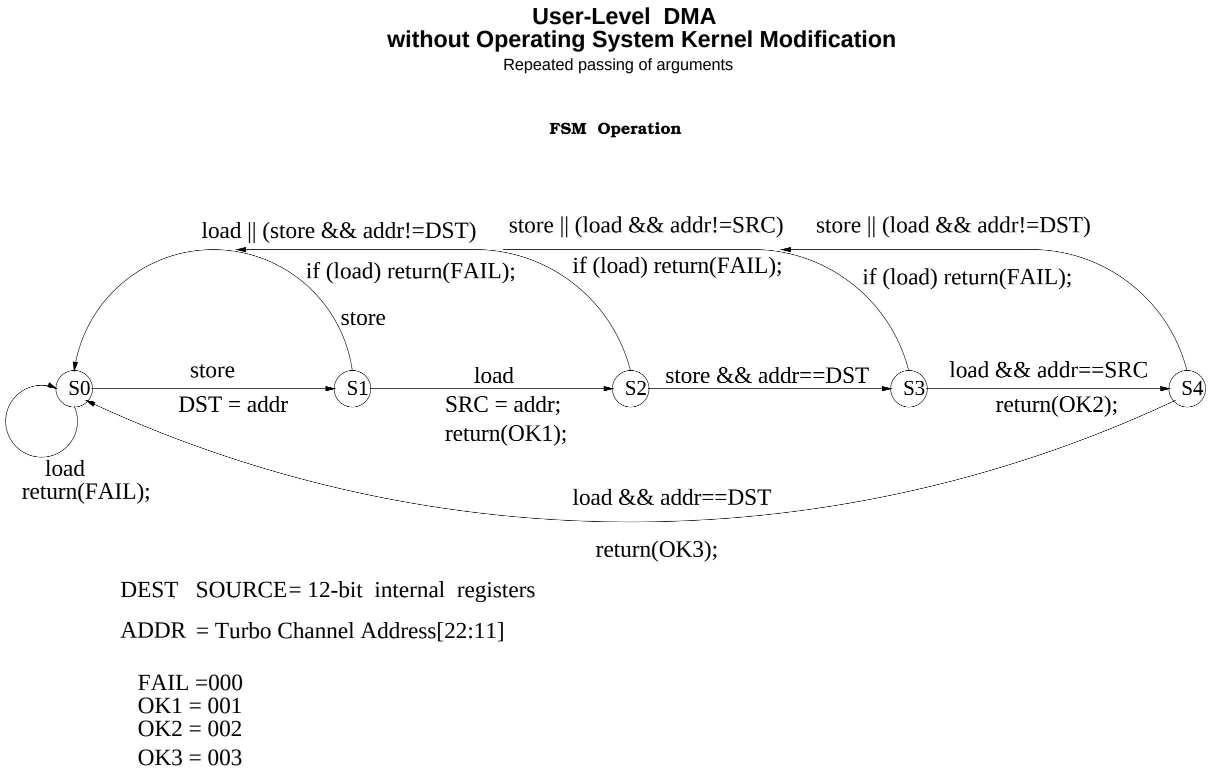

Figure 8: Finite State Machine for User-level DMA Prototype

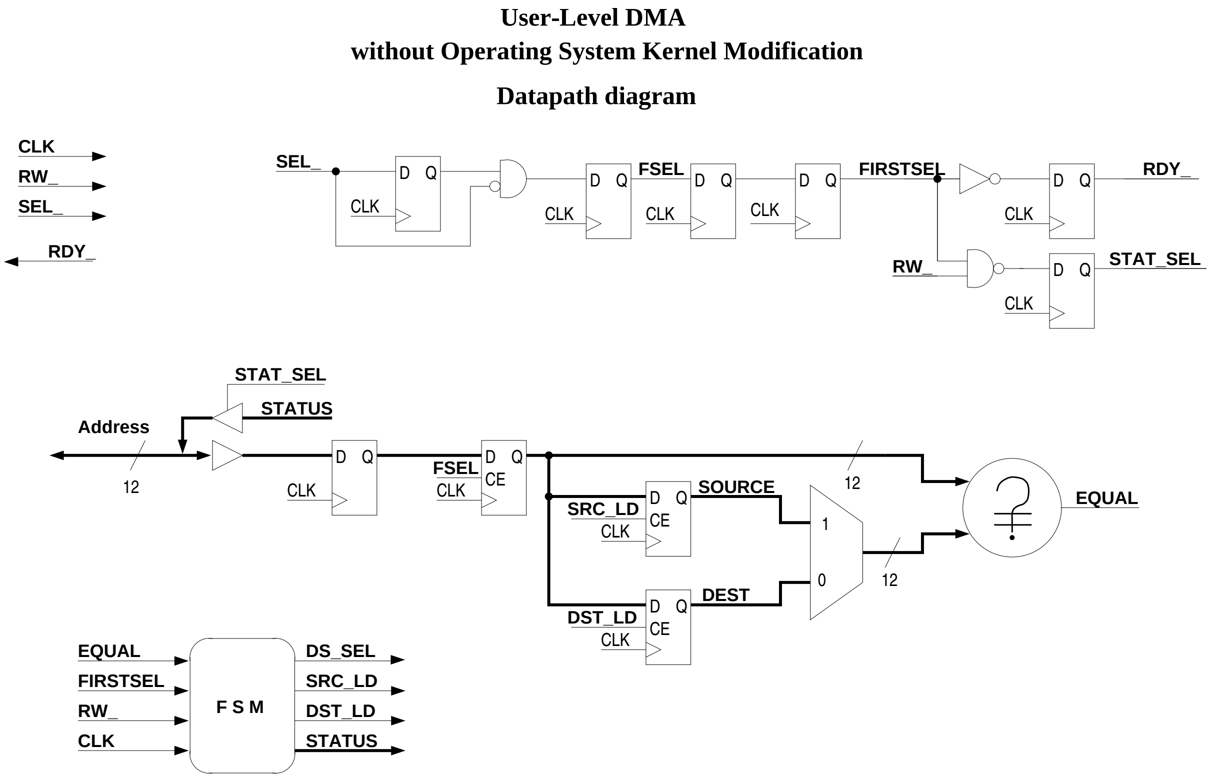

Figure 9: Datapath for User-level DMA Prototype

assigned by instruction 5 would be the value stored by instruction 3. We forced the assembler to keep the memory access stated in instruction 5 by using the `.set volatile` directive. [10]

```
DMA(vsource, vdestination, size)
        MB ; barrier
    1:  STORE size TO shadow(vdestination)
        MB ; barrier
    2:  LOAD return_status FROM shadow(vsource)
        If (return_status != DMA_OK1) goto 1:
    3:  STORE size TO shadow(vdestination)
        MB ; barrier
    4:  LOAD return_status FROM shadow(vsource)
        If (return_status != DMA_OK2) goto 1:
    .set volatile
    5:  LOAD return_status FROM shadow(vdestination)
    .set novolatile
        If (return_status != DMA_OK3) goto 1:
```

Figure 10: Assembly pseudo-code for user-level DMA by repeated passing of arguments.

4.3 Performance Evaluation

Our DMA prototype plugs in the Turbo Channel I/O bus of a DEC Alpha 3000/300 workstation, and runs at 12.5 MHz. We used our prototype to evaluate the performance benefits of user-level DMA. For each DMA method we perform a simple test of initiating 1,000 DMA operations. [11] Successive DMA operations were done to(from) different addresses, so as to eliminate any caching effects that intervening write buffers may induce. In the Repeated Passing of Arguments method, a memory barrier was used to make sure that repeated accesses to the same address were not collapsed in (or serviced by) the write buffer. Table 1 presents the (average) time it took for each algorithm to pass all arguments for starting a DMA operation.

We see that kernel level DMA costs close to 25 μs, which is a little more than the cost of an empty system call on this workstation. Fortunately, we see that all user-level DMA methods perform about an order of magnitude better than the kernel-based DMA. Best of all methods is the "Extended Shadow Addressing", which takes a little more than one microsecond. This is as expected, since this method needs only two assembly instructions to pass all DMA arguments to the network interface. The other user-level DMA methods take 4.4-4.6 microseconds, which is also expected since they use twice as many accesses to the network interface.

We should mention, however, that our implementation is pessimistic, and user-level DMA can achieve quite better performance in modern systems, that use faster buses. The TurboChannel bus that we used runs at 12.5 MHz, while recent buses, like the PCI bus run at frequencies as high as 66 MHz. We believe that faster buses will have a direct performance improvement on user-level DMA, while they will only marginally improve the performance of kernel-level DMA.

4.4 Verification

To verify the correctness of our prototype, we conducted the following experiment: We started several processes on the same processor. All processes initiate user-level DMA operations as fast as they can. Each process runs for one quantum (10 ms). After the quantum expiration, the process is suspended, put at the end of the ready queue, and the next process is scheduled for execution in a round-robin fashion. At every context switch operation (every 10 ms), we expect an error (unsuccessful DMA initiation) to

[10] As an aside note, we must refer that the assembler implemented this optimization although the option -O0 was used at compile time. By using this option the assembler must not implement any optimization.

[11] No DMA data transfer was actually performed. Only the DMA arguments were passed to the network interface.

DMA algorithm	DMA initiation (μs)
Kernel-level DMA	25.1
Ext. Shadow Addressing	1.5
Rep. Passing of Arguments	4.6
Key-based DMA	4.4

Table 1: Comparison of DMA initiation algorithms.

happen, since the DMA of the previously running process will be half-started. We expect the error to be reported to the currently running processes. Thus, we expect the DMA engine to report around 100 errors per second. We run the system with various configurations, consisting of two, three and four concurrently running processes. In all cases we measured the number of errors per second and found it to be between 112 and 115. This means that when a process is scheduled to run it may experience *two* unsuccessful DMAs instead of one. The reason is that the following interleaving may occur:

```
  PROCESS 1                      PROCESS  2
1: STORE size TO    shadow(A)
2: LOAD rs    FROM  shadow(B)
3: STORE size TO    shadow(A)
4: LOAD rs    FROM  shadow (B)
5:                              STORE size TO    shadow(C)
6:                              LOAD rs    FROM  shadow(D) <-DMA is rejected
7:                              lots of successful DMAs
8:                              STORE size TO    shadow(C)
9: LOAD rs    FROM  shadow (B)                             <-expects OK3 but gets OK1
10:STORE size TO    shadow(A)                              <-start all over
11:LOAD rs    FROM  shadow(B)                              <-expects OK1 gets FAIL
12: STORE size TO    shadow(A)                             <-start all over
```

In instruction 9 the first process wants to complete a DMA started in instruction 1, but the DMA engine "thinks" that its is the second instruction of the DMA started in 8, and returns OK1, while the process expects OK3, leading to an unsuccessful DMA initiation. Then, the first process starts the DMA from the beginning in 10, but gets a FAIL in 11, since the DMA engine "thinks" that the DMA operation asked consists of instructions 8-11, which do not comprise a valid DMA initiation prefix sequence. Thus, the process experiences a second unsuccessful DMA initiation. Finally, the first process starts a DMA from the beginning in instruction 12, and it will probably succeed from then on. Summarizing we, see that after some context switches, a process may experience two unsuccessful DMA initiations instead of one. To verify that this is the reason why we see more than 100 errors per second, we re-run the experiment, counting only the times the FPGA returns FAIL. In all our experiments, this happened 93-97 times per second, which is pretty close to what we expected. Note that it is not probable to measure 100 errors per second, since sometimes, a process may be suspended after it has completed a successful DMA initiation, but before issuing the first instruction of the next DMA operation.

Acknowledgments

This work is supported by the ESPRIT/OMI project "ARCHES" (ESPIRIT 20693), funded by the European Union. The major focus of the project is to accelerate the uptake of the high-speed HIC interconnect technology and IEEE 1355 standards in the marketplace. The HIC technology addresses in particular the marketplace for parallel systems interconnect and provides a major enabling technology for the Open Microprocessor systems Initiative. We deeply appreciate this financial support, without which this work would have not existed.

Czarek Dubnicki helped us shape some of the ideas described in the paper, and provided useful feedback for the rest of them. Bob Dobinson provided useful feedback during an early presentation of the ideas described in the paper. Shubu Mukherjee and Mark Hill pointed out the need for synchronization between cooperating processes that need to use user-level DMA. We thank all them.

References

[1] M. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. 21-th International Symposium on Comp. Arch.*, pages 142–153, Chicago, IL, April 1994.

[2] M.A. Blumrich, C.Dubnicki, E.W. Felten, and K. Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proc. of the 2nd International Symposium on High Performance Computer Architecture.*, pages 154–165, San Jose, CA, February 1996.

[3] Czarek Dubnicki. Personal Communication, 1996. Princeton University.

[4] J. Heinlein, K. Gharachorloo, S. Dresser, and A. Gupta. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proc. of the 6-th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1994.

[5] Dolphin ICS. PCI-SCI Cluster Adapter Specification. ARCHES project Working Paper No. 12.

[6] E. P. Markatos and M. G.H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proc. of the 2nd International Symposium on High Performance Computer Architecture.*, pages 144–153, Feb 1996. URL: http://www.csi.forth.gr/proj/arch-vlsi/papers/ 1996.HPCA96.Telegraphos.ps.gz.

[7] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proc. of the USENIX 1996 Technical Conference*, pages 279–294, San Diego, CA, January 1996.

[8] John Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the USENIX Summer '90 Technical Conference*, pages 247–256, June 1990.

[9] M. Rosenblum, E. Bugnion, S.A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proc. 15-th Symposium on Operating Systems Principles*, December 1995.

[10] R. Sites. Alpha AXP Architecture. *Communications of the ACM*, 36(2):33–44, February 1993.