User-Level DMA without Operating System Kernel Modification

Evangelos P. Markatos and Manolis G.H. Katevenis*
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

In Third International Symposium on High Performance Computer Architecture (HPCA-3)
San Antonio, TX, Feb. 1997†
URL: http://www.ics.forth.gr/proj/arch-vlsi/telegraphos.html

Abstract

Direct Memory Access (DMA) is frequently used to transfer data between the main memory of a host computer and the interconnection network, in order to free the host processor from the burden of the transfer. DMA operations are traditionally initiated by the operating system kernel, mainly to prevent one application from tampering with another applications' data. Recent architecture trends suggest that interconnection networks get faster, while operating systems get slower (compared to processor speeds). These trends imply that the initiation of a DMA operation becomes slower (due to operating system involvement), while the DMA data transfer itself becomes faster with time. Soon, the operating system overhead associated with starting a DMA will be larger than the data transfer itself, esp. for small data transfers.

This paper proposes several algorithms that allow user-level applications to start DMA operating without the involvement of the operating system. Our algorithms allow user applications to have direct (but controlled) access to the DMA engine registers. Low overhead user-level DMA is achieved without compromising protection, and without requiring changes to the underlying operating system kernel. Using our proposed algorithms, a DMA operation can be initiated in 2 to 5 assembly instructions. By comparison, operating system-based initiation of DMA requires thousands of assembly instructions.

*The authors are also with the University of Crete.

1 Introduction

Popular contemporary computing environments are comprised of powerful workstations connected via a network which, in many cases, has a high throughput, resulting in systems called *workstation clusters*, or Networks of Workstations (NOWs) [1]. The availability of such computing and communication power gives rise to new applications like multimedia, high performance scientific computing, real-time applications, engineering design and simulation, and so on. Up to recently, only high performance parallel processors and supercomputers were able to satisfy the computing requirements of these applications. Fortunately, the development of superscalar RISC processors increases the computing ability of modern workstations and microcomputers significantly. At the same time, recent improvement in high speed link technology has lead to the development of communication networks that sustain bandwidth in the order of Gigabits per second (Gbps). To allow fast processors to make efficient use of all the available bandwidth, several user-level memory-mapped network interfaces have been developed [2, 5, 9] and manufactured [7, 4, 14]. Most of these interfaces use Direct Memory Access (DMA) operations to transfer data from one workstation to another. DMA has been heavily used to transfer data between (fast) main memory and (slow) magnetic disks to free the host processor from the burden of transferring the data itself.

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, the 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 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 [2] and FLASH [8] 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 [3] (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.
- 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. [1]

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

[1] To our knowledge these are the first publicly available solutions to user-level DMA that do not require modifications of the underlying operating system kernel.

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.

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 [11, 12]. 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 buses 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 [10]. At the same time, we witness a impressive improvement in network throughput. ATM networks that provide 155 Mbps are common today, and will soon be upgraded to 622 Mbps. 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 [3, 8]. 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. 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)`.[3]

2.4 The first SHRIMP solution

Blumrich *et al.* described one of the first user-level DMA solutions [2], 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 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.[4]

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

[3] 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.

[4] Recall, no `destination` argument needs to be passed, because the destination for the DMA transfer, is the mapped-out

```
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

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.* [3] 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.

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.* [8] 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.

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 many 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 [13]. 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)
    /* 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.

```
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 [9].

3 User-level DMA without Kernel Modifications

3.1 User-level DMA based on keys

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 `vaddr`, 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

```
/* 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

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 guess the UNIX password of another user, rather than to guess a DMA key!

3.2 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. [5] 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 [6]: 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

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

```
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

two successive accesses to the same shadow address. Dubnicki's solution uses a three-instruction sequence as follows:

```
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. [6] 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.

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 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 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.

3.3.1 Proof of Correctness

In this section we will (intuitively) show that the "Repeated Passing of Arguments" method initiates a DMA operation correctly. A DMA operation would be initiated incorrectly, only if the user-level processes that attempt to start a DMA are interrupted and interleave their arguments. Suppose that process `P1` wants to start a DMA operation from memory location `A1` to memory location `A2`. Suppose also that several other processes interleave their instructions with `P1`'s. Although malicious processes may have read-only access to (possibly public) data `A1`, they do not have any access to private data `A2`. Assume, in the worst case, that all five instructions are issued by different processes as shown in figure 8(a). This interleaving implies that processes `P1`, `P3`, and `P5` at about the same time want to make DMA operations with the same destination (source). That is different processes want at the same time to write to (or read from) the same physical address. If processes `P1`, `P3`, and `P5` belong to different applications, then they should not be able to write-share the same physical memory lo-

[6] 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: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).

```
  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:).

cation, since different applications do not write-share physical memory - thus, such an interleaving can't happen. If P1, P3, and P5 belong to the same application, then there should be some synchronization operation before they all attempt to write(read) the same memory location. We assume that applications that want to use user-level DMA are well written, which implies that there is a synchronization operation between any conflicting accesses to the same memory location by different processes of the same application. This synchronization operation should serialize DMA transfers. Thus, in any successfully started user-level DMA, instructions 1:, 3:, and 5: must come from the same process, resulting in the interleaving shown in figure 8(b).

If all accesses to A1 were issued by process P1, that process has also issued two intervening LOAD instructions to address A2, as well. Thus, if all accesses to address A1 have reached the DMA engine, the accesses to address A2 issued by process P1, must have reached the DMA as well. Thus, if a DMA is started all five instructions must have been issued by the same process (see figure 8(c)).

Thus, in any successfully started DMA, all instructions come from the same process, and there is no way for a malicious user to tamper with the mechanism of starting a DMA.

3.4 Performance Evaluation

We have designed and implemented a prototype board to evaluate the performance of various DMA initiation algorithms. The board is plugged on the TurboChannel I/O bus of a DEC Alpha 3000 model 300 workstation. All the logic is contained in a single FPGA that is directly accessible from user applications via shadow addressing. The board runs at 12.5 MHz. For each DMA method we perform a simple test of initiating 1,000 DMA operations. [7] 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 start a DMA operation.

We see that kernel level DMA costs close to 19 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 2.3-2.6 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.

3.5 User-Level Atomic Operations

Recently, several network interfaces that provide a shared-memory abstraction on a Network of Workstations have been developed [9, 14]. To facili-

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

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

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

DMA algorithm	DMA initiation (μs)
Kernel-level DMA	18.6
Ext. Shadow Addressing	1.1
Rep. Passing of Arguments	2.6
Key-based DMA	2.3

Table 1: Comparison of DMA initiation algorithms.

tate shared-memory programming, these interfaces also provide atomic operations that allow different processes to protect their accesses to shared data. Such atomic operations include `atomic_add`, `fetch_and_store`, `compare_and_swap`, etc. All these operations need to pass one physical address to the network interface, one or more data arguments, and return back the result of the atomic operation. In most processors, at least two instructions are needed to pass all these arguments to the network interface securely. These instructions also need to execute atomically, otherwise a malicious user may disrupt the atomic operations done by legitimate users.

Initiating atomic operations from inside the operating system kernel (in order to achieve protection and atomicity) would result in significant overhead, since the operating system overhead would be much higher than the time it takes to do the atomic operation itself. Thus, atomic operations will benefit significantly if initiated from user-space. Fortunately, the user-level DMA methods that we described in the previous section can be easily adapted to initiate atomic operations from user-level. User-level initiation of atomic operations is a similar problem to user-level DMA, albeit somewhat simpler, since only one physical address is needed for each atomic operation.

4 Summary

In this paper we addressed the problem of user-level DMA, that is, starting a DMA operation from user-level without the help of the operating system kernel. Previous approaches to user-level DMA (although managing to launch DMA operations from user-level) require that the operating system kernel be modified to avoid race conditions from multiple users trying to start DMA operations at the same time. We believe that modifying the operating system kernel is a major obstacle in the widespread use of DMA, since most users are unwilling or unable to modify their underlying operating system kernel in any way.

In this paper we proposed several methods that achieve user-level DMA without any modifications to the operating system kernel. These methods vary in their simplicity and in their requirements of the host computer. Our "PAL Code" method is the simplest of all, but requires the existence of the DEC Alpha processor as the host processor. The "Extended Shadow Addressing" is simple as well, but it requires a large physical address space. The other two methods ("Key-based DMA", and "Repeated Passing of Arguments") although a bit more elaborate, function correctly in the general case, without any assumptions about the host computer. Using our proposed algorithms, a DMA operation can be initiated in only 2-5 assembly instructions all issued from user level.

We believe that our user-level DMA methods should be seriously considered for inclusion in high-speed network interfaces. Research prototypes have shown that the hardware cost of user-level DMA is low [3, 8, 9], while in this paper we show that the software cost of user-level DMA is also low, since it can be achieved without operating system kernel modifications.

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. George Kalokerinos, Thanos Oikonomou, and Gregory Maglis designed and implemented the prototype board described in the paper. George Milolidakis helped with the performance evaluation of the various DMA al-

```
Process P1    Process P2      Process P3    Process P4      Process5
STORE TO A1
              LOAD  FROM A2
                              STORE TO A1
                                             LOAD FROM A2
                                                             LOAD FROM A1
```

(a)

```
Process P1    Process P2      Process P3    Process P4      Process P5
STORE TO A1
              LOAD  FROM A2
STORE TO A1
                                             LOAD FROM A2
LOAD FROM A1
```

(b)

```
Process P1    Process P2      Process P3    Process P4      Process P5
STORE TO A1
LOAD  FROM A2
STORE TO A1
LOAD FROM A2
LOAD FROM A1
```

(c)

Figure 8: Possible interleaving of instructions in the *Repeated passing of argument DMA* approach.

ternatives. Bob Dobinson provided useful feedback during the development of the ideas described in the paper. We thank all them.

References

[1] T.E. Anderson, D.E. Culler, and D.A. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[2] 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.

[3] 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.

[4] N.J. Boden, D. Cohen, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29, February 1995.

[5] G. Buzzard, D. Jacobson, S. Marovich, and J. Wilkes. Hamlyn: a High-performance Network Interface, with Sender-Based Memory Management. In *Proceedings of the Hot Interconnects III Symposium*, August 1995.

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

[7] R. Gillett. Memory Channel Network for PCI. *IEEE Micro*, 16(1):12, February 1996.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] Dolphin Interconnect Solutions. Dolphin Breaks Cluster Latency Barrier with SCI Adapter, 1995. Press Announcement, http://www.dolphinics.no.