User-Level DMA Extension for NOW/Cluster Applications

Alexander P. Kemalov[1]

Abstract – **Direct Memory Access /DMA/ is previously used to transfer data between the main memory of host computer /PC/ and the network $\rightarrow$ to another one. This method is used to free the processor from the burden of transfer operations. DMA procedures commonly are initiated by the operating system kernel to separate one application and its data with another.**

A Network of Workstations /NOW/ architecture suggest that interconnections get faster and overhead and latency in networks go down while operating system operations get slower. In NOW or clusters these factors are very important because an intensive data transfers between hosts. These trends imply that DMA operation becomes slower /using operating system kernel/, compared to interconnection network.

This paper proposes several algorithms that allow applications to start DMA operation without OS kernel. The algorithms allow user-level applications to have direct access to the DMA engine. This approach is achieved without requiring changes to the OS kernel. Using our algorithms, DMA operation can be initiated faster /in comparison to OS kernel/.

Keywords – **DMA operation, memory allocation, networks, operating system kernel**

I. Introduction

Direct Memory Access /DMA/ is a common method for routing data directly between memory of host computer /PC/ and an input/output device /network controller/ without requiring enervation by the CPU. DMA management has been traditionally done by Operating System kernel, which provides protection, memory, buffer management, DMA registers and address translations. The overhead of this kernel initiated DMA transaction is hundreds of CPU instructions. There are two reasons for the necessity of the OS involvement in starting a DMA operation:

1. Atomicity – DMA operation start with transfer to DMA engine a source address, destination address and size of DMA packet. The process invokes OS to start an and schedule DMA operation and when finished, start another.

2. Protection from program errors – DMA engine works only with physical addresses, not allowed to access to user programs.

The user program use virtual address and must be translated to physical one. Virtual-to-physical address translation is performed by OS kernel. The physical memory pages used for DMA must be pinned to prevent the virtual memory system form paging them out while DMA data transfers are in progress.

[1]Al. Kemalov is with ICCS institute of BAS, Akad. G. Bonchev str. Bl.2 1113 Sofia Bulgaria, sasho@hsi.iccs.bas.bg

In the last years, high-speed LANs offers great performance and communication throughput and overheads of the OS involvement in DMA operation are still sensitive. For this reason, several researches have started to address the problem of letting user applications initiate DMA operation. Projects SHRIMP [1] and FLASH [2] have pinpointed the importance of user-level DMA. A disadvantage of these approaches is needs of modification of OS kernel.

A DMA procedure has three arguments: *source address, destination address* and *size* of packet. A DMA engine is responsible to perform above sources.

The OS translate the virtual source and destination addresses to their corresponding physical addresses and size to the DMA engine registers and start a DMA transfer. One of the common used techniques to secure virtual-to-physical address translation is the notion of *shadow addressing* [3]. For each virtual address **vaddr** correspond physical – **paddr** and **shadow(paddr)**. The shadow address is concatenating the physical one. The difference is shadow bit in address / for example 0x0FFFF $\rightarrow$ regular address; 0x1FFFF $\rightarrow$ its shadow address; range is within the physical address range; it is made in initialization time/.

An access to shadow address is interpreted by the DMA engine – virtual address **vaddr** is mapped to physical address **paddr** and virtual shadow address **shadow(vaddr)** - to **shadow(paddr)**. A transformation virtual-to-physical addresses use TLB (page-table) and is performed by memory controller. When a user application tray to pass the DMA engine, it will be treated as an argument passing operation and reject access to regular physical address. Thus it makes an access to virtual **shadow(vaddr)**. The DMA engine recognizes the shadow address and takes the physical address **paddr** by applying function shadow to physical address **shadow(paddr)**.

The mechanism of shadow addressing is fast and reliable, to pass physical addresses to a DMA engine from user memory space.

Another problem is to guarantied atomicity of a DMA operation. If there were a way to execute two instructions uninterrupted, then the problem will be solved. But from security point of view, it is dangerous because malicious users may be monopolizing an execution of programs – a decision is OS control.

II. First User-Level DMA Algorithm

The DMA engine is equipped with /4 to 8/ register contexts. Each context has a source, destination and size registers with their meaning. Each context is mapped into memory address space so that the processor can access it. Distinct context are

mapped into distinct memory pages so that each process gets access rights for only a single context. Each process can start user-level DMA operation /to write into single group context registers/. Thus if a process gets interrupted while starting a DMA, its arguments can't be mixed with another process's arguments. Each process has its own space in the DMA engine.

Unfortunately, user-level application can't use regular load/store operations to access these registers and load them with arguments of a DMA operation. Thus a process that would like to pass a physical address to a register context will pass context identification as a data argument of store operation, since the address argument of store has already been reserved to pass the shadow address:

STORE context_id **TO** shadow(vaddr)

The DMA engine extracts the **paddr** from **shadow(paddr)** and put it in register context **context_id**. To start a DMA, a process makes a sequence of above store operations. Unfortunately any process will be allowed to write an address argument into any register context. To prohibit this, we introduce a key that implies the user process is allowed to register context. Thus a physical address is passed to a DMA engine:

STORE key#context_id **TO** shadow(vaddr),

i.e. to proof key in OS and in an instruction is permitted to store a physical address as an argument in the register context. Using the above instruction the address arguments are securely passed to the DMA engine.

The last argument that must be passing is size of DMA packet. In this case we used regular store operation to the address that corresponds to the register context /size register/.

A user-level DMA operation is performed in fig. 1.

The last argument that must be passing is size of DMA packet. In this case we used regular store operation to the address that corresponds to the register context /size register/.

A user-level DMA operation is performed in fig.1.

We used store /not load/ instructions to load address arguments because a process that have both read and write access to the source address will be able to start user-level DMA operation from it. Most parallel and distributed applications use DMA procedures that have both read and write accesses to these data.

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

Fig. 1.

III.   Second DMA Algorithm

The algorithm proposed above, achieves user-level DMA operation without OS kernel modification. But theoretically may be broke from a lucky user, who manages to guess another user's key. To avoid this one, we make the identification of the process part of the shadow address.

We introduce some bits of the physical address that will be passed as an argument to the DMA engine corresponds to the process identification. These bits are set by the OS when it creates the mappings from shadow virtual addresses $\rightarrow$ to shadow physical addresses. Part of the shadow physical address is now the context_id /2 bits/, i.e. 4 processes will be able to start user-level DMA operation from the same processor /fig. 2/.

```
DMA(vsource, vdestination, size)
   /* pass physical address shadow(vdestin.) and size to the DMA engine */
   STORE size TO  shadow(vdestination);
   /* pas s physical addr shadow(psouce) to the DMA engine and read if successful */
   LOAD return_status FROM shadow(vsouce)
```

Fig. 2.

By checking the context_id , the DMA engine knows which process the shadow address belongs to. The DMA engine has several register contexts to save these addresses, receives in the appropriate contexts and start the DMA operation when all arguments are available. If there are no register contexts and DMA engine receives pairs of STORE and LOAD instructions, it checks for the context_id value of the two physical addresses. If they are different, DMA is not started and an error is returned by the last LOAD instruction.

IV.   Third Algorithm

In the last algorithm we tray to start user-level DMA operation without the need extra bits in the physical address /context_id /. If a process passes at least one shadow address more than once, the DMA engine may be able to determine if the user process was interrupted. The proof is checking the two successive accesses to the same shadow addresses. The DMA engine initiates a DMA operation only if it sees a sequence of the form LOAD, STORE, LOAD and arguments of the two load instructions are the same. If the process is interrupted while trying to start a DMA, then the DMA engine will receive a non valid sequence of shadow addresses, and DMA is not start.

The above sequence may lead to error data transfer, if abused by malicious user – a possibility of interleave of shadow address.

We introduce additional instruction to protect this situation:

DMA(vsouce, vdestination,size)
  STORE size **TO** shadow(vdestination)
  LOAD return_stat.1 **FROM** shadow(vsouce)
  STORE size **TO** shadow(vdestination)
  LOAD return_stat.2 **FROM** shadow(vsource)
  LOAD return_stat.2 **FROM** shadow(vsource)

If a malicious user does not have access to addresses vsouce, vdestination, above sequence will work correctly. To provide this and avoid interleaving, we include additional instruction: /Fig. 3/

```
1: STORE size TO shadow(vdestination)
2: LOAD return_stat.1 FROM shadow(vsouce)
   If (return_stat.1==FAILURE) go to 1:
3: STORE size TO shadow(vdestination)
4: LOAD return_stat.2 FROM shadow(vsource)
   If (return_stat.2==FAILURE) go to 1:
5: LOAD return_stat.3 FROM shadow(vdestin.)
   If (return_stat.2==FAILURE) go to 1:
```

Fig. 3.

The shadow(vsource) address pass twice to the DMA engine, while shadow(vdestination) address – three times. The DMA engine is prepare to receive 5 instruction sequence to shadow address space and a DMA operation start only if there are sequence STORE, LOAD, STORE, LOAD, LOAD and the address arguments in instructions 1,2,5 and 2,4 are the same.

V.   Proof of Correctness

We proof above algorithms with a testbed including a pair Pentium III workstations, running MSC.Linux OS rev. nov.02 /special version for cluster and distributed applications/. The test applications consists a server and client /ping-pong/ messages with acknowledgments and different size of packets.

A DMA operation would be initiated incorrectly if a user process attempt to start a DMA, are interrupted and interleave their address arguments. Suppose that process P1 want to start DMA from A1 memory location → to A2. Suppose that there are several other processes P2...Pn interleave their instructions with P1. Although other processes may have read only access to A1, they do not have access to A2. Assume that all P2Pn execute subroutine in fig. 3 and want to write/read the same physical address. If processes P2...Pn belong to different applications, then they should not be able to write-share the same physical memory location, since different applications do not write-share physical memory. Thus such an interleaving can't happen.

If P2...Pn belong to the same application, then there should be some synchronization operations be include before they all attempt to write/read the same memory location. This synchronization should serialize DMA operations.

If all access to A1 were issued to P1, that process has also issued two interleavening LOAD instructions to A2 as well. Thus all trying to access to A2 is reached from DMA engine. If a DMA started all five instructions must have been issued by the same process /P1-successfully started DMA/.

VI.   Conclusion

UDMA allows user process to initiate DMA procedure to or from I/O nodes at a cost of only two user-level memory reference and additional instructions. These extremely low overheads with using of UDMA.

The UDMA procedure does not require much additional hardware because it takes advantage of both hardware and software in the existing virtual memory system. This is very important in a process of an implementation of cluster architectures in practice.

In the future we tray to implement these procedures in I/O operations in a cluster architecture and GRID middleware. This task transform DMA procedure in remote I/O paradigm in which applications use familiar parallel I/O interfaces to access remote file systems.

References

[1] M. Blumrich, R. Alpert, Y. Chen, D. Clark, C. Dubnicki, Design Choices in the SHRIMP system: An Empirical study. In *Proc. Of 25th Intern'l Symp. On Computer Architecture*, 1998

[2] J. Heinlein, K. Gharachorloo, S. Dresser, A. Gupta, Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. In *Proc. of 6th Intern'l Conf. on Architectural Support for Progr. Languages and OS*, 1994

[3] C. Dubnicki, A. Bilas, Y. Chen, K. Li, VMMC-2: Efficient Support for Realible Connection Orienteted Communication. In *Proc. of Hot Interconnects*, 1997

[4] R. Dimitrov, A. Skjellum, An Efficient MPI Implementation for Virtual Interface (VI) Architecture - Enabled Cluster Computing. http://www.mpi-softtech.com

[5] Intel Corp. Intel Virtual Interface Architecture - Developer's Guide http://developer.intel.com/design/servers/vi/developer

[6] M. Buchanan, A. Chien, Coordinated Thread Scheduling for Workstation Clusters under Windows NT. In *Proc. of USENIX Windows NT Workshop*, 1997

[7] NERSC PC Cluster Project at Lawrence Berkeley Nat'l Laboratory M - VIA: A High Performance Module VIA for Linux; http://www.nersc.gov /research/FTG/via

[8] St. Muir, J. Swift, Functional divisions in the Piglet multiprocessor operating system, In *ACM SIGOPS European Workshop*, 1998

[9] K. Schwan, R. West, M. Rosu, A Network Co-processor based Approach to Scalable Media Streaming in Servers. In *Intern'l Conf. on Parallel Processing*, 2000

[10] G. Banga, J. Mogul, Scalable kernel performance for Internet servers under realistic loads. In *USENIX Technical Conference*, 1998