Algorithms for DMA communications

Alexander P. Kemalov

Abstract: *A Direct Memory Access /DMA/ is previously used to transfer data between the main memory of host computer /PC/ and the network → to another one PC. 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 cluster of PCs architecture suggests that interconnections get faster and overhead and latency in networks go down while operating system operations get slower. In clusters these factors are very important because an intensive data transfer between hosts. These trends imply that DMA operation becomes slower /using operating system kernel/, compared to interconnection network. The important aspect here is ability to transfer data directly between the network interface and application buffers. Such direct data path requires the network interface "to know" the virtual-to-physical address translation of a user buffer.

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 and TLB library. 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/.

Key words: *DMA operation, memory allocation, TLB library, networks, operating system kernel*

INTRODUCTION

A Direct Memory Access /DMA/ is a common method for transport data directly between memory of host computer /PC/ and an input/output device /network controller/ without requiring enervation by the CPU. A 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 a 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 a DMA engine a source address, destination address and size of a DMA packet. The process invokes OS to start and schedule a DMA operation and when finished, start another.
2. Protection from program errors – a 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.

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 translates the virtual source and destination addresses to their corresponding physical addresses and size to the DMA engine registers and starts a DMA transfer. One of the common used techniques to secure virtual-to-physical address translation is 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 → regular address; 0x1FFFF → 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 a memory controller. When a user application tries 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.

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 contexts 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 puts it into 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 passed is the 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 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 destination 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

We used store / not load / instructions to load address arguments because a process that has 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.

SECOND DMA ALGORITHM

The algorithm proposed above achieves user-level DMA operation without OS kernel modification. But theoretically it may be broken by a lucky user, who manages to guess another user's key. To avoid that, 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 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);
 /* pass 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.

THIRD ALGORITHM

In the last algorithm we tray to start user-level DMA operation without the need of 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 addreses vsouce, vdestination, the above sequence will work correctly. To provide this and avoid interleaving, we include additional instruction:/Fig3/

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 passes twice to the DMA engine, while shadow(vdestination) address – three times. The DMA engine is prepared 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.

DESIGN OF UTLB

A direct data path requires the network interface and DMA engine "to know" a virtual-to-physical address translation TLB of a user buffer - User-managed TLB /UTLB/ which eliminates system calls and device interrupts from the common communication path. The UTLB only invokes the OS for pinning the user buffer when it is first used in communication. The mechanism does not rely on OS modifications - only a device driver that access the OS pinning and unpinning facility is required. The main ideas are demand-driven page-pinning, protect translation table, user-level lookup:

a/ demand-driven page-pinning – pin the local buffer when it is used in communication for the first time and supplying the address translations to the network interface. The buffer remains pinned in the physical memory so that subsequent data transfers using this buffer can be initiated directly of the user level.

b/ protected translation table – UTLB allocates for each process. A translation table contains physical address for process virtual pages that have been pinned in the physical memory. The translation table is invisible to the user process but it can specify where in the table to store the physical translations for given virtual buffer. To transfer data on a virtual page, the user process specifies to the network interface the index in the translation table where the pages physical address stored; using the index, the network interface reads the physical address directly from the translation table.

c/ user-level lookup – the user process has to keep track of the mapping between the translation table indices and the pinned virtual pages. The lookup table uses a standard two-level page table architecture. It contains one entry for each virtual page. An entry can either be invalid or contain the index where the physical address for virtual page is stored.

Only two memory references are required to obtain the UTLB index for given virtual page address.

Combining the above tree ideas results in the proposed UTLB. The communication subsystem allocates a fixed-sized translation table for each started process and is allocated directly in the network interface memory. They are protected from user process. The user process asks OS to pin certain virtual pages and install their physical translations at defined locations in its translation table. This can be done with a device driver call. A user-level library maintains the mapping between the translation table indices and the virtual page address in the two-level lookup tree. The user-level library can detect capacity misses and evict some translations already in the UTLB. A eviction results in unpinning of the virtual pages. The UTLB decides which translation table entries to evict and ask OS to unpin corresponding virtual pages and invalidate the entries. To ensure correctness, UTLB must only select virtual pages that will not be involved in any outstanding send requests.

When a miss occurs the network interface simply reads the entry from the translation table in physical memory. Misses fall into three categories: capacity misses, conflict misses and compulsory misses[11].When only one process is started, both capacity misses and conflict misses may occurs. Capacity misses can be reduced by enlarging the size of UTLB. Conflict misses can be reduced by making UTLB a set-associative. The set-associativity can also reduced conflict misses that are coused by multiprogramming. A simple scheme to reduce the conflict misses is to offset a translation table index by process-dependent constant. The same index from different location tables will be hashed into different locations in UTLB.

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 P2…Pn 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 accesses 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/.

SUMMARY

In this paper we presented the problem of DMA transfer in I/O operations. Initiating atomic operations inside OS kernel / for protection and atomicity/ [4,8] would result in significant overheads. Starting DMA operation without the help of OS kernel results in freeing the processor and OS and down overheads in I/O. Fortunately, our user-level DMA methods can be easily adapted to initiate DMA operation from user application without modification of OS kernel. This is very important in process of an implementation of cluster architectures in practice /massively I/O operations/.

We proposed three algorithms that achieve user-level DMA without any modifications to OS.

The results show that the UTLB approach has fewer misses including both user-level check misses and network interface misses than interrupt-based approach. The UTLB can detect most translation misses at user level to avoid interrupts, whereas the interrupt-based approach requires an interrupt on every translation miss. The UTLB is less sensitive to the translation table sizes than interrupt-based approach.

The UDMA mechanism does not require much additional hardware because it takes advantage to the both hardware and software in the existing virtual memory system.

In the future we tray to implement these procedures in I/O operations in a GRID middleware – remote memory access via LAN and WAN.

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

[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

[11]. M. Hill Aspects of Cache Memory and Instruction Buffer Performance University of Berkeley, Tech. Report

http://sunsite.berkeley.edu/TechRepPages/CSD-87-381