FEEDING HUNGRY PROCESSORS: REAL-TIME I/O DEMANDS OF HIGH-PERFORMANCE MULTIPROCESSING COMPUTERS

Bruce H. Johnson
Silicon Graphics Computer Systems
Houston, Texas

ABSTRACT

It has been documented that microprocessor performance doubles about every 21 months. Much is published and reported on the technology that delivers this impressive computing power. Much less is said, however, about the unique Input/Output (I/O) demands that are presented when using these microprocessors in high-performance, real-time, multiprocessing environments. Raw computing power is seldom questioned anymore. Of more concern today is the ability of a computer system to deliver data to and from these high-performance processors.

For example, it is not difficult to select a computing engine that is capable of performing the computations necessary to drive a Full Flight Simulator (FFS) or a Weapons Systems Trainer (WST). It is, however, a significantly greater challenge to determine how the simulation I/O can be performed so as to eliminate bottlenecks and latencies. The training value of a simulator can easily be lowered by the stepping or jumping of an instrument, visual system, or motion base that is due to the inability of the I/O to keep up with the processors.

This paper will explore some of the technology available that can be used to "feed" today's high-performance, real-time, multiprocessing systems. Both advances in hardware and software will be discussed, advances that give developers the tools they need to deliver I/O to and from a simulator with determinism and realism.

ABOUT THE AUTHOR

Mr. Bruce Johnson is a Systems Engineer for Silicon Graphics Computer Systems and consults to the real-time, simulation industry. He has spent the last twelve years working for both simulation/training contractors and computer system manufacturers. He holds a Bachelor of Science Degree in Engineering and Computer Science from the University of Florida.

FEEDING HUNGRY PROCESSORS: REAL-TIME I/O DEMANDS OF HIGH-PERFORMANCE MULTIPROCESSING COMPUTERS

Bruce H. Johnson
Silicon Graphics Computer Systems
Houston Texas

INTRODUCTION

There is no doubt that I/O issues are important to the simulation community. Virtually all simulation systems have some sort of specific I/O requirements that must be met. In some cases, the industry has even defined specific I/O standards for simulators. For example, the Federal Aviation Authority (FAA) requires that FAA certified simulators have a transport delay of less than 150 milliseconds (FAA Advisory Circular AC 120-40B). Though not as easily quantified as transport delay, nearly every simulator specification requires that all components be free of jumping and stepping. A simulator does not have to be directly involved with pilot training in order to have specific I/O requirements. There are also some very specific transport delay requirements identified in the Communication Architecture for Distributed Interactive Simulation (CADIS) document.

How data is moved between the simulator and the CPU (or CPUs) can make a big difference in the amount of effort it will take to meet these types of requirements. While the heart of a real-time, host computer system may still be the CPU (or CPUs), the I/O interconnect and how well it is utilized will often determine the level of realism and fidelity of a given simulation.

The computer systems of today differ dramatically from those offered only a few years ago. Along with the changes in features and capabilities, there comes a need to change the way that much of the system design is being done. This is particularly true in terms of designing and optimizing the flow of data from a simulator, to the CPUs, and then back again. Practices that provided efficient use of resources a few years ago could potentially "bottleneck" today's high-performance systems that rely heavily upon data buffering and pipelining.

THE FIRST STEP: PASSING DATA TO/FROM MEMORY

When considering the I/O of a high-performance computer system, the first area to explore is the path from processors to memory. Certainly not unique to the real-time industry, all computing applications are concerned with providing the widest, shortest path from CPUs to memory. Memory subsystems of modern computers are generally made up of Dynamic Random Access Memory (DRAM). While it was stated previously that CPU performance has been doubling every 21 months, DRAM access times have decreased only by a factor of about one third over the past decade. This difference in performance change is shown in Figure 1.

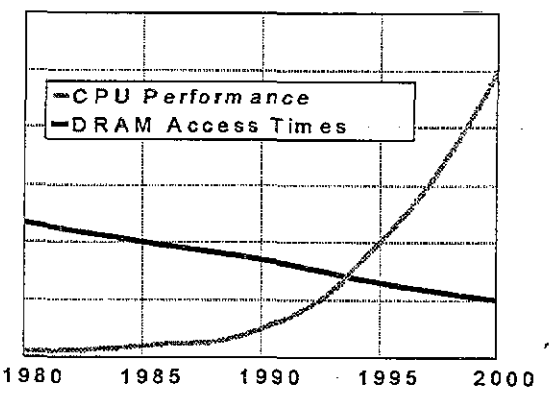

Figure 1

Amdahl's law says that the performance of a computer system as a whole will increase as a result of better performance in a component only to the extent that the improved component may be used. For example, if the speed of a CPU is dramatically increased, but the memory speeds are kept relatively constant, the CPU will spend increasing numbers of clock cycles waiting for memory to respond. Computer designers must clearly provide some techniques that permit faster paths to memory in order to keep the CPUs of the system well fed.

One solution to this problem is in the use of higher speed Static Random Access Memory (SRAM). Ideally, to maximize the bandwidth between CPU and memory (and also provide more deterministic memory access times), all memory subsystems would consist entirely of SRAM memory. Indeed some computer systems offer (or have offered) large SRAM memory subsystems as key components of their design (e.g., Encore RSX and CRAY C-90). The downside to SRAM is in its additional cost when compared to that of DRAM. As can be seen from the following equations, in order to equip an average size simulator host computer with SRAM as opposed to DRAM would increase its price approximately 100,000 dollars.

200$ / MegaByte (MB) * 64MB DRAM = $.12,800
2000$ / MegaByte (MB) * 64MB SRAM = $128,000

Realtime developers and engineers demand high-performance and determinism, but delivering a system with entirely SRAM memory is certainly cost prohibitive to most.

Overview of Caching

A more cost effective solution to the problem lies in the use of caches. Caches are SRAM-based subsets of the lower cost DRAM that are provided on a system so that the CPUs have faster access to data. Caches are generally implemented with either a "write-through" or "write-back" strategy. Write-through caches will write modifications to their memory immediately back to main memory. Write-back caches will wait to write modified memory back to main memory until the cache line is needed to hold other data. For almost all applications, write back caches provide better performance by reducing the amount of traffic between the cache and main memory.

When cache data is read in, it is normally done using a "cache line" of data. Reading data in cache lines takes advantage of the fact that the next piece of data that a processor wants is very likely to be near the one that was just accessed. This is why, when designing I/O transfers, an engineer should always think in terms of long bursts of data (i.e., a cache line size) since it is likely that this is how data is being moved internally through the system.

All popular RISC microprocessors today incorporate small on-chip caches that generally range from 8 - 128 KB in size. These on-chip caches, or primary caches, are most often supplemented with larger secondary caches that reside external to the CPU itself. The size of secondary caches varies even more, secondary caches exist that are as large as 4MB per CPU. To emphasize the importance of caches in application performance, Table 1 is presented to show access cycles of caches versus that of main memory (numbers are representative only and do not necessarily reflect any particular computer system):

read cache hit primary cache	1 cycle
read cache hit secondary cache	4-11 cycles
read miss (access main memory)	100-160 cycles

Table 1

Caching and Multiprocessing

The use of cache in computer systems is not a new design concept, but integrating of cache with multiprocessor computer systems is. The challenge for a multiprocessing system when using caches is to ensure that each CPU has a consistent view of the system memory. When using

caches, it is certainly not uncommon for the same data to be kept in multiple locations and this data must be efficiently synchronized and exchanged. A typical solution to this problem is through the addition of bus snooping to the system architecture. Bus snooping means that each interface to the system bus (i.e., CPU cards, memory cards, I/O interfaces, etc.) monitors, or snoops, the bus traffic. Upon a read request, the interface with the latest copy of the data responds to the read. The techniques implemented by multiprocessing systems to optimize cache coherency can make a big difference in overall system performance.

Some multiprocessing systems implement a duplicate set of cache "tags" in order to minimize contention between processors and the bus snooping mechanism. Cache tags are used in a system to identify the addresses for the data that reside in cache. They are used by both the CPUs and bus snooping mechanism in a multiprocessor system. The CPUs use cache tags to index into the cache and pull data when there is a cache hit. The bus snooping mechanism utilizes cache tags when monitoring the addresses on the bus. When snooping and CPU access to cache occur simultaneously, CPU cycles can be lost due to contention for tags. Duplicate cache tags can greatly reduce the contention between CPU and the bus snooping mechanism.

Another cache coherency technique used in some architectures is the three-party transaction. A three-party transaction is implemented as follows. Whenever a read request is satisfied by data from a second processor's cache, the main memory interface "accepts" the read as if it were a write. For example, if processor A accesses memory for which the only valid copy is contained in processor B's cache, the block of memory will be simultaneously written back to main memory and transferred to processor A's cache.

Memory Interleaving

Another technique used for maximizing the bandwidth to memory is memory interleaving. Memory interleaving is a way of organizing the DRAM memory of a system into "leaves" that are capable of independently processing a memory request for a processor. Each memory leaf can be thought of as an independent path from CPUs to memory. By interleaving memory, the negative affect of DRAM access latency (when compared to CPU clock rate) is hidden by overlapped memory accesses in multiple memory leaves.

Memory interleaving is especially advantageous in multiprocessor systems. Most operating systems ensure proper distribution of sequential memory accesses by a single process. However, this is normally not done for multiple processes executing on multiple processors. Therefore, to minimize the probability of two successive memory accesses to the same leaf, memory interleaving is provided. As the number of processors increases, the number of memory leaves should have the capability of being increased as well.

THE NEXT STEP: PASSING DATA TO/FROM THE I/O BUSES

As with advances in the access times of memory, growth in the performance of I/O buses has not kept up with growth in CPU performance. Figure 2 and Table 2 compare the growth in the performance of industry standard I/O buses with that of RISC microprocessor performance (years indicates years of use, not introduction).

Years	Popular I/O Bus	Bandwidth
< 1980	MultiBus-1	10 MB/sec
1980s	SELbus	26.67 MB/sec
1990s	VMEbus	40-65 MB/sec
2000s	Futurebus+	100 MB/sec

Table 2

Table 2

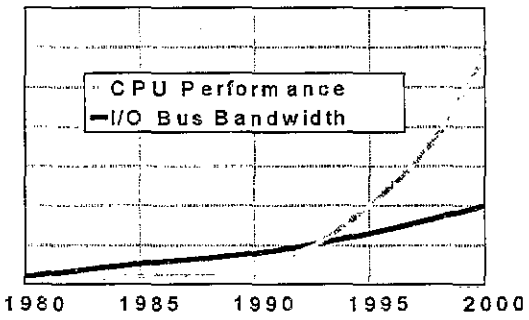

Figure 2

Not only has I/O bus performance growth lagged CPU performance growth, but the data path from CPU to I/O bus in today's systems is generally longer than what it used to be. Data often must be scaled from the large internal buses of RISC processors to narrower I/O buses such as SCSI and VME.

Figure 3 is a simplified example of how the buses of a high-performance computer system are often scaled down from larger system buses to smaller I/O buses.

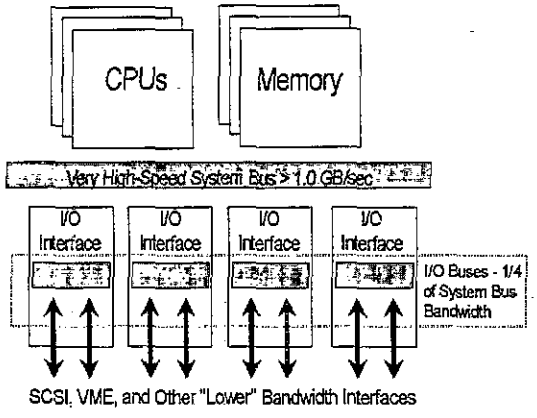

Figure 3

In a similar fashion, the bus bandwidth of the system bus is generally much higher than that of I/O and peripheral buses. This presents system designers with the unique challenge of trying to maximize the throughput of a lower performance bus (i.e., SCSI or VME), while minimizing the negative effect that the slower data transfers will have on the high-speed system bus. Some features that are used to accomplish this are I/O caches, data buffering, and the pipelining of bus grant and requests on the I/O bus.

Maximizing the Performance of a VMEbus

Whether incorporating a single board interface or a complete I/O subsystem, the VMEbus has become a de-facto standard in the simulation and training industry. While most real-time computer systems are designed with an integral VMEbus, the performance features and integration capabilities of the bus can often vary greatly between manufacturers.

In theory, boards that comply with the IEEE-1014 VMEbus Specification should work together on the same bus. In practice, however, every design engineer knows that VME board integration can be a difficult task. At least part of the problem can be attributed to the format of the VMEbus Specification as it leaves room for interpretation in many areas. As a result, boards can be functionally correct yet employ marginal design practices that are exposed with increased bus activity and contention. In addition, a myriad of software issues can increase the complexity of VME board integration as well.

VME board selection is crucial to optimizing the bandwidth of the VMEbus. The bandwidth of the VMEbus in a given application will most likely be determined by the boards on the bus, not the host computer controlling the bus. Boards that support block transfers, Direct Memory Access (DMA) transfers, and 64-bit data transfers are going to be able to pass data at much faster rates than those that support only Programmed I/O (PIO). Table 3 shows an example of the differences in throughput that can be expected between data transferred via PIO versus that of DMA (while data is representative in nature it is based on actual testing results).

Transfer Type	PIO Bandwidth	DMA Bandwidth
8-bit(D8) Read	.5 MB/sec	3.0MB/sec
8-bit(D8) Write	.75MB/sec	3.75MB/sec
32-bit(D32) Read	2.0MB/sec	12MB/sec
32-bit(D32) Write	3.0MB/sec	15MB/sec

Table 3

Some of the Features Available for VME I/O

One feature that several real-time computer systems are offering is the capability to support multiple VME buses from a single computer. In fact, systems are available today that offer up to five separate VME buses per system. Not only does this dramatically increase the I/O bandwidth of a system but it also adds a great deal of flexibility to the I/O design. For example, it is often a wise practice to isolate slower VME boards (i.e., those that are accessed via programmed I/O) from boards that are performing faster DMA block transfers.

Many vendors now also provide support for 64 bit data transfers on the VMEbus per Revision D of the VME Specification. 64-bit transfers can effectively double the throughput capabilities of a VMEbus. In order to implement 64-bit transfers, both the host and the target (VME board) must provide D64 support. It appears that more and more VME board manufacturers are adding this capability to their products.

For VME boards that do not have DMA capabilities, some host computer vendors provide a DMA master capability on their VMEbus interfaces. Commonly referred to as a DMA engine, this DMA master capability enables boards that lack support for DMA to increase their performance by using the DMA capabilities of the VME interface. Using the DMA engine facility, applications have been shown to deliver higher performance even when using transfer sizes as small as 128 bytes. Figure 4 shows relative performance differences between I/O transfers using PIO compared with those of a typical DMA engine.

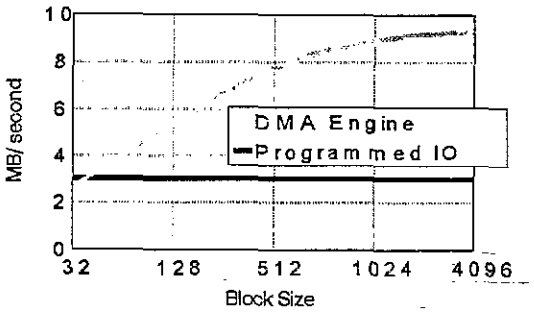

Figure 4

Perhaps unique to real-time systems is a feature that gives the capability to route VMEbus interrupt levels to individual CPUs. In this way, real-time tasks can be isolated to execute in response to interrupts mapped to a specific CPU - thereby reducing interrupt latencies incurred by the processing of non-realtime interrupts on the CPU. Additionally, for systems that operate under a UNIX-based OS, it is virtually imperative that the clock scheduling interrupt be removed from an isolated processor to ensure determinism in responding to high priority interrupts.

Calculating Available Bus Bandwidth

A very important metric in determining the I/O capabilities of a computer system is the available bus bandwidth. Available bus bandwidth refers to the amount of data that an I/O backplane can transfer at any one time. Often vendors quote an I/O bus bandwidth that reflects only the performance of a single type of operation - invariably, the fastest. In almost all simulation applications, however, the I/O bus incorporates several different types and sizes of transfers on the bus.

Available bus bandwidth varies greatly with each application and is dependent upon both the host and the I/O boards. The following example demonstrates this. A given simulation application has the

following I/O requirements every frame of its simulation:

5 KB	read from reflective memory
30 KB	written to reflective memory
100 KB	read from DMA interface device
100 KB	write to VME-based SCSI disk for record/playback

For realistic data presentation the I/O is performed at 60 Hertz (Hz):

5 KB * 60 =	0.3 MB
30 KB * 60 =	1.8 MB
100 KB * 60 =	6.0 MB
100 KB * 60 =	6.0 MB
Total =	**14.1 MB per second**

Only 14.1 MB/sec need to be sustained, which is well within the vendor quoted bus bandwidth of 25 MB/second. Yet taking a closer look at the type of I/O that is actually being performed reveals a real I/O bottleneck. For purposes of this discussion, assume the computer system is capable of delivering the following bandwidths with various operations on the VMEbus:

D32 PIO Non-block write transfers	4 MB/sec
D32 PIO Non-block write transfers	1 MB/sec
D32 DMA Block read transfers	30 MB/sec
D64 DMA Block write transfers	55 MB/sec

Applying these numbers to our previously generated throughput requirements yields a problem:

1.8 MB @ 4 MB/sec =	45%
0.3 MB @ 1 MB/sec =	30%
6.0 MB @ 30 MB/sec =	20%
6.0 MB @ 55 MB/sec =	11%
	106% utilization

A solution to this engineer's problem could lie in the availability of a VME DMA master capability on the host computer which could be used with VME cards that support only PIO. If provided, this DMA "engine" should be able to deliver the following bandwidth for writes to the VMEbus:

DMA Engine D32 Nonblock writes	9 MB/sec

Plugging this into the equations, we now have a different result:

1.8 MB @ 9 MB/sec =	20%
0.3 MB @ 1 MB/sec =	30%
6.0 MB @ 30 MB/sec =	20%
6.0 MB @ 55 MB/sec =	22%
	92% utilization

These numbers are given only as an example, yet they underscore the *importance of understanding the various* factors that influence available bus bandwidth.

Using SCSI for Realtime

SCSI controllers are inexpensive DMA devices that are generally optimized to make maximum use of a system's resources. When using 16-bit SCSI, useful bandwidths of up to 14 MB/sec can be achieved on each SCSI bus. Factoring this into an average price for bus expansion and comparing it to VME yields the insightful results of Table 4.

Device to Add	Price	Bandwidth	$/MB
SCSI bus	< $1000	14 MB/sec	71
VMEbus	> $ 10000	50 MB/sec	200

Table 4

While SCSI may still present a non-traditional approach to interfacing and controlling real-time hardware, its price/performance advantage certainly cannot be ignored. Already, commercial interface manufacturers are realizing the advantages of SCSI by introducing SCSI-based I/O devices such as terminal servers. Real-time *SCSI interfaces should be considered as a* cost effective solution for virtually any future I/O requirement.

MULTIPROCESSING AND I/O

While very powerful simulators and simulation systems are being developed today that are driven by single-processor computer systems, multiprocessor systems are becoming more and more the norm. A few of the reasons why multiprocessor

systems are advantageous for simulation applications are identified in the following paragraphs:

- Many simulators and simulated systems have increased in complexity of late and require additional system resources to process the load. Development of simulators for such devices as the Air Force's F-22 and NASA's Space Station Freedom are reaching new limits in processing power demands.

- Recent trends show an increase in the incorporation of special purpose processing at the host computer level. Image generation graphics, motion-base flight models, control loading forces, DIS packet processing, etc., are now all capable of being processed by host computer processors. This design permits better data localization and simpler process synchronization than do designs that incorporate loosely-coupled processors.

- Multiprocessing also permits isolation of processors to perform specific real-time tasks. Processor isolation means that all ancillary processes to the real-time process (i.e., system daemons, graphics processing, etc.) can be forced away from real-time CPUs, thereby achieving improved realtime response and determinism.

Symmetric Multiprocessing and I/O

A Symmetric Multiprocessing (SMP) system is a balanced computational system where multiple processors equally share the resources of the system in order to process a given workload. The sharing of resources is generally done via a shared system bus that connects the CPUs, memory, and I/O of the system. In a true SMP system, ALL processors of a system have equal access to ALL resources of the system including memory, I/O, and the Operating System (OS) itself.

This simple fact is a key to maximizing the advantage of an SMP system. Not all multiprocessing systems are fully symmetric and, therefore, leave a potential for different performance results based on varying factors of asymmetry. For example, a multiprocessor system may have memory areas that favor certain processors such that there can be dramatic differences in performance based on which memory is being utilized by which processor.

Likewise, I/O interfaces that reside on the CPU boards of a system often provide fast access to peripheral devices from only a subset of the available CPUs. The remaining CPUs of the system are likely to either have much slower access to the interfaces or no access at all. In general, an SMP system will be more predictable in nature if resources of the system are able to be evenly distributed to the available CPUs of the system.

DISPELLING SOME COMMON MISCONCEPTIONS ABOUT REALTIME SIMULATION I/O

"Realtime simulation applications have poor cache hit ratios"

While this statement was largely true for computer systems that were being designed and manufactured only a few years ago, it is far less true today. It is rather difficult to generalize about the characteristics of all simulation code, but for purposes of this discussion a few assumptions will be made. Simulation code, as compared to code of a purely scientific application or that of a DataBase Management System (DBMS), has a greater percentage of branch instructions in its instruction stream and a larger percentage of scalar data located randomly in memory. Because of these characteristics, simulation code does not take advantage of the cache as well as do other applications.

Yet the reason that simulation applications are caching better today is due to the

simple fact that the size of caches has dramatically increased over the past few years. Table 5 shows examples of how cache sizes of simulation host computers have increased over the past ten years.

Year	Example Cache per Processor
1980	32 KB
1985	up to 128 KB
1990	up to 256 KB
1994	up to 4MB

Table 5

So substantially have cache sizes increased, that many industry "cache busting" benchmarks no longer bust caches. By the time this paper is published, there will be high-performance computer systems available with up to 4 MBytes of cache per processor. With cache sizes this large, simulation code now caches better than ever.

"Multiprocessing systems will saturate their system bus any time more than a few processors are added to the system"

As with the previous misconception, this statement is much less true today than it was a few years back. The main reason for this is in the dramatic increase in system bus bandwidth in recent years. Table 6 shows examples of changes in system bus bandwidth over the past ten years.

Year	SystemBus Bandwidth
1985	26.67 MB/sec
1990	100 MB/sec
1994	1200 MB/sec

Table 6

"I/O cycles can easily "starve" the system bus away from the CPUs"

This is a misconception that often is brought about by a lack of understanding of how data is actually transferred throughout a high performance computer system. Here is a classic example of how this misconception is formulated. A systems engineer is trying to understand the way that DMA transfers occur in his computer system in order to estimate what resources are being consumed:

1. His computer system has a sustained system bus (or memory bus) bandwidth of 1.2 Gigabytes/second.
2. His application requires a SCSI device to perform 2 KB DMA transfers into main memory at a rate of 5 MB/second.
3. He concludes that once the data transfer phase begins, his transfers will require about 400 microseconds (µs) to complete and will tie up the system bus during that period of time.

Taking a closer look at how data actually flows through this system, however, reveals quite a difference. The basic unit of transfer on this system bus is actually 128 bytes (the size of a full cache line of data) and the bus transfers each 128 byte block in 100 nanoseconds (ns). Since the interface to the SCSI bus contains enough buffering and data funnel operations to pack all operations and keep the system bus from being used inefficiently, the system bus will be used for only 16 100ns transfers. This equates to 1.6 µs on the system bus - hundreds of times less than what was originally expected.

IMPORTANCE OF UNDERSTANDING HARDWARE ISSUES AT SOFTWARE LEVEL

As the complexity of simulators and simulation systems increases, there are more and more attempts to minimize the complexity of the system to the average software developer. Many higher level languages provide constructs that encourage abstraction in an effort to simplify the amount of detail that must be understood by each programmer. For the most part, these concepts are desirable and should be encouraged. Yet this can lead to some dramatic performance implications if details of how I/O is handled by the system are not well understood.

Programmed I/O Pitfalls

By using standard system calls of virtually any OS, one may "map" VME address space to the virtual memory of a user process. This permits a user to easily read/write to the memory and registers of a VME board as if using global variables of their program. A user process then may program the VME card to perform different functions and/or send and receive data through Programmed I/O (PIO) reads/writes. Reflective memory boards are often integrated this way and provide a simple way for separate computer systems to share memory.

There are, however, several potential problems with implementing PIO that should be considered when evaluating for potential I/O bottlenecks. First of all, from a systems point of view, PIO is a very inefficient way to move data around. PIO operations cannot be pipelined and the entire I/O path from the VMEbus to the CPU remains "tied up" during the PIO operation. In addition, tests have shown that PIO utilizes 100% of a CPU's resources during large I/O transfers - leaving it unusable to other processes. This contrasts with DMA transfers which can be extensively pipelined through the use of data prefetching, freeing up the CPUs considerably.

What can also happen when using PIO is that developers may begin to treat the memory that resides on the VME as they would the normal virtual memory of their process. Simulation variables are often defined and manipulated on the VME-mapped memory such that hundreds of VME accesses are occurring with every pass through their programming model. In many cases the software developer is not even aware that VME memory is being accessed since often the address mapping occurs external to their process or program. The results of this can be disastrous. A read or write to VME memory can take hundreds of times longer than that of normal memory access.

Pipelining, Caching, Etc.

Another potential area for severe performance degradation that is often overlooked by the average software developer is the caching characteristics of a data area. As an example, the software design may utilize a large shared memory area (or datapool) for the sharing of data between processes. Then a portion of this area may be accessed from a DMA master that resides on the VMEbus. Since certain system architectures will mark entire DMA buffers uncacheable, developers must be careful that the entire shared memory area is not also marked uncacheable. To maximize the use of cacheable shared memory partitions, it is often advisable to split up normal data areas from those that will be used for I/O (i.e., multiple datapools or memory partitions).

CONCLUSION

Computer systems today offer incredulous computing power from the desktop to the supercomputer. Yet even the most powerful of computing engines cannot guarantee that a simulation be free of excessive I/O latencies and "glitches". Effectively harnessing the available computing power in the simulation and training community requires both an understanding of computer system architecture and the application itself.

This paper has presented some details about the real-time, I/O capabilities of today's high-performance, multiprocessing computer systems. It has also presented some design hints that can be used to minimize I/O problems that invariably crop up during simulator design (or even long after simulator delivery). While today's processors are hungrier than ever, understanding how to keep them well fed can ultimately help the industry to meet the challenges of the 1990s and beyond.