Test and Integration Environment for PCI Coprocessor Cards

A Thesis

Presented to the

Faculty of the

Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment of the

Requirements for the Degree

Master of Science in

Electrical Engineering

By

Mark De Ford

August 2001

Test and Integration Environment for PCI Coprocessor Cards

Mark De Ford

Approved:

Chairman of the Committee
Martin Herbordt, Associate Professor,
Electrical and Computer Engineering

Committee Members:

John Glover, Professor,
Electrical and Computer Engineering

Jaspal Subhlok, Associate Professor,
Computer Science

E. J. Charlson, Associate Dean,
Cullen College of Engineering

Fritz Claydon, Professor and Chair
Electrical and Computer Engineering

Acknowledgements

Special thanks and gratitude to my advisor Dr. Martin Herbordt for his guidance and friendship without which this thesis would not be possible. Thanks to Dr. Glover and Dr. Subhlok for serving on my committee. I would also like to extend my heartfelt thanks to my family, friends and coworkers for supporting my educational pursuits all these years.

Test and Integration Environment for PCI Coprocessor Cards

An Abstract of a Thesis

Presented to the Faculty of the

Department of Electrical and Computer Engineering

University of Houston

In Partial Fulfillment of the

Requirements for the Degree

Master of Science in

Electrical Engineering

By

Mark De Ford

August 2001

Abstract

Application specific coprocessor cards are omnipresent in today's high performance computing environment. Computationally intensive applications such as graphics, computer vision, and DNA string matching benefit from the extra computing power supplied by coprocessor cards.

This thesis is one aspect of a larger project to implement a high performance SIMD coprocessor card (SCC). Overviews of the SCC design, PC architecture and Microsoft Windows NT are presented to familiarize the reader with the underlying technology required to host coprocessor cards.

In order for the SCC to realize its full potential, the host environment must be optimized to provide high throughput, low latency communications. The focus of this thesis is to design a suitable host platform for the SCC and to investigate potential performance hindrances. A suite of benchmarks was developed to test the host/ SCC communication throughput. The testing methodology and an analysis of the results are presented.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Application specific coprocessor cards are become increasingly prevalent in today's high performance computing environment. Computationally intensive applications such as graphics, computer vision and DNA string matching are examples of the beneficiaries of extra computing power supplied by coprocessor cards. By nature of their design, coprocessor cards are heavily dependent on the hardware platforms that host them. The implication is that the design of the host software system (operating system and applications) and the CPU interconnect bus assert significant influence over the performance of coprocessors.

Several standard buses such as ISA, EISA, SCSI and SBus are available on PCs and workstations to allow additional hardware to be integrated into the system. However, these buses have a relatively low throughput. Recently the computer industry has adopted the Peripheral Component Interconnect (PCI) bus to alleviate the I/O bottleneck between CPUs and system peripherals. This is increasingly important as new generations of high performance peripherals become available.

Much work has been done on the development of coprocessor cards [8,9,11,12]. The unifying factors of these designs are that they solve domain specific problems and that PCI was selected as the datapath. However, all of the studies have a common flaw: they neglect to investigate the effects of host system software overhead on system performance. In particular, these works focus on the designing the cards, but ignore the practical issues involved with using them. Although their intent was to improve system performance by optimizing the hardware to suit the application [8,9,11,12] or simply to measure system bus performance [7,10], they failed to address one key issue: the overall system including host, operating system and peripherals needs to be optimized, not just the hardware in the coprocessor. These issues are addressed by this thesis.

1.2 Context: The SIMD Coprocessor Card

Herbordt et al. [1] have proposed a design for a SIMD Coprocessor Card (SCC). The SIMD array is designed to provide high performance for computer vision applications by providing a large number of processing elements (PE) and an operating frequency greater than 1 GHz. As the designs mentioned above, it also uses PCI.

The main difference is that the SCC control is completely hardwired; there is no general purpose CPU to control the array. This approach is necessary to maintain high array utilization. This approach, however, does place additional control responsibilities on the host: if the host cannot react immediately to array requests, then the array will idle. It is therefore of the utmost importance to optimize the host software to minimize the host response latency. The SCC design is presented in detail in Appendix A.

The research for this thesis has been done in the context of designing a host and software support system for the SCC. Therefore, it is the intent of this thesis to provide a method for quantifying OS overhead and to use this information to optimize the SCC system.

1.3 Research Motivation

The main shortcoming of other research efforts is that they neglect to consider the effects of the host design on their coprocessor board. Instead, their focus was on designing high performance logic for the coprocessor. They all used the PCI bus to interface to the system, presumably because of its potentially high peak throughput (132 MB/s) and its emergence as the de facto PC coprocessor interface standard. The issue becomes designing a host hardware and software system that can fully use the PCI peak bandwidth.

There are several reasons why both hardware and software design decisions can severely affect PCI bus utilization, two of which are as follows. First, the PCI bus is shared among multiple peripherals. It may be accepTable to block their access to the bus for a short time, but not indefinitely. A potentially catastrophic example of this is when the host operating system (OS) needs to swap memory to the hard drive, but the PCI is occupied with a large data transfer. Second, most modern OS's are multitasking (some are also multiprocessing) which by its very nature implies processes cannot monopolize the processor (CPU). Any data transfers to the PCI bus by an application or device driver can (and will) be interrupted.

1.4 Related Work

The PCI Pamette system designed by Moll and Shand [10] is the work most closely related to this thesis. This project provides excellent insight into PCI bus performance, including measurements related to application implementation. The design of the PCI Pamette board itself is not important to this thesis as most logic analyzers can perform similar functions by using a PCI bus monitor module. What is important is their testing methodology and results.

Their results show relatively high bus throughput measurements, but this is somewhat misleading as they only partially take host configuration (and OS overhead) into account. Their programmed I/O (PIO) benchmarking is straightforward and accounts for OS overhead. However, their DMA performance test procedure only measures bus bandwidth since the traffic was generated by custom hardware and targeted the host memory. This method is accepTable for measuring the bus performance alone, but has little bearing on overall system. This issue needs to be addressed if host applications are going to realize benefits from using high performance coprocessor cards.

1.5 Design Criteria and System Specifications

The design criteria for the host system are as follow:

1) The host must communicate with the SCC using the PCI bus.
2) The host and SCC must not monopolize the PCI bus when communicating.
3) The main thread of the SIMD application must execute on the host CPU. Also, it may not alter the normal operation of the host OS.
4) A programming environment must be provided so that the SCC can be easily used.

These design criteria are satisfied by the following design specifications:

1) The host platform is an Intel x86 based system with a PCI system bus.
2) The host uses the Microsoft Windows NT 4.0 (WinNT) operating system. All applications and device drivers adhere to WIN32 standards.
3) The SCC PCI bus device driver conforms to standard WinNT 4.0 I/O Manager requirements.

4) The SCC API library provides a consistent, hardware independent way for application programmers to access the SCC.

1.6 Overview of Results

The main goal of this research is to investigate the effects of the host system implementation on the SCC performance. This is accomplished by executing benchmark applications on the host that perform various sized block data transfers to a simulated SCC card (the SCC hardware is still under development). The benchmarks use different combinations of WinNT drivers and interface libraries to see what impact different implementations have on the system. This is quantified by measuring the read and write transfer time from the application, driver and PCI bus perspectives. The results are analyzed to identify possible I/O bottlenecks.

The results are surprising in several respects. First, the data throughput is independent of the driver and library implementation. This is completely unexpected for reasons that are discussed in Chapter 4. Second, the peak throughput is shown to be independent of the host CPU frequency. This result is as expected, since the PCI bus is the limiting factor. Finally, the measured peak data transfer rates are an order of magnitude slower than the peak PCI throughput of 132 MB/s. In most cases, the write throughput is approximately 20 MB/s and the read throughput is approximately 5 MB/s. The theoretical peak PCI throughput is based on the assumption that the PCI initiator can monopolize the bus for the duration of the transfer and that the target memory is fast enough to avoid inducing wait states. This environment is not representative of a normally configured system; however, Moll and Shand [10] provide results for a usable system that are still significantly better than the results presented in Chapter 4.

1.7 Thesis Outline

The next chapter provides an overview of PC hardware and Microsoft Windows NT architecture to familiarize the reader with the underlying technology required to host coprocessor cards. Chapter 3 presents the software system designed to host the SCC. It also discusses issues encountered during the development process. Chapter 4 discusses the system performance testing methodology and analyzes the results. We conclude with a discussion and suggestions for future work.

2 Underlying Technology

There are two pervasive standards in the PC industry: Intel x86 processor architecture and Microsoft Windows NT. Any device that is to find wide spread acceptance must conform to these standards collectively known as the “WINTEL” architecture. This chapter provides background material related to the WINTEL architecture necessary to understand the research results presented later in this thesis.

2.1 PC Architecture

The PC architecture has evolved significantly since its inception in the early 1980’s when most of the system was controlled directly by the CPU. In these XT/AT type systems performance was not a significant issue since memory access speeds were considerably slower than device access times. However, over time, I/O bottlenecks formed as CPU and memory access speeds increased. These issues began to be resolved by the Intel Pentium series CPUs by distributing system control over a set of system chips otherwise known as a motherboard chipset. This distributed control model freed the CPU from stalling on I/O operations and lead to an increase in system performance.

The PC motherboard designs are heavily influence by the Intel x86 architecture. The distributed CPU/chipset model is pervasive in the industry. This structure is depicted in Figure 1.

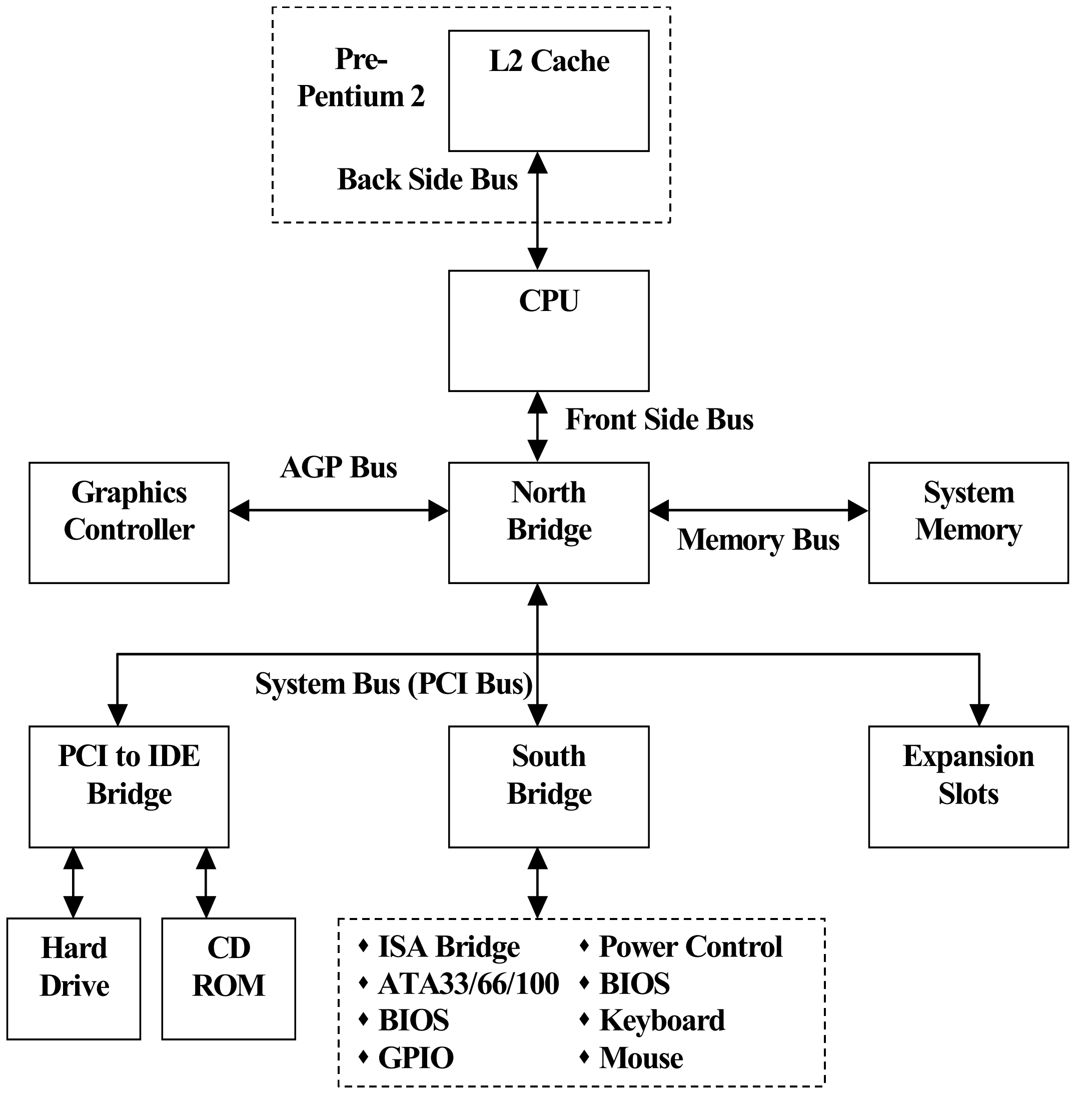

Figure 1: Block Diagram of Typical PC Architecture

The salient feature of this Figure is the hub-like bus structure centered on the North Bridge. The intent of this design is to relieve the CPU of its I/O responsibilities. The CPU only has to contend with accessing memory and performing computations. Pre-Pentium 2 CPUs used the Back Size Bus (BSB) as a high-speed, dedicated connection to the L2 cache. The BSB was rendered obsolete by the Pentium 2, which integrated the L2 cache into the CPU chip. This innovation left the CPU with only the Front Side Bus

(FSB) to service. The FSB is a high-speed (currently 200-266 MHz) local bus that connects the CPU to main memory, the graphics controller and the system peripheral bus. It is important to note that the FSB operates orders of magnitude faster than the system peripheral bus. The CPU offloads I/O operations to the North Bridge and continues operation. Equally important is the fact that the main memory is on the FBS and not the system bus. This allows the CPU to perform memory operations at high frequencies and eliminates system bus contention.

Aside from the CPU, the North Bridge is the most vital component in the system. Its purpose is to provide the interface to main memory, the graphics controller and the system bus. The main idea is that the CPU is now relieved of most I/O processing responsibilities.

In most recent motherboards, the graphics controller is integrated into the system via the AGP bus. The Accelerated Graphics Port (AGP) is a standard proposed by Intel. AGP relieves the graphics I/O bottleneck by adding a new dedicated, high-speed datapath directly between the chipset (North Bridge) and the graphics controller. This removes bandwidth intensive 3D and video traffic from the constraints of the system bus (PCI bus). AGP allows the graphics controller to access system memory directly rather than having to pre-fetch all data into local graphics memory. While the PCI bus supports a maximum of 132 MB/second, AGP operates at 66 MHz and has a 533 MB/s peak throughput. AGP performance is increased by transferring data on both edges of the 66 MHz clock and with efficient data transfer modes. AGP supports overlapped requests and has extra address lines so a new request can be started while waiting for previous access to complete (sideband addressing). The system level performance increase realized by this design is due to reduced system bus congestion. AGP operates concurrently with, and independent from, most transactions on PCI. Further, CPU accesses to system memory can proceed concurrently with AGP memory reads by the graphics controller.

Most of the discrete system peripheral control logic has been consolidated into the "South Bridge". This makes all the peripherals appear as a single device on PCI bus. This is important since each PCI bus can only support eight devices without a bridge.

The final component is the system bus, which connects the various system peripherals to the CPU via the North Bridge. The Peripheral Component Interface (PCI) bus was introduced by Intel Corporation in July 1992. It was originally designed as a local bus, but was later changed to a high-speed expansion bus. Since its inception, PCI has become the computer industry de facto standard for system buses. This point is supported the Microsoft/Intel PC 99 Guidelines [13] state that computer containing ISA and EISA buses will not be certified; effectively "obsoleting" them. The PCI standard specifies four options for address and data bus configurations (refer to Table 1) of which the most commonly deployed is 32 bit address/32 bit data with an operating frequency of 33 MHz. This configuration has a maximum theoretical throughput of 132 MB/s.

Aside from throughput, PCI has several other advantages over old buses. First, it is a synchronous bus with block transfer capabilities. This maximizes data transfer while reducing transfer setup overhead. Second, PCI provides multiple bus master capabilities. This allows peripherals to perform peer-to-peer communications without using the CPU as an intermediary. Finally, PCI provides special bus cycles for dynamically configuring devices. This allows devices to be configured before they have been assigned an address.

2.2 Coprocessor Cards

The PC architecture was specifically designed to be extensible by means of connecting expansion cards to the system bus. These cards provide diverse functions such as graphics controllers, modems, sound cards and network interfaces. However, they all generically fall into the category of coprocessor cards, i.e., they all offload a specific type of processing responsibility from the CPU.

The most interesting coprocessor cards (to this thesis) are custom boards designed to perform dedicated tasks. These coprocessors only use the PC system as a host to provide basic I/O processing. The main issue is that while coprocessor cards can be built to process data with very high throughput, connector pin characteristics and interconnect bus speeds restrict data transfer rates between the host and the coprocessor.

The AC impedance of connector pins significantly affects bus operating frequency. The capacitive and inductive components limit signal rise and fall time. This dictates the maximum operating frequency. The resistive component limits the pin drive capabilities, which determines how far signals can propagate without affecting signal integrity. This situation is exacerbated by the fact that the system bus traces have varying impedances and lengths. This is a normal part of PCB layout, but the damaging result is that signals propagate at different rates. Standard board layout techniques can be employed to reduce signal propagation delay; however, they cannot eliminate signal propagate mismatch. Thus, bus interface logic timing must be modified to accommodate the slowest signal.

The system bus I/O bottleneck is a significant issue for coprocessors. Instruction issue and data transfers are throttled by the system bus peak operating frequency and bus utilization. One solution is to create a custom coprocessor bus; however, this necessitates building a custom host or heavily modify an existing one. This situation can be avoided by employing an industry standard system bus.

Bus Type	Maximum Throughput	Notes
EISA	33 MB/second in burst.	Standards proposed for 66 and 133 MB/s bursts. Has a 32-bit data path.
SCSI-I	3 MB/second Asynchronous 5 MB/second Synchronous	Has an 8-bit data path.
Fast SCSI (SCSI-II)	10 MB/second Synchronous	Has an 8-bit data path.
Wide SCSI (SCSI-II)	40 MB/second Synchronous	Fast and Wide SCSI together. Has a 32-bit data path.
PCI	132 MB/second 264 MB/second 264 MB/second 528 MB/second	33 MHz/ 32 bit data path 33 MHz/ 64 bit data path 66 MHz/ 32 bit data path 66 MHz/ 64 bit data path
VESA Local Bus	264 MB/second	66 MHz/ 32 bit data path
IEEE 1496 SBus	200 MB/second	25 MHz/ 64 bit data 25 MHz/ 64 bit data

Table 1: Bus Speed Comparison

Most of buses listed in Table 1 are available in standard PCs and workstations with the exception of ISA and EISA. These buses are being phased out in favor of PCI. It is also interesting to note that the Microsoft/Intel PC 99 Guidelines discourage the inclusion of ISA and EISA buses in new computers.

Despite these issues, coprocessor cards are the best method of implementing application specific processors without designing custom host platforms: a standard host means that a standard operating system to control it.

2.3 Windows NT Architecture

Microsoft Windows NT (Windows New Technology) 4.0 is a secure, 32-bit operating system (OS) that uses a Graphical User Interface (GUI) for graphical, interactive user control. WinNT is a preemptive, multi-tasking OS based on a hybrid layered and microkernel architecture. At the time NT was designed, it was not certain what direction operating systems would take in regard to kernel design, POSIX support, OS/2 support, etc. Therefore, the NT architects designed it to be both flexible for adding and removing components and porTable by isolating the OS from the hardware with the Hardware Abstraction Layer (HAL). WinNT services, drivers, and HAL are implemented in Dynamic Link Libraries (DLL) that are loaded at runtime. This allows them to be changed without relinking the kernel. This modular design has proven useful since WinNT has been ported to four platforms: Intel x86, DEC Alpha, PowerPC and MIPS, although support for the PowerPC and MIPS have recently been dropped. A block diagram of the WinNT OS architecture is shown in Figure 2.

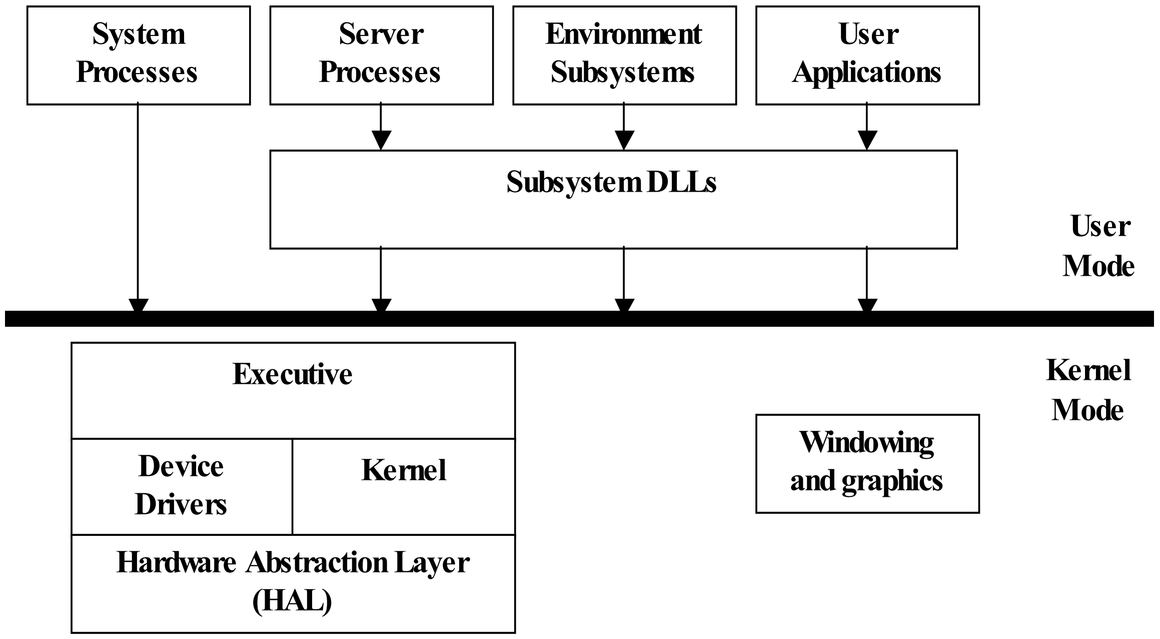

Figure 2: High Level Block Diagram of Windows NT Architecture

The WinNT environment is divided into two distinct parts based on memory and instruction access privileges. Most processing is done in the user mode. In this mode, the Virtual Memory (VM) system protects the processes' memory and the CPU blocks access to privileged mode instructions. This mode is considered secure since processes cannot affect each other's memory or access hardware directly. All user applications and most WinNT processes run in this mode. Time critical and I/O related processes execute in kernel mode. In this mode, the entire address space is accessible and the CPU permits privileged instructions to execute. The best system performance is achieved in kernel mode, but the lack of protection allows any kernel mode process to corrupt the OS environment.

There are three groups of WinNT processes that operate in user mode. System Processes are kernel support components that execute outside of kernel mode, but are required for WinNT to operate. These processes include Window Logon, Session

Manager and Services Controller, which is used to manage server processes. The server processes provide optional system level services that are not part of the core OS. Examples of server processes are the print spooler, the Windows Event Logger and the RPC service locator. The final set of OS related user mode processes is the environment subsystems. User applications use the services provided by these subsystems to emulate the WIN32, POSIX and OS/2 programming environments.

All server processes, environment subsystems, and user applications interact with the kernel through the subsystem dynamic link library (DLL) called NTDLL.DLL. NTDLL.DLL translates documented user system calls into the appropriate undocumented WinNT kernel service call.

The WinNT design goal of portability is met in part by implementing the Hardware Abstraction Layer or (HAL.DLL). Its purpose is to encapsulate all hardware and CPU specific functions into a single DLL. The HAL provides hardware support for accessing timers, the BIOS, and interrupt controls; translating bus addresses; and anything else that is machine dependent. An interesting caveat is that there is no mechanism to force kernel mode applications to use the HAL. However, applications that circumvent the HAL risk losing portability.

The Executive Services and the Kernel comprise what is traditionally thought of as the operating system "kernel". The WinNT Kernel handles the lowest level OS functions. It is responsible for thread scheduling, exception and interrupt handling, and providing low level CPU-specific services to the Executive. It is loaded into non-paged memory, and can never be preempted. The Executive is the upper layer of the Kernel. It exports kernel services to user mode applications and contains five vital system services:

1. The Process Manager is responsible using Kernel services to create and destroy processes and threads.
2. The Virtual Memory Manager (VM) is responsible for mapping processes' virtual memory into physical memory when they execute. It is also responsible for swapping memory pages to disk when the system needs more memory.
3. The Cache Manager is responsible for caching recently used file data in memory. Note: this service does not control the CPU cache.
4. The Security Monitor enforces system security policies as they pertain to system resource access.
5. The I/O Manager provides device independent I/O processing. It provides the only mechanism for user applications to interface to device drivers. The I/O Manager will be discussed in more detail later in this chapter.

From the beginning, WinNT was designed to be a fully protected OS. Security in this sense is not specifically aimed at preventing unauthorized use of the system (though the Security Monitor provides these services), but refers to preventing processes from inadvertently interfering with each other. The interprocess security policies are enforced in hardware by using the virtual memory and privileged instruction capabilities of the supported CPUs.

The Virtual Memory (VM) system makes it impossible for a user mode application to directly access a physical address. The intent of VM is to provide processes with what appears to be unlimited memory. A positive side effect is that applications are prevented from corrupting each other's memory. Normally this is desirable, but in the case of accessing memory mapped system hardware, this is a significant issue. This is a basic reason why device drivers are required by WinNT.

Processors also enforce security by providing at least two modes of operation: privileged mode and user mode. When in user mode, the CPU can only execute a subset of the full CPU instruction set. Instructions that are excluded from user mode include I/O, CPU mode switching, and special register access instructions. The CPU must be in privileged mode to access these instructions or a protection fault is generated. This also necessitates the use of device drivers.

2.4 Windows NT I/O Manager

The I/O Manager is the Executive component that provides user mode applications access to hardware resources while still protecting system resources. The upper level of the I/O Manager makes drivers appear as File Objects (similar to VMS and UNIX). User applications use the standard WIN32 file access functions to interface to the driver. The lower level of the I/O Manager packages I/O request information into packets call I/O Request Packets (IRP) and delivers the IRP to the appropriate driver. This same mechanism can be used by device drivers to communicate with each other to create layered drivers. The basic flow of IRP processing is shown in Figure 3.

The I/O Manager has several features worth noting. First, it allows drivers to be dynamically loaded and unloaded. This allows drivers to be managed without rebooting the system. The I/O Manager is also multiprocessor safe. This feature permits a device driver to function properly in multiprocessor systems. Finally--and most important to developers--the I/O Manager supplies the common epilog and prolog required by all drivers. It builds the IRPs, manages the IRP buffers, routes the IRPs, provides operation watchdog timers, and performs clean up functions when the I/O operation is complete. All this makes drivers more compact and easier to develop.

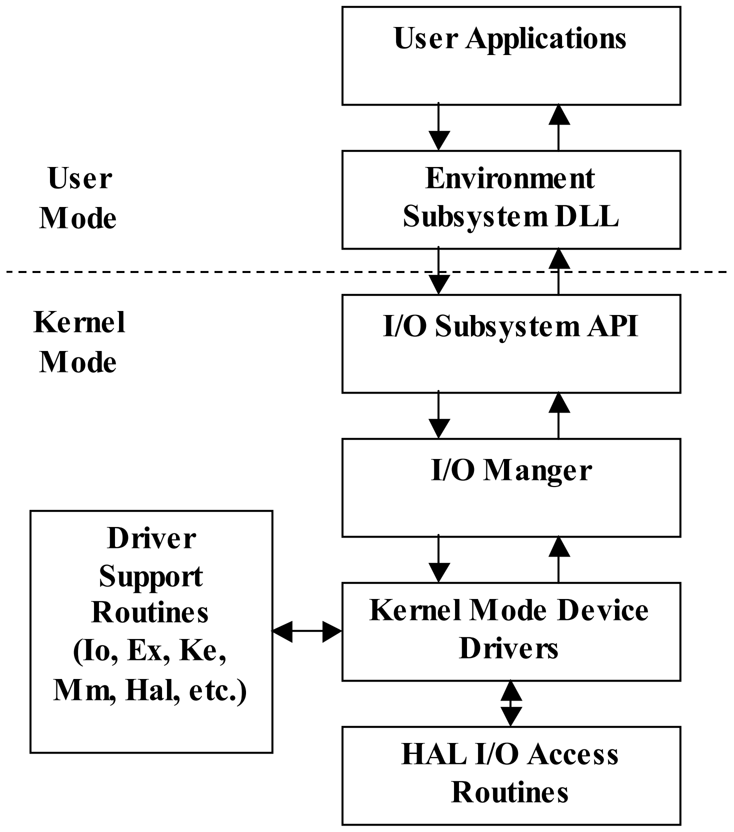
User Applications
User Mode
Environment Subsystem DLL
Kernel Mode
I/O Subsystem API
I/O Manger
Driver Support Routines (Io, Ex, Ke, Mm, Hal, etc.)
Kernel Mode Device Drivers
HAL I/O Access Routines

Figure 3: Windows NT I/O System Structure

2.5 Driver Structure

The purpose of using IRP packets is to provide a generic method for the I/O Manager to communicate with drivers without having specific knowledge of them. The I/O Manager maintains a function dispatch Table for every executing driver. The type and order of the dispatch Table entries are identical for all drivers.

Every driver is required to provide an entry point called DriverEntry(). This code is responsible for initializing the driver environment, device hardware, and populating the dispatch Table. There are five required dispatch Table entries (functions):

1) IRP_MJ_CREATE. This function is called when the driver is opened by an application.
2) IRP_MJ_CLOSE. This function is called when the driver is closed.
3) IRP_MJ_READ. This function is called when an application calls the WIN32 ReadFile() function. This function is generally only used for file system drivers.
4) IRP_MJ_WRITE. This function is called when an application calls the WIN32 WriteFile() function. This function is generally only used for file system drivers.
5) The Driver Unload function is called by the I/O Manager when the driver is unloaded. It must disable the device hardware and release all OS resources claimed by the driver.

An important optional function is IRP_MJ_DEVICE_CONTROL. It is used to implement a custom interface to the driver. When a user application calls the WIN32 function DeviceIoControl(), the input buffer is sent directly to the driver and the driver returns directly data in the output buffer. The I/O Manager does not interpret or otherwise use the data.

The I/O Manager has two different methods for handling input and output buffers passed to drivers: direct I/O and buffered I/O. In direct I/O, the I/O Manager passes a Memory Descriptor List (MDL) containing the location of the input and output buffers in the user memory space. The driver uses the MDL to map the buffers into the driver's address space so that they can be accessed directly by the driver. In buffered I/O, the I/O Manager allocates I/O buffers from the kernel non-pages memory pool. The contents of the user space input buffer are copied into the input buffer allocated by the I/O Manager. The input buffer allocated by the I/O Manager is then forwarded to the driver. The driver output buffer is processed similarly. The driver stores return data in the system allocated output buffer, which is then copied into the user space buffer by the I/O Manager when the I/O request is completed. It seems reasonable to assume that these interface methods add significant overhead data transfer operations. This hypothesis is investigated in this thesis.

When writing device drivers or dealing with the I/O Manager, developers need be cautious while performing pointer operations. Solomon [4] reiterates a previously stated point: “Windows NT doesn’t provide any protection for components running in kernel mode.” Code executing in kernel mode has unlimited access to all kernel memory and CPU instructions which gives it the power to corrupt the operating system.

3 Software System Design

3.1 Overview

A flexible, extensible software system is required to support the SCC hardware design presented in Appendix A. In order to the SCC to be usable, several host system issues must be addressed: first, how the host interfaces to the SCC and second, how user applications use the SCC. These design criteria must be met within the WinNT and Intel x86 architectures. Fortunately, WinNT provides the well-defined driver structure for accessing the hardware, as well as an interface for applications to access driver services.

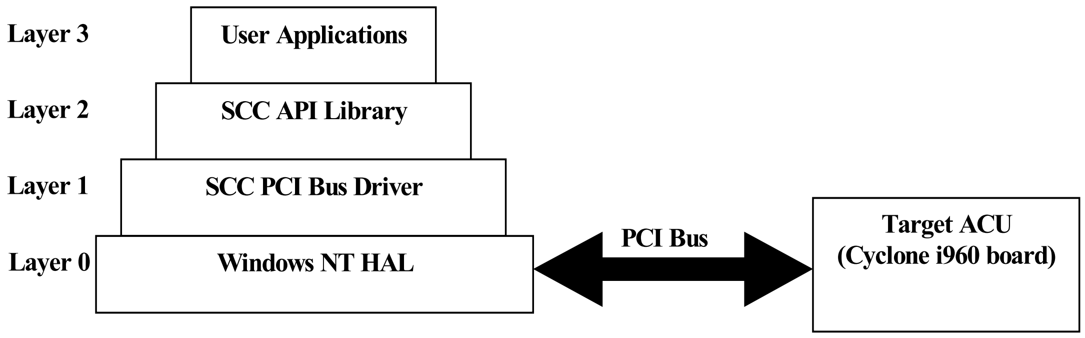

Figure 4: SCC Software Architecture

The SCC support software structure follows a layered approach as illustrated in Figure 4. This layered architecture assures that each level is isolated from changes in the others. Each interface for layer 0 through layer 2 is dictated by Microsoft. Adhering to these standards provides portability across the various hardware platforms that support WinNT. The layer 0-1 interface is defined by HAL functions. These functions are only accessible while the CPU is in privileged mode. The layer 1-2 interface is defined by the WIN32 file access functions: CreateFile(), WriteFile(), ReadFile() and

DeviceIoControl(). No other functions are provided to access the driver from user mode. Finally, the layer 2-3 interface is controlled by the SCC Array Interface Library (AIL). Its purpose is to provide user applications with a consistent interface to the SCC services without requiring specific knowledge of the SCC hardware. It also provides portability by obscuring the specifics of driver calls.

3.2 SCC PCI Driver

The software system requires a Windows NT device driver to communicate with the SCC ACU via the host PCI bus. This component is not optional, as memory mapping and privileged mode instructions are required to interface with the SCC memory.

3.2.1 Design Criteria

The primary goal of the driver development was to optimize the code to increase overall system performance. The optimization effort focused on the IOCLT dispatch code since this is the only code that has a bearing on runtime performance. The speed of the initialization and de-initialization code is irrelevant since their execution occurs when the SCC is unusable. In the end, not much performance was gained from hand tuning the driver. It is impractical to write the driver in any language other than C; recent C compilers generate execuTable code almost as efficiently as that produced by a good assembly language programmer. The CPU cache structure can alleviate some of the inefficiencies as well.

The driver was developed to accommodate reentrancy and multiprocessor operation. These capabilities were designed in to provide a path for future system expansion. The only coding overhead incurred is ensuring all variables are allocated from the stack and adding a spin lock to driver dispatch function to ensure exclusive access.

3.2.2 Implementation

There are essentially four types of WinNT drivers, buffered I/O, direct I/O, memory mapped, and DMA based. The SCC PCI Driver provides functionality to support the first three modes in the same driver. This is possible by defining IOCTLs to support the three modes in the same source file.

The SCC PCI Driver implements the essential WinNT driver functions: a driver entry point, device I/O control dispatcher, and a driver unload routine. These are now described.

DriverEntry() is the driver entry point. It has four major responsibilities:

1) PCI bus enumeration. It probes the PCI bus for SCC cards and creates a device object for each SCC card found.
2) Creating a symbolic link so that the driver can be accessed by WIN32 file functions.
3) Initializing SCC hardware for use and mapping its memory into kernel memory.
4) Initializing the driver function dispatch Table.

WIN32 DeviceIoControl() calls (which generate IRP_MJ_DEVICE_CONTROL IRPs) are handled by the driver Dispatch() function. It supports nine device I/O control functions (IOCTLs), but only four warrant discussion. The first pair invoke IOCTL_SCC_MAP_USER_PHYSICAL_MEMORY (which calls MapMemory()) and IOCTL_SCC_UNMAP_USER_PHYSICAL_MEMORY (which calls UnMapMemory()). The MapMemory() function maps the SCC memory into the calling application's memory by using several HAL functions. The virtual address is return to the caller in the pointer supplied by the call. An excellent example of this is provided in Dekker and Newcomer [6] page 375. UnMapMemory() simply removes the mapping. The other pair is IOCTL_SCC_LOAD_DATA_BLK and IOCTL_SCC_READ_DATA_BLK. These IOCTLS perform the block transfers to and from the SCC memory. These are noteworthy because they use the HAL functions WRITE_REGISTER_BUFFER_ULONG() and READ_REGISTER_BUFFER_ULONG() to effect the transfer. The code behind these functions is not remarkable (simply assembly language loops), but the functions should be used in order to ensure driver portability. The implementation of these functions may vary across NT platforms, but the function prototypes are immuTable.

The driver unload function, DriverUnload(), must reverse the set-up performed by the DriverEntry() routine. The cleanup process consists of five steps:

1. Delete the symbolic links. This deregisters the driver with the NT Object Manager. The driver can no longer be reference by name.
2. Disable the SCC board interrupts and disconnect the driver ISR from the NT ISR list.
3. Unmap the SCC board memory from kernel address space.
4. Release the resources that were assigned by HalAssignSlotResources().
5. Release the device object.

The driver functions for creating/opening (IRP_MJ_CREATE), closing (IRP_MJ_CLOSE), reading data (IRP_MJ_READ), and writing data (IRP_MJ_WRITE) to a device are stubbed. They have no significant function in this driver, but they must be implemented according to WinNT driver standards.

3.2.3 Validation Testing

The SCC driver functionality was verified in several ways. One is that all driver development has an inherent “Go. No Go.” test referred to as the “Blue Screen of Death”. Drivers execute unprotected in kernel mode. If they contain an error, the results are typically devastating to WinNT. If the error is not too severe, the kernel catches it and displays the “Blue Screen of Death”. If the error is server enough, the computer locks up and the effects are completely indeterminate.

The SCC Driver functionality was more methodically tested using two test tools. The basic functions were tested using Microsoft WinDBG. This debugger displays messages embedded in checked build drivers if the debugger is active and the debug feature of WinNT is enabled at boot time. Otherwise, message support is disabled. Test messages are embedded in the major blocks of the SCC PCI driver. A test application was written to exercise all of the SCC driver functions and the debug output was monitored with WinDBG. This method proves basic driver functionality, but does not verify memory transfer operations.

Since the SCC board is still under development, it is desirable to have a known good target to verify memory operations. The Cyclone i960 development board provides a suiTable environment for this task. The board provides an Intel i960 CPU with an onboard debug monitor and a user interface via a serial port. The i960 also has a PCI interface that provides access the board DRAM. The validation test consists of a test application executing on the host that uses the driver to perform a write/readback/verify test on the i960 memory. The i960 debug monitor is used to examine the i960 memory for the proper test patterns.

3.3 Array Interface Library

3.3.1 AIL Design

The purpose of the Array Interface Library (AIL) is to provide programmers with a consistent environment for accessing the SCC. This serves several purposes. First, the programmer is not required to know the inner workings of the SCC hardware. The application programming interface (API) allows data and/or programs to be sent/ retrieved from the array by making a function call. Second, applications are completely insulated from changes to the hardware and/or device driver. A library port allows the system to target alternate hardware platforms and/or host operating systems. The AIL was designed to be fully reentrant and to support multiple SCC cards in the host platform.

There are four versions of the AIL available:

1. Static linked C library
2. Static linked C++ library
3. C WIN32 Dynamic Link Library (DLL)
4. C++ DLL

This provides programmers with as many implementation options as possible. In addition, this provides a means to compare the system level performance of static libraries vs. DLLs and C vs. C++.

The different versions of the AIL are derived from the static link C library source stream. A stub was added to facilitate DLL use and a C++ wrapper class was developed to support object oriented programming. For the remaining discussion, AIL will refer to the static linked C version of the library.

The AIL provides API functions to:

1. Read and writes blocks of data from/to SCC memory.
2. Read the SCC driver information.
3. Load basic program blocks into SCC memory and execute them.
4. Issue directives to the ACU.
5. Wait for and process feedback from the SCC array.

These functions segregate into two general categories: low level primitives to interface to driver and perform basic data transfers, and higher level, SCC specific interface functions. The following sections discuss the design concepts of the AIL functions and are not intended to be an AIL tutorial.

3.3.2 Library Interface Functions

SccLibraryOpen() opens a connection to the specified SCC. It returns a handle that uniquely identifies this connection and this handle used by the other library functions to identify which SCC they are accessing. The complement function is SccLibraryClose(). This function must be called when the connection to the SCC is no longer needed.

The definition of the handle is arbitrary from the user viewpoint; it is simply a way to identify with which SCC card to communicate. From the AIL implementation standpoint, the handle is actually the WIN32 handle returned from the CreateFile() call that opened the SCC driver. This is a good example of the complex implementation details obscured from the application programmer by the AIL.

The AIL provides the SccWriteBlock() and SccReadBlock() functions to facilitate transfer between user applications and the SCC. These are the only functions that user applications should use for data exchange with the SCC, mainly because they obscure the transport mechanism. Based on the experimental test results presented in Chapter 4, the AIL uses the services of the direct I/O driver for data transfer. In this configuration, the WinNT I/O Manager provides addition protection against programming errors.

In the ideal AIL implementation, all other AIL functions would use these low-level data transfer primitives. This would minimize the impact of changing layer 1 drivers. However, the current AIL functions call the WIN32 driver interface functions directly in an effort to improve performance. This optimization, however, is shown to be unnecessary based on the library performance test results in Chapter 4.

The ACU interface functions are built on the low-level primitives and they provide the programmer hardware-independent methods for controlling the SCC ACU. They obscure details such as control and status register format and memory address. A description of the functions is listed in Table 2.

Function	Description
SccWaitOnCondition()	This function polls the SCC status register until the conditions specified by the bit mask are met. This is used primarily to stall the main application thread until array feedback is available.
SccReadFeedback()	This function returns the current value of the SCC feedback queue.
SccWriteImmediate()	This function inserts an immediate value into the SCC input queue.
SccReadStatusRegister()	This function returns the current value of the SCC status register.
SccWriteStatusRegister()	This function writes a value to the SCC status register.

Table 2: List of SCC Interface Functions

A group of functions is provided to interface with the layer 1 driver (See Table 3). These functions are provided primarily for testing purposes.

Function	Description
SccGetDriverInfo()	This function returns the copyright and version information from the driver. It can be used as a simple test to verify driver operation.
SccGetPciInfo()	This function returns the PCI bus information detected by the driver during bus enumeration.
SccGetCardCount()	This function returns the number of SCC boards found on the PCI bus during bus enumeration.
SccResetCard()	This function forces the SCC board to reset.
SccGetStatus()	This function returns the current status of the driver. This is currently limited to indicating if another application has locked the SCC board as a resource.

Table 3: List of Driver Interface Functions

The AIL provides two functions to gather system performance information and are used here to gather test data during the system performance evaluation. The function SccGetTimer() calls into the driver to return the value of the CPU 64 bit performance counter. The instruction (RDTSC) used to read this timer is a privileged mode instruction so it must be executed from within the driver. The function SccGetElapsedTime() returns the time the previous driver call took to complete. This was used here to determine the amount of time data transfers to the SCC took from the perspective of the driver.

The AIL also contains a pair of functions to memory map the SCC memory directly into user application memory. SccMapMemory() performs the map function and SccUnmapMemory() frees memory when it is no longer needed. These functions are provided primarily for use in the memory mapped layer 1 driver and should not be used by user applications. The main problem is there is no protection for the kernel memory when mapped to user memory space and an errant program could easily corrupt WinNT operation.

The AIL software architecture provides a flexible, modular interface to the SCC. Each layer provides an avenue for portability and protection from errant programming. It would be simple to port the software system to a non-Intel WinNT environment; essentially a recompile. For other operating systems, however, the driver (which is inherently OS dependent) would have to be rewritten; however, its basic structure could be applied to UNIX and LINUX.

3.4 Benchmark Applications

The AIL affords the application programmer a high degree of flexibility in how to use the SCC. The question becomes, how much does all this flexibility cost? Part of the SCC software system is a series of benchmark applications to measure the performance of the driver and the AIL components. More specifically, they attempt to quantify the OS overhead associated with each WinNT driver type discussed previously as well as the library implementation overhead (static linked library vs. DLL). The goal is to determine what effect drivers and/or libraries have on the overall system performance. The benchmark applications specifics are discussed in the next chapter.

4 Results

This chapter discusses experimental results in this thesis. The primary goal here is to investigate the effects of the host software implementation on the SCC performance. This is accomplished by executing benchmark applications that perform various sized block transfers targeting a simulated SCC. The Cyclone i960 development board is used for testing since the SCC is still under development. The i960 board also has the added advantages of providing an onboard debug monitor and a known good PCI interface.

The benchmarks used different combinations of WinNT drivers and interface libraries to test the impact of various software implementations on the SCC system. Data transfer times are measured from the application, driver and PCI bus perspectives in an effort to identify possible I/O bottlenecks.

The rest of the chapter presents the testing methodology followed by the SCC driver, the AIL library overhead, and finally the PCI bus utilization test results.

4.1 Testing Methodology

The research focuses on determining the effects of WinNT overhead on system performance. There are two hypotheses under study. The first is that memory mapped drivers have a significant performance advantage over both the buffered I/O and direct I/O drivers. This seems intuitive, as the involvement of the I/O Manager in data transfer operations must be slower than writing directly to the target memory. The second hypothesis is that the statically linked implementation of the AIL provides better performance than the C++ and DLL versions. The rationale is that the code from static linked libraries is directly linked into the application; it is executed from the same code segment as the rest of the application. On the other hand, WinNT DLLs are loaded into kernel memory on demand and applications resolve the execution addresses at runtime. C++ versions are tested primarily because of the long-standing debate as to its impact on code efficiency.

A suite of benchmark applications was developed to test these hypotheses. The benchmarks are standard WIN32 console applications written to test the PCI throughput between the host CPU and the SCC. PCI bus throughput is considered the critical factor in determining overall system performance since SCC array and host CPU operating frequencies are significantly faster than the PCI bus. The benchmarks are designed to write then read back data blocks from the i960 board (simulated SCC). The block sizes range from 4 bytes to 2 MB with the block size doubled for each iteration. The benchmark pseudo code is:

```
Main()
{
    for(  block_size=4; block_size<=2MB; block_size*=2 )
    {
        read timer for app start time;
        do driver transaction;
        get driver run time;
        read timer for app end time
        Tapp = Tend – Tstart;
        Write numbers to log file;
    }
}
```

Block transfer time is an important performance measure for all systems that use coprocessor cards. The SCC is designed for vision and graphics applications. These types of problems require large blocks of data in the form of images to be frequently moved between the host and SCC. Code segments also have to be loaded from the host to the SCC. Particularly for the SCC, small block transfers are also important. The user applications on the host regularly send directives to the ACU and write to the control register. User applications also read the SCC status and feedback registers to control program flow. These operations are all time critical.

Three performance measurements are taken by the benchmarks. First, the block transfer time is measured from the application viewpoint. This is the most important parameter since it reflects the total runtime of the application. It includes OS overhead, compiler and library inefficiencies, and system bus utilization (assuming a constant workload). The second measurement is the time spent in the driver performing the data transfer. The purpose of this measurement is to quantify how much OS overhead is associated with the application and how efficiently the driver executes. In theory, the OS overhead is the time in the driver subtracted from the value of the application timer. The final measurement is the amount of time the transfer actually takes on the PCI bus. This can be measured using a PCI bus analyzer; in the case of this investigation an HP 1671E with a FuturePlus Systems FS2005 PCI probe. This is a difficult measurement to take, but it shows the actual transfer time. This value subtracted from the application time and the driver time represents the OS overhead for the respective operation. The application and driver overhead measurements indicate where the system inefficiencies reside.

Collecting accurate timing information is challenging. WinNT provides software timers, but they suffer from OS overhead when updating when they are read. They are also have relatively low resolution; on the order of milliseconds. The PC hardware provides timers that would be useful except that the WinNT VM system prevents them from being accessed directly by user applications. Also, there is no way to know if or how these timers are allocated by WinNT.

The solution is to use the 64 bit performance timer supplied by the CPU. This timer is reset when the CPU is reset and incremented on every instruction cycle. However, this method has some limitations. Mainly, it uses the RDTSC instruction, which is native to Intel Pentium and above processors. RDTSC is a privileged mode instruction so it must be invoked within the context of a driver. This skews the accuracy of the measurement. The skew has two components. First, there is the code overhead of calling the driver and the driver handling the call. This can be accounted for by counting instruction cycles expected from the assembly language listing and then subtracting them from the timer value. This does not completely account for the call overhead since any part of the operation can be preempted. This is the second component of the measurement skew and it is exact impact is indeterminate. As long as WinNT is running, there is potential OS overhead associated with every operation. The exact effects vary with workload, OS configuration, interrupt frequency, and hardware configuration.

The WinNT workload directly affects the performance measurements. WinNT requires certain processes be executing for the system to function (see Chapter 2). For testing purposed, all nonessential user applications and system processes were terminated. The goal was to provide a lightly loaded system to get the best performance possible though this is not a normal system configuration. It is also important to restrict the workload to avoid loading the PCI bus with hard drive and system peripheral accesses. These directly contend with the benchmark applications for use of the PCI bus and skew the test results.

4.2 Test Procedures

The test environment is shown in Figure 5. The system contains the PC under test, a PC to access the Cyclone i960 debug monitor and an HP1671E logic analyzer with a FuturePlus Systems FS2005 PCI probe.

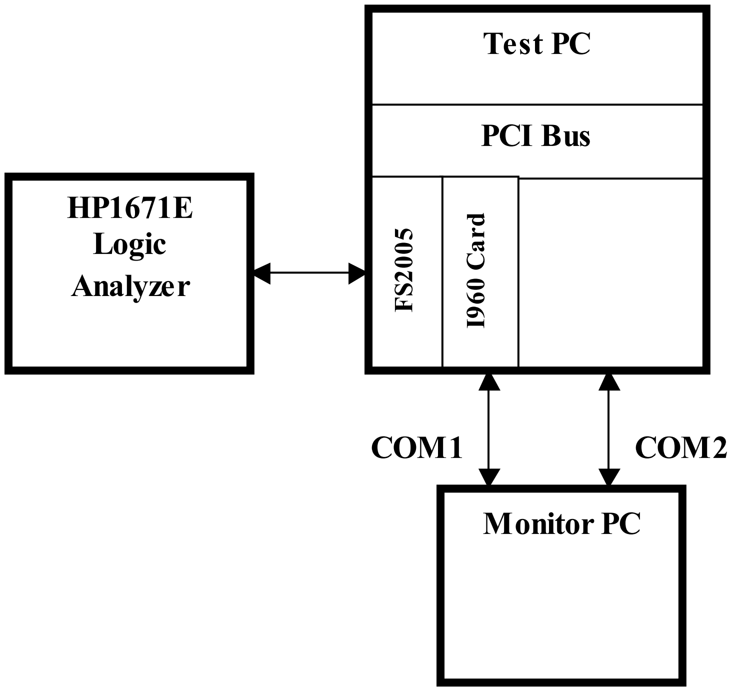

Figure 5: Hardware Test Setup

The Test PC is loaded with WinNT 4.0 and the benchmarks. All extraneous processes and services are terminated, but as previously mentioned, some system processes are required for WinNT to operate. The FS2005 probe is placed in any open PCI slot in the Test PC. The cables from the HP1673E logic analyzer are connected to the FS2005 as demonstrated in the FS2005 User Manual. The PCI bus monitor software is loaded into the logic analyzer by selecting the "CP256_1" configuration file on the system disk, selecting “Load from flexible disk” followed by pressing the “Execute” button. The trigger is configured as described in FuturePlus application note “Capturing PCI Bus Transactions.” The i960 board should also be inserted into a PCI slot on the Test PC. The serial line is connected from the RJ-11 jack on the i960 board to a COM port on the Monitor PC. Note: plugging the i960 into the FS2005 expansion connector is not recommended. The electrical characteristics of PCI are such that the impedance of the extra trace lengths on the FS2005 could corrupt the signals supplied to the i960 board.

The main purpose of the Monitor PC is to provide a serial console for accessing the i960 debug monitor. Its use is not required while running the benchmark tests, but it can be used at any time to verify the correctness of the data transfers. Windows HyperTerm can be used to communicate with the debug monitor. The communication parameters are 115 kbps, 8 data bits, 1 stop bit and no parity. To start the communication session with the debugger, press the <Enter> key 8 times. The debug monitor supports a variety of commands; however, the most useful commands are “dd” (display doubleword) and “mo” (modify memory contents). These commands are described in detail in the Intel "MON960 Debug Monitor User’s Guide" (Document Number: 484290-006).

The Monitor PC can optionally have the WinNT kernel debugger (WinDBG) installed. WinDBG is used mainly for driver development; not performance testing. In fact, WinDBG was disabled during testing because of overhead it introduces into the system. Enabling the debug features of WinNT slows it down immensely. In addition, debugger communication occurs at 115 kbps over the serial port. This greatly affects system performance.

The following sections present the results of the benchmark tests discussed in Section 4.1. The driver test results are analyzed first, followed by the library overhead tests and finally the bus utilization test. Tests were executed on a 133 MHz Intel Pentium processor with an Intel Triton II chipset (hereafter referred to as Pentium) and a 1.2 GHz AMD Athlon with a VIA KT133A chipset (hereafter referred to as Athlon). This seems like an unfair comparison, but the disparate systems (CPU and chipset) were selected because there is roughly an order of magnitude difference in their processing capabilities. The goal is to expose possible benchmark CPU dependencies.

4.3 Driver Test Results

The following sections present the throughput results for the memory mapped, direct I/O and buffered I/O drivers. The benchmark applications did not use the AIL functions; they call DeviceIoCtrl() directly. This was done to remove library related performance issues. The AIL impact on system performance is presented in Section 4.4.

4.3.1 Memory Mapped Driver

This section presents the results for the memory mapped driver. The designation memory mapped driver is slightly misleading. The driver is not actually involved in the data transfers. The only service it provides is to map the i960 memory into the benchmark's user memory space. The memory mapping function must be implemented in this manner since kernel mode privileges are required to call the HAL mapping functions. This driver was expected to have the best performance since the applications are writing directly to the i960 memory without the overhead of involving the I/O Manager. ????????????????????????

The write throughput results for the Pentium and Athlon are shown in Figures 6 and 7. No results for the driver execution time are presented, as the driver is not involved in the data transfers.

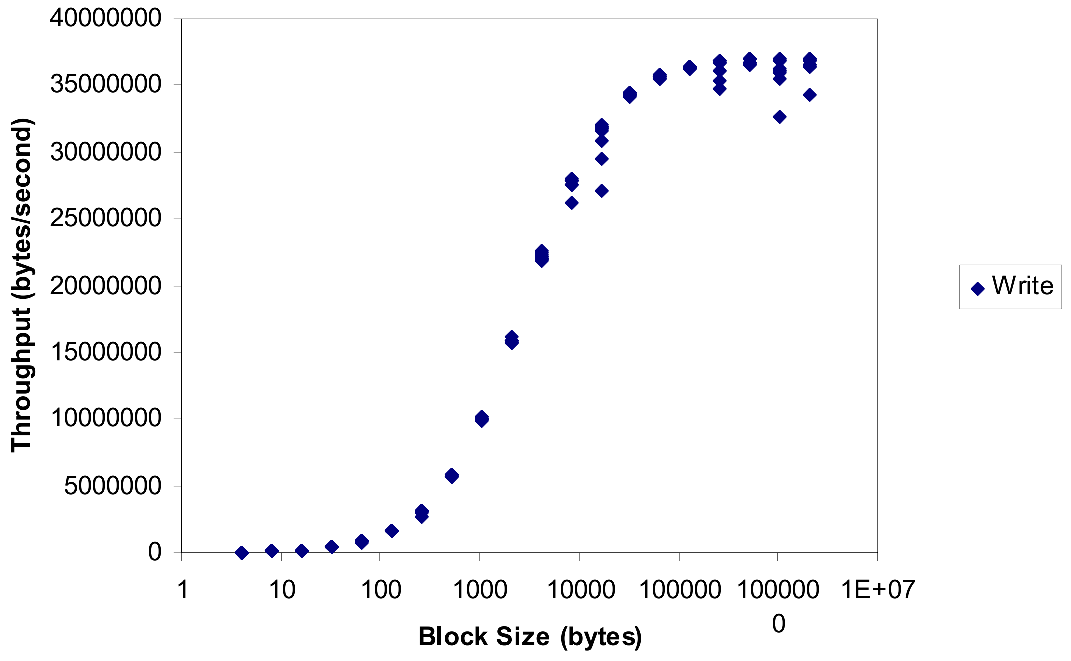

Figure 6: Memory Mapped Driver Write Throughput on Pentium

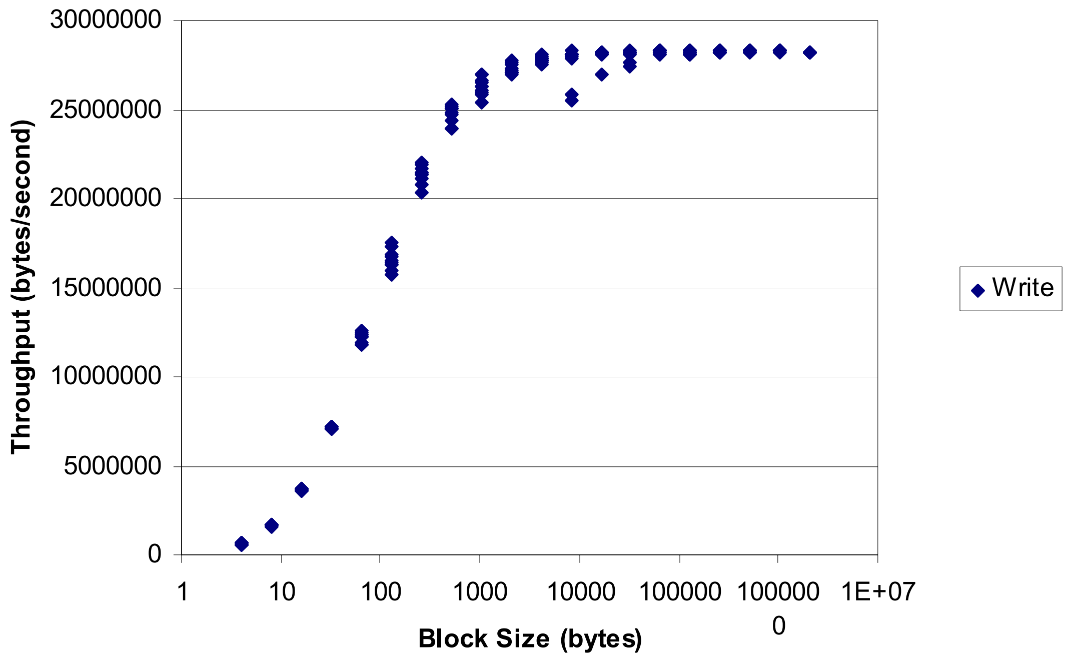

Figure 7: Memory Mapped Driver Write Throughput on Athlon?????

It is easily gleaned from these figures that the write throughput is substantially lower than PCI theoretical peak throughput of 132 MB/s. This is attributed mainly to operating system overhead, though the shared system bus contributes a small component.

The read throughput results for the Pentium and Athlon are illustrated in Figures 8 and 9. These results are invalid????? for the purposes of this study because the data transfer times for all block sizes are identical. The most plausible explanation for this is that both CPUs are reading from cache even thought the driver designates the i960 memory as non-cacheable. This poses a serious problem for the SCC which uses memory mapped control and status registers. Writes and reads are required to return the current state of the i960 memory otherwise the host cannot make proper control decisions.

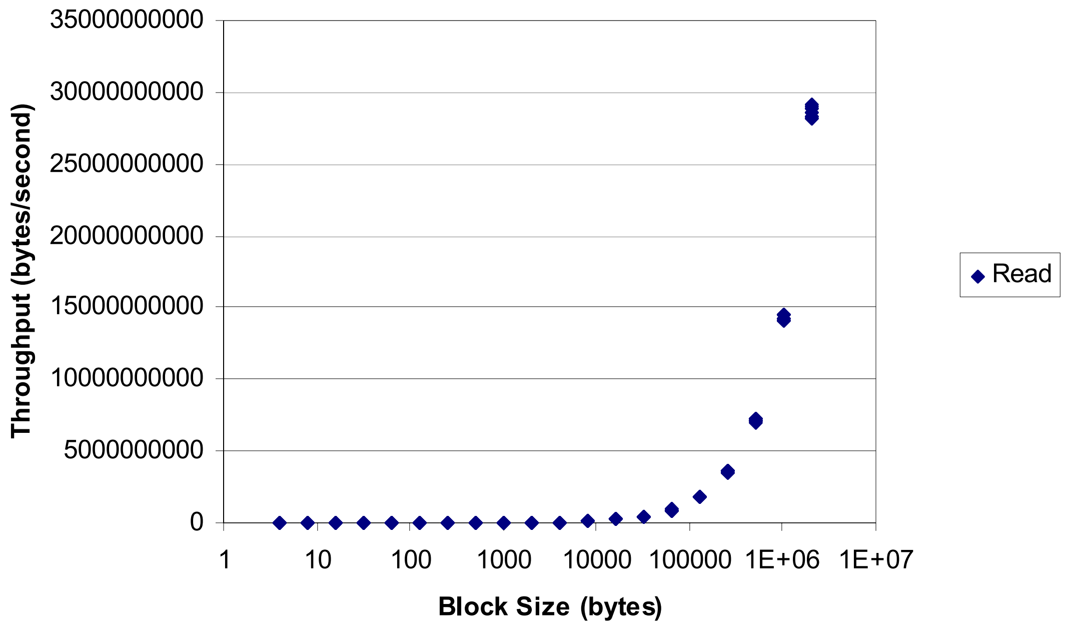

Figure 8: Memory Mapped Driver Read Throughput on Pentium

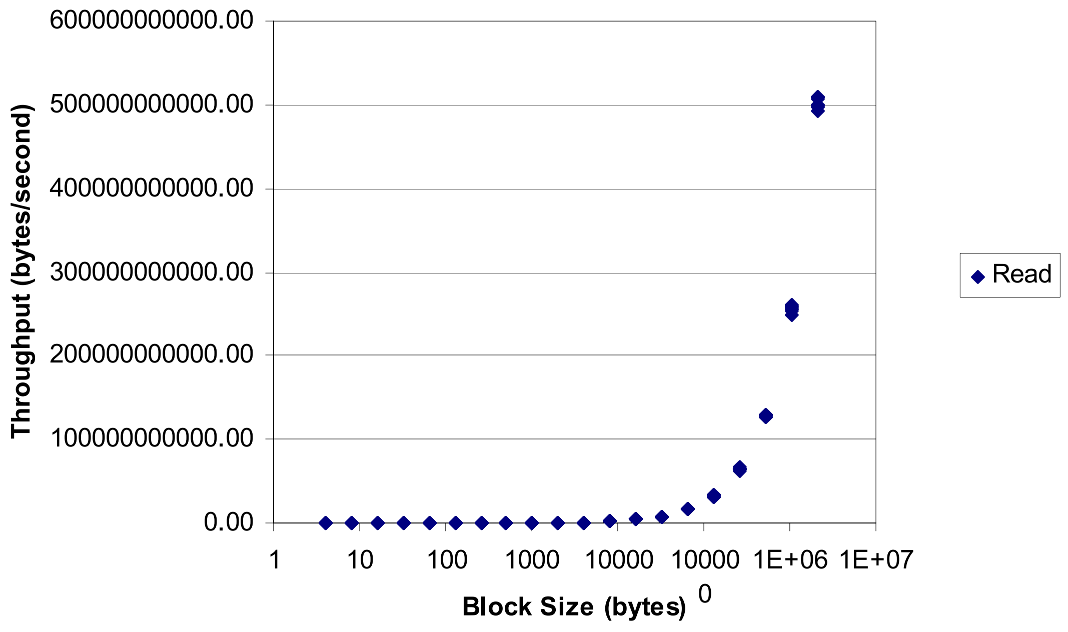

Figure 9: Memory Mapped Driver Read Throughput on Athlon

The test results for this driver were at best disappointing. As will be shown in the following sections, this driver does not have a performance advantage over the buffered

I/O or the direct I/O drivers. This is completely counterintuitive since accessing memory directly is typically faster than involving a third party (the I/O Manager in this case) in every data transfer. The best possible explanation for this is based on the WinNT scheduling priority scheme (see Solomon [4] page 187 or Dekker and Newcommer[6] page 10 for background information). Device drivers always run at a higher priority than user applications; therefore, they are less likely to be preempted. It is possible that performance of the benchmark could be improved by raising its thread priority. However, this violates the design principle of not monopolizing the CPU stated in the introduction and could destabilize the host operating system environment.

4.3.2 Direct I/O Driver

As described in Chapter 2, the direct I/O driver is a hybrid of the memory mapped and buffered I/O drivers. Data input to the driver is copied into a kernel buffer before passing it to the driver. The user supplied output buffer is directly mapped into kernel memory; eliminating the need to copy the receive data from a kernel buffer to the user buffer.

The write performance for this driver is comparable that of the buffered I/O driver. This is to be expected since they both buffer user input. The disparity between the write and read throughputs is attributed to write merge logic either in the CPU or the North Bridge. This was verified by modifying the driver to write 8 bit words instead of 32 bit words; the write throughput was unchanged.

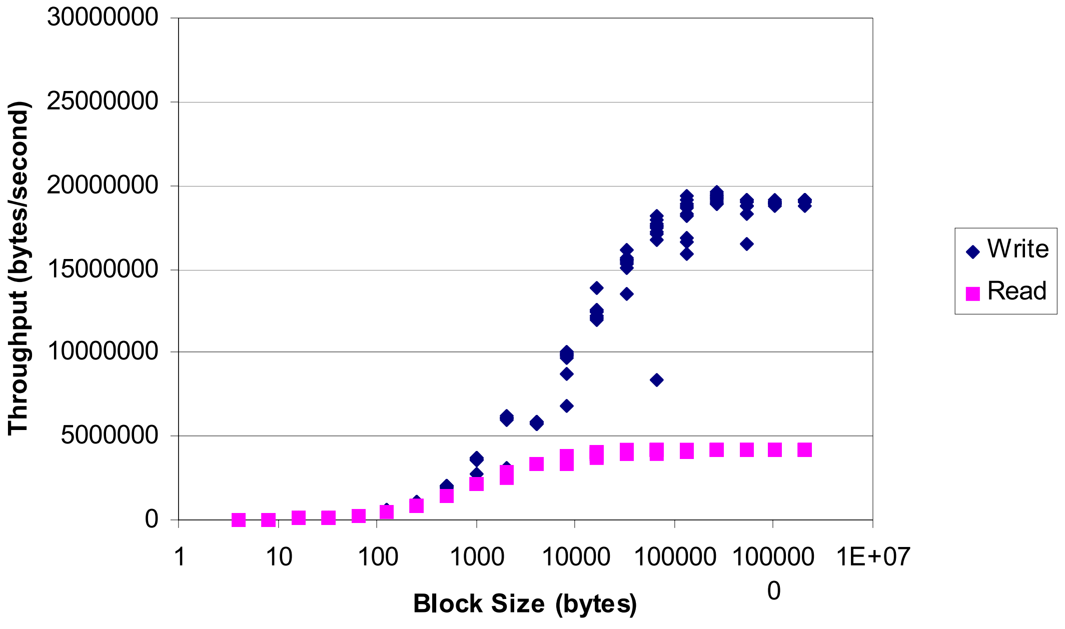

Figure 10: Direct I/O Driver Throughput for Pentium

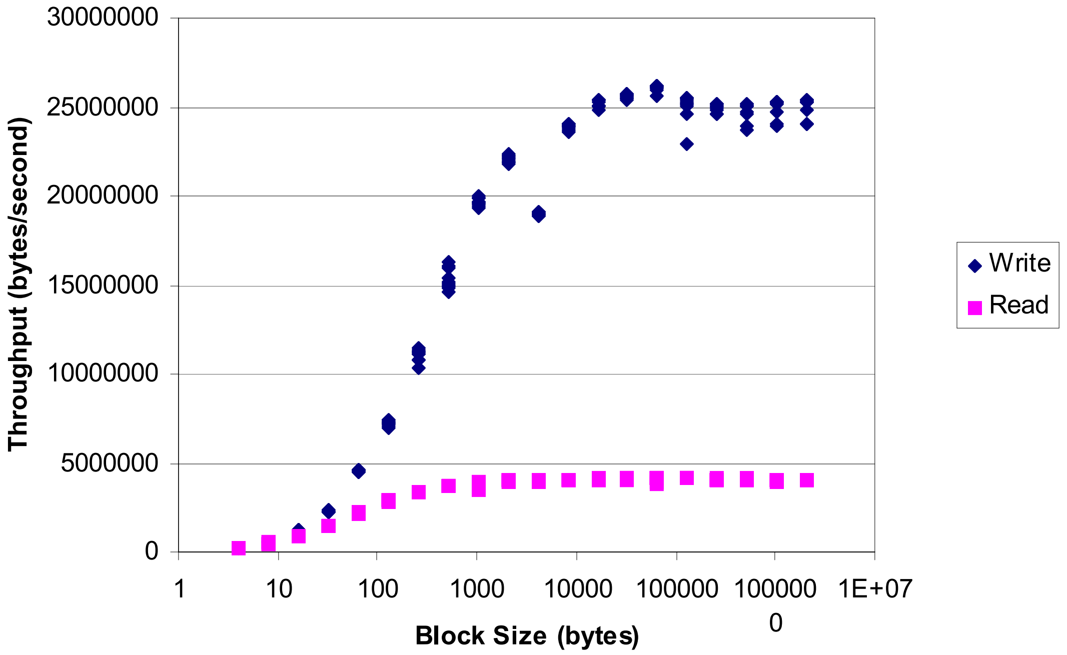

Figure 11: Direct I/O Driver Throughput for Athlon

The read throughput for the direct I/O driver is slightly higher than for the buffered I/O driver. The typical read throughput for both the Pentium and Athlon using the direct I/O driver was approximately 4.1 MB/s. This same measurement using the buffered I/O driver was 3.4 MB/s for the Pentium and 4.0 MB/s on the Athlon. The difference is attributed to the direct I/O driver eliminating the output buffer copy.

The time spent in the driver vs. the time spent in the benchmark (see Figures 12 and 13) affirms that the benchmark itself has little impact on the throughput. Its only overhead is a call to DeviceIoCtrl(); the remaining part of the transfer is handled by the I/O Manager and the driver.

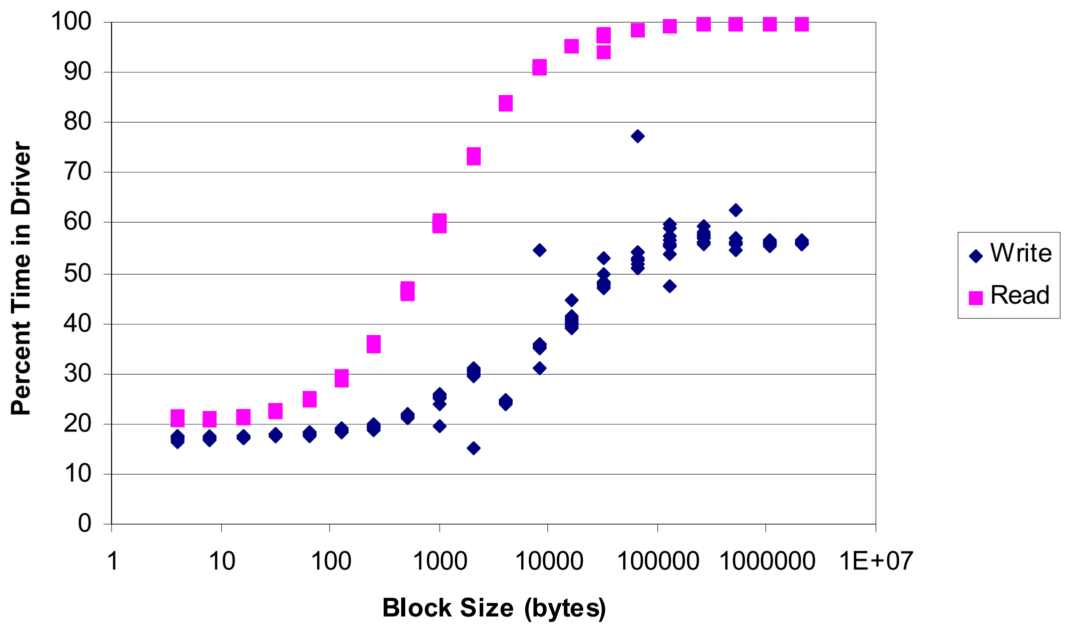

Figure 12: Direct I/O Driver Percent Time in Driver for Pentium

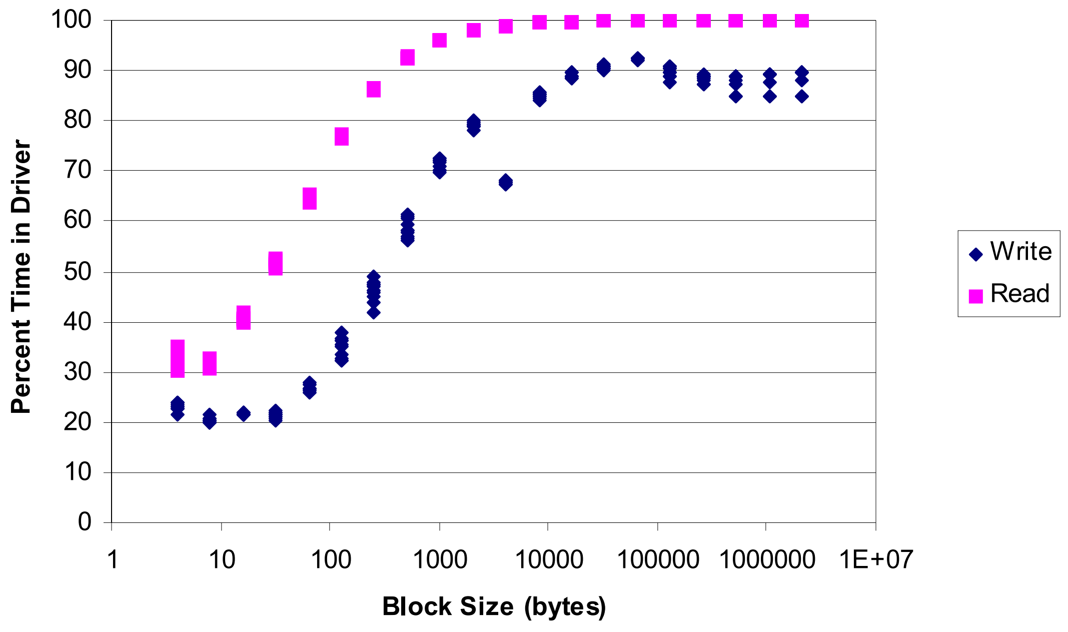

Figure 13: Direct I/O Driver Percent Time in Driver for Athlon

The Pentium time-in-the-driver ranges from 40 µs (1 byte) to 61 ms (2MB) for writes and 49 µs to 498 ms for reads. The Athlon time in the driver ranges from 2.7 µs to 74 ms for writes and 4.4 µs to 511 ms for reads. It is interesting that the Athlon is faster for the small transfers, but the CPUs have similar performance for the larger blocks. This is quite unexpected since the Athlon is an order of magnitude faster than the Pentium.

The implication is that software is the limiting factor for smaller transactions and hardware limits larger transactions. This assertion is supported by the fact that the Athlon processes small transaction an order of magnitude faster than the Pentium, which corresponds to the difference in their CPU speeds. As the transaction size increases, hardware becomes the limiting factor. This explanation is based on the throughput performance plateau experienced by both CPUs where they exhibit the same peak throughput (see Figures 12 and 13). However, the Athlon reaches this plateau when the block size is greater than 1024 bytes while the Pentium does not reach it until 10 kbytes. Again, we have an order of magnitude difference. The conclusion is that both CPUs are capable of outperforming some part of the PCI interface logic, presumably the North Bridge.

4.3.3 Buffered I/O Driver

As anticipated, the buffered I/O driver proved to be the slowest driver. This is illustrated by Figures 14 and 15. One interesting feature of these graphs is that the write throughput is the same for both CPUs and it is identical to the results for the direct I/O driver. This is to be expected since both drivers copy the user input into a kernel buffer. In addition, the maximum write throughput is similar to the peak write throughput of the memory mapped driver. This is significant since the memory mapped driver writes directly to the i960 while the buffered I/O driver has to copy the input buffer before writing to the target. This further supports the assertion made in Section 4.3.2 that write merge logic is employed in the PCI interface logic.

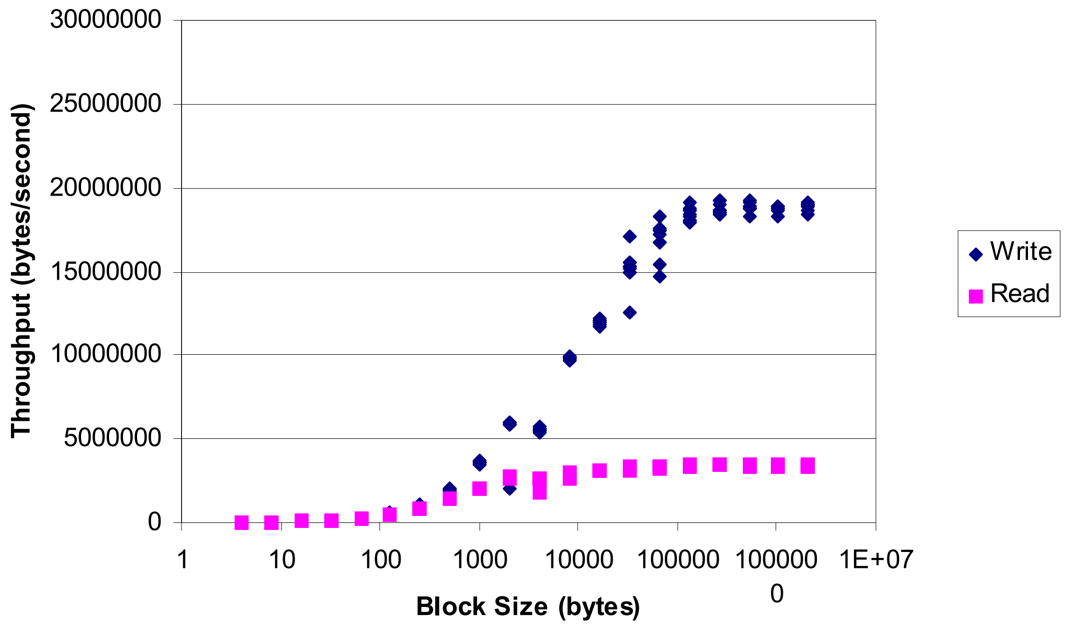

Figure 14: Buffered I/O Driver Throughput for Pentium

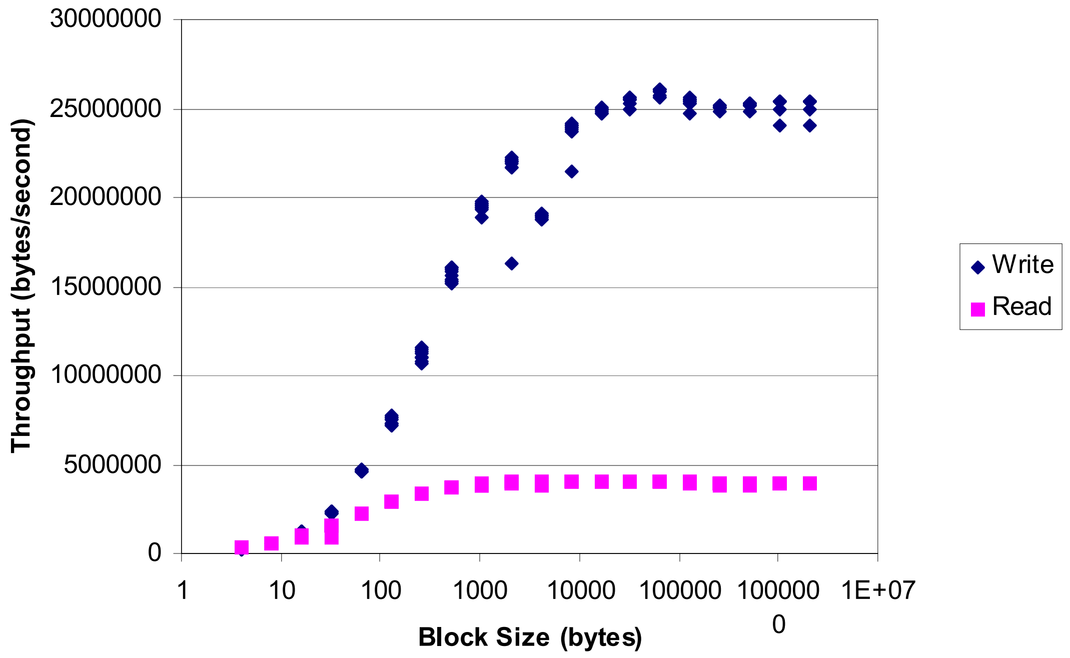

Figure 15: Buffered I/O Driver Throughput for Athlon

The read throughput for the buffered I/O driver is slightly lower than for the direct I/O driver. The typical read throughput for the buffered I/O driver was 3.4 MB/s for the Pentium and 4.0 MB/s for the Athlon. The direct I/O driver was approximately 4.1 MB/s for both the Pentium and Athlon. As discussed in Chapter 2, this inefficiency is attributed to the I/O Manager copying the received data from kernel memory into the user output buffer. This subtle performance degradation is not exposed by the driver time measurements because the copy occurs in the I/O Manager which is outside the context of the driver. The I/O Manager overhead is accounted for by the applications runtime.

The driver time results in Figures 16 and 17 reaffirm that the benchmark has minimal effect on the throughput. ?????????? Most of the CPU time is spent in the driver performing data transfers. This is consistent with the results for the direct I/O driver.

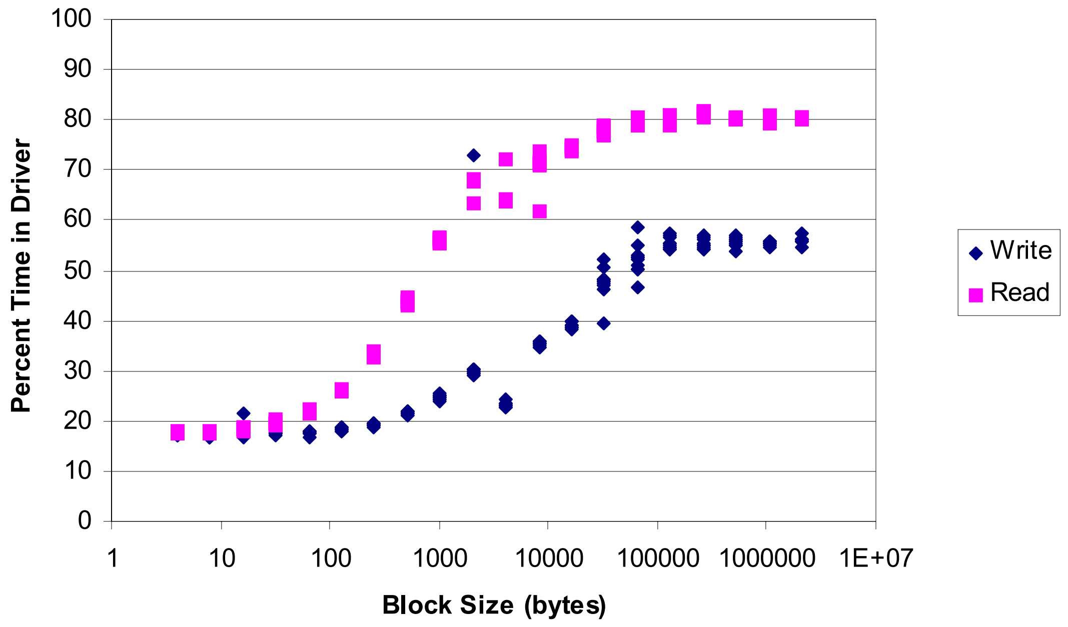

Figure 16: Buffered I/O Driver Time in Driver for Pentium

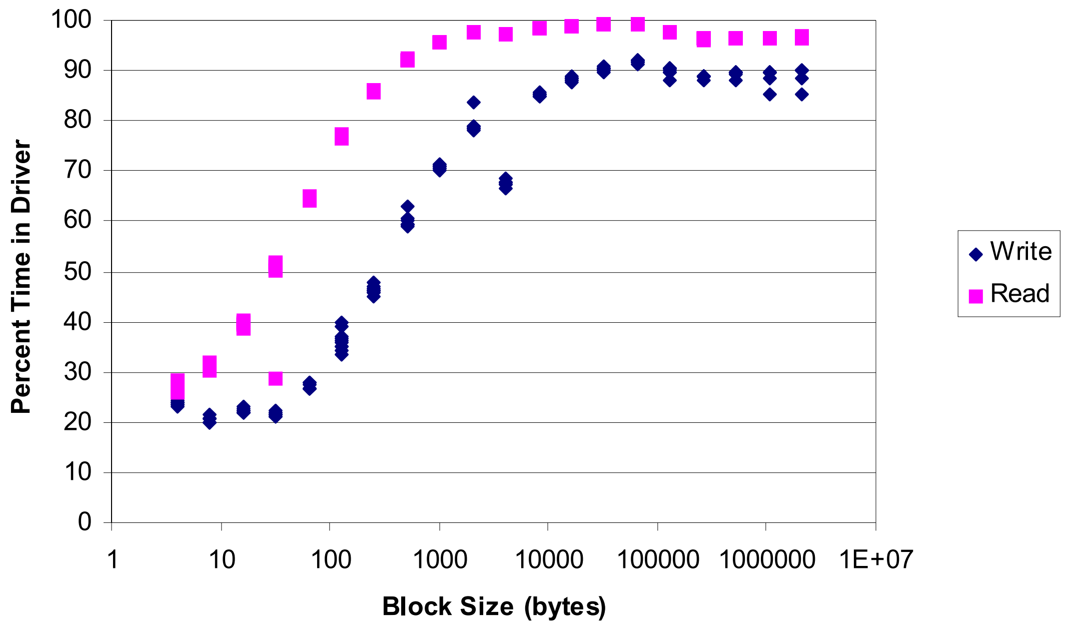

Figure 17: Buffered I/O Driver Percent Time in Driver for Athlon

The buffered I/O driver was modified to perform 8 bit data transfers instead of 32 bit transfers. The write throughput was unchanged, further supporting the assertion that write merging logic is present in the PCI logic. Read throughput dropped by a factor of four, as expected, indicating that the i960 is not prefetching read data.

4.3.4 Overall Driver Test Analysis

At a high level, the driver tests share several common results. First, there is no significant performance difference among the different driver implementations. This was completely unexpected. It was anticipated that the memory mapped driver would provide the best performance.

Second, the maximum data throughput is significantly less than the peak PCI throughput. This is to be expected since the peak PCI throughput measurement in based on using dedicated hardware with no OS overhead. However, the degree of performance degradation was severe even in comparison to the PCI Pamette [10]. In a system configured similarly to the Pentium, the PCI Pamette had a PIO write throughput of 65 MB/s and a read throughput of 15 MB/s. This is a stark contrast to the 25 MB/s write throughput and 4.5 MB/s read throughput derived from the SCC driver tests.

Finally, it is interesting that the Pentium and Athlon have the same peak throughput despite the Athlon being ten times faster than the Pentium. This result is expected to an extent because the PCI bus--not the CPU--will eventually limit the data transfer rate. The unexpected result is that the CPUs consistently reached the same peak throughput at different block sizes independent of driver implementation. This is because the CPUs outperformed the PCI interface logic.

4.3.5 DMA Driver

The results from the SCC driver tests indicate that programmed I/O has severe performance limitations. The next logical step (which is beyond the scope of this thesis) is to develop a DMA based driver. The DMA controller would relieve the host CPU from transferring data a word at a time as well be able to consolidate data transactions into PCI burst bus cycles. The DMA controller and North Bridge designs would need to be examined to determine how to best use these capabilities. The issue is then determining how to force the DMA to use PCI burst cycles from within a WinNT system.

Theoretically, the DMA write throughput should be better than depending on write merging. The DMA controller is given a definite address, block size and a starting time. In contrast, write merging logic must buffer data, determine whether it can be merged, wait for a specified timeout period to ensure no more data is coming, and perform the write transaction. This overhead has a negative impact on the write throughput; however, it is significantly better than writing a word at a time.

The read throughput has the most potential for improvement by using a DMA driver. Most of the OS overhead experienced by the SCC drivers is eliminated since the CPU is longer be involved in the read transfer. In addition, the DMA controller could use PCI burst read cycles to reduce PCI bus overhead.

A few issues need to be explored before implementing a DMA driver.

1) The amount of time a device can own the PCI bus must be restricted. The system bus is shared resource and there are system critical components connected to it that must be serviced regularly.
2) The DMA controller allows the host CPU to continue executing after starting the transaction. This is only a performance gain if the CPU has work to perform.
3) Finally, the SCC driver tests demonstrate a point where throughput plateaus. This must have been caused by a hardware limitation in the PCI interface. It is possible that the DMA controller would encounter this same issue.

A possible alternative to using the host DMA controller to perform reads is to have the target support bus mastering. In this design, the host programs a DMA controller in the target to effect the read transaction. The target executes the data transfer and then signals an interrupt when the transfer is complete. This design completely circumvents WinNT and North Bridge interfacing issues. The test procedures and corresponding results from Moll and Shand [10] on the PCI Pamette seem to validate this design.

4.4 Library Test Results

The purpose of the Array Interface Library tests was to determine if the driver interface library implementation affected system performance. Static linked libraries and DLLs for both the C and C++ languages were tested using the buffered I/O driver to interface to the i960. The AIL tests were executed on both the Pentium and the Athlon for completeness.

Figures 18 though 25 illustrate the results from the AIL testing. The results depicted match the results in Section 4.3.3 for the buffered I/O driver benchmark, which does not use a driver interface library. This implies that there is no appreciable difference between library types, at least at the macro level, although there are known differences at the micro level.

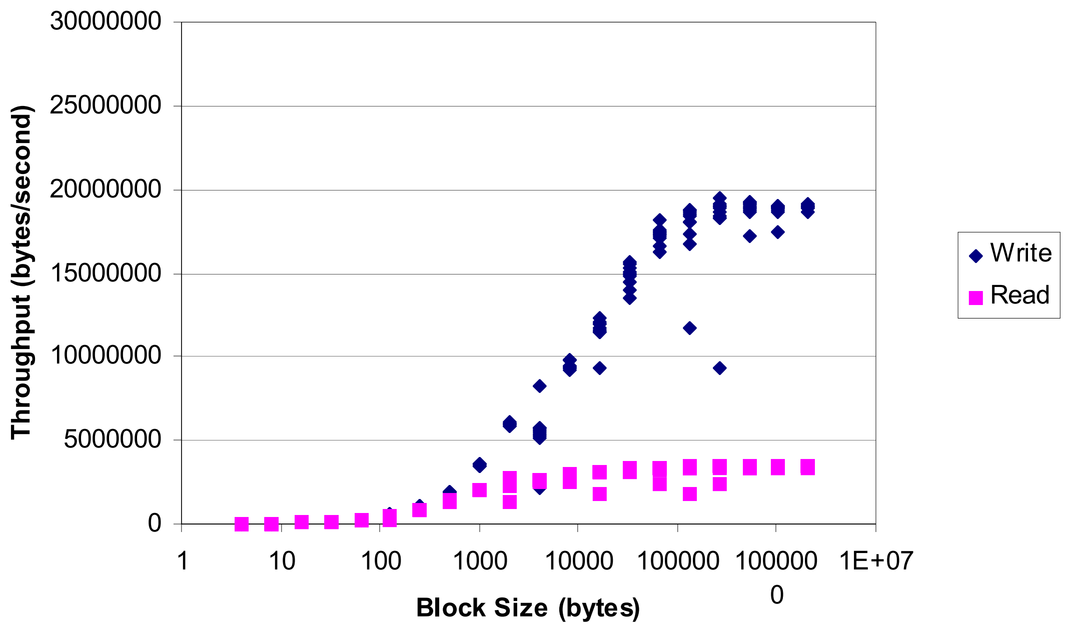

Figure 18: C++ DLL Throughput for Pentium

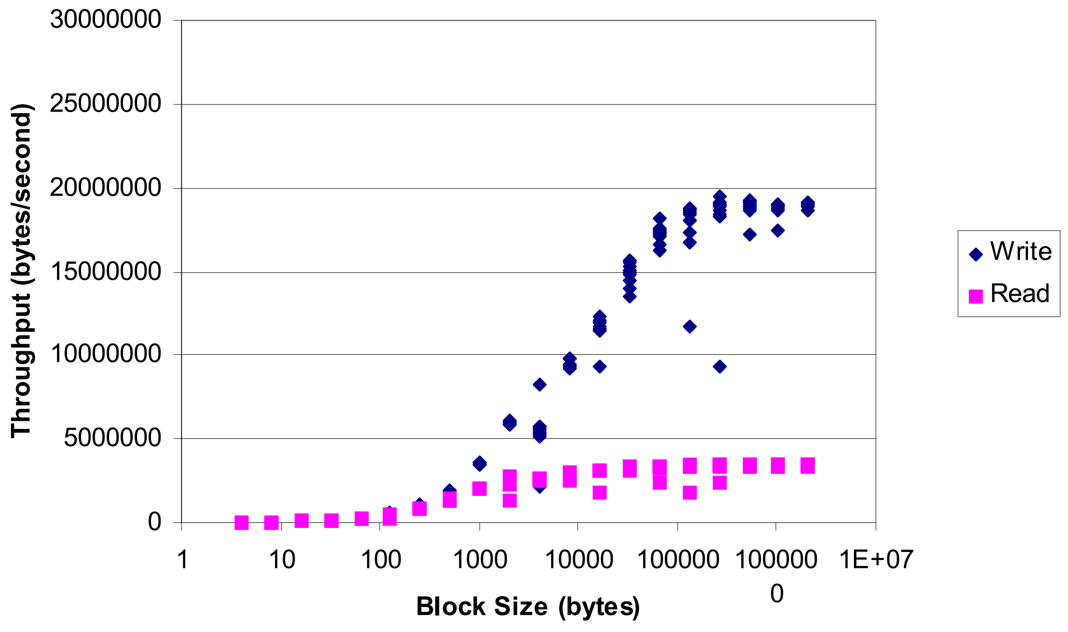

Figure 19: C++ DLL Throughput for Athlon

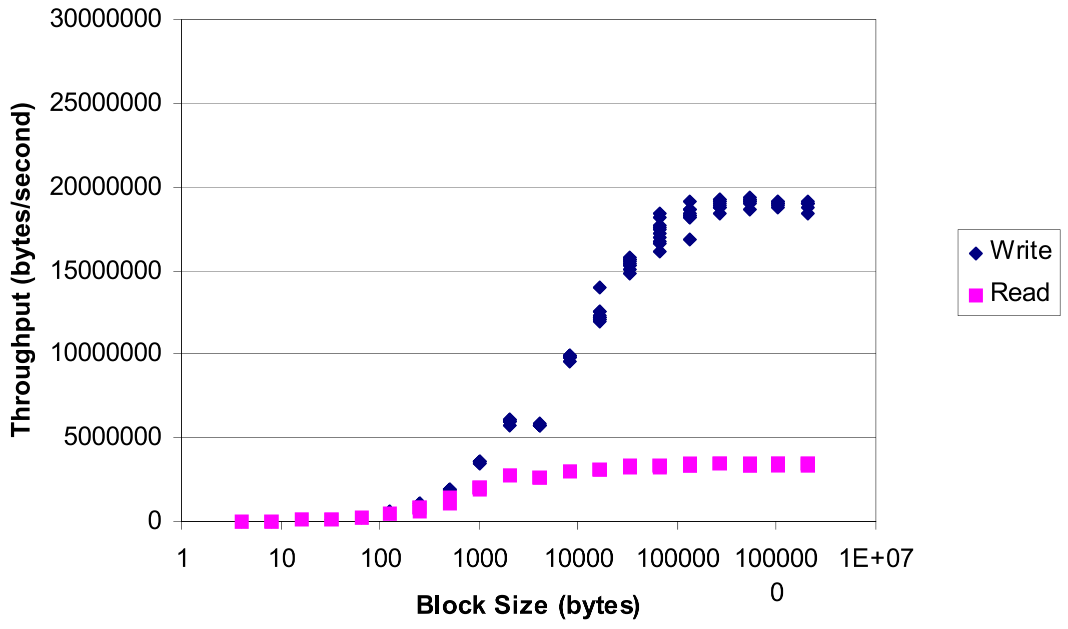

Figure 20: C++ Static Linked Library Throughput for Pentium

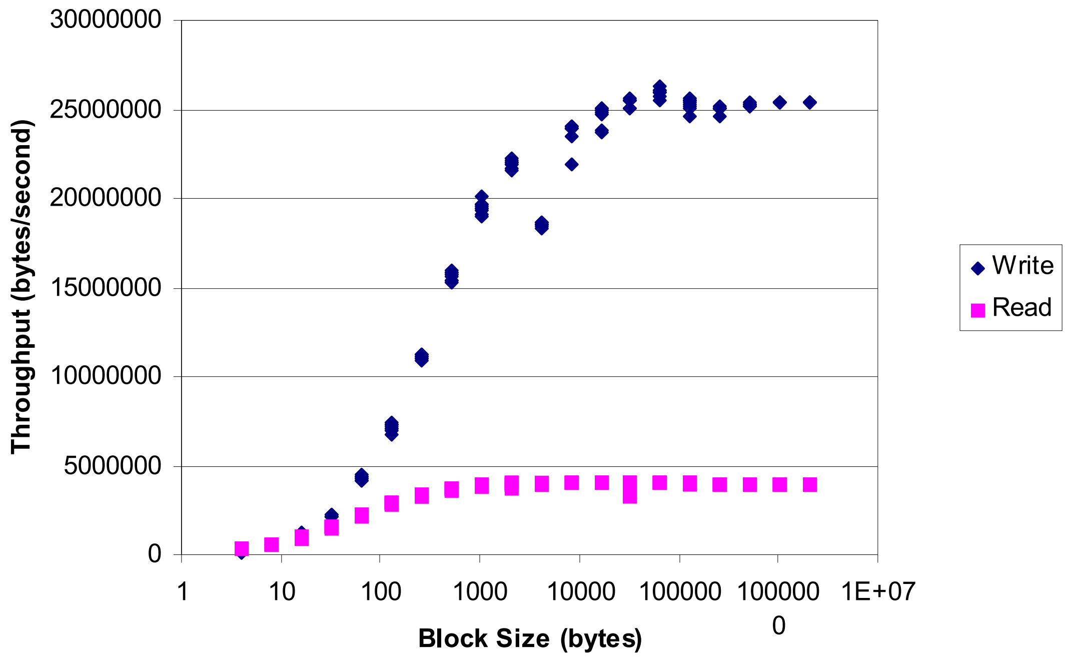

Figure 21: C++ Static Linked Library Throughput for Athlon

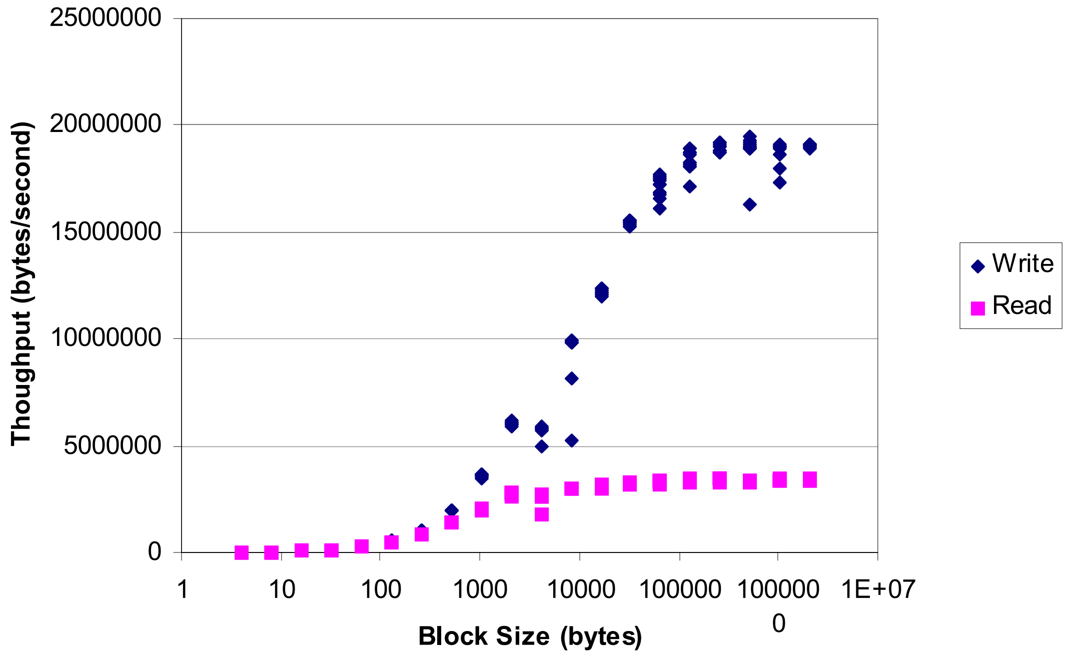

Figure 22: C Static Linked Library Throughput for Pentium

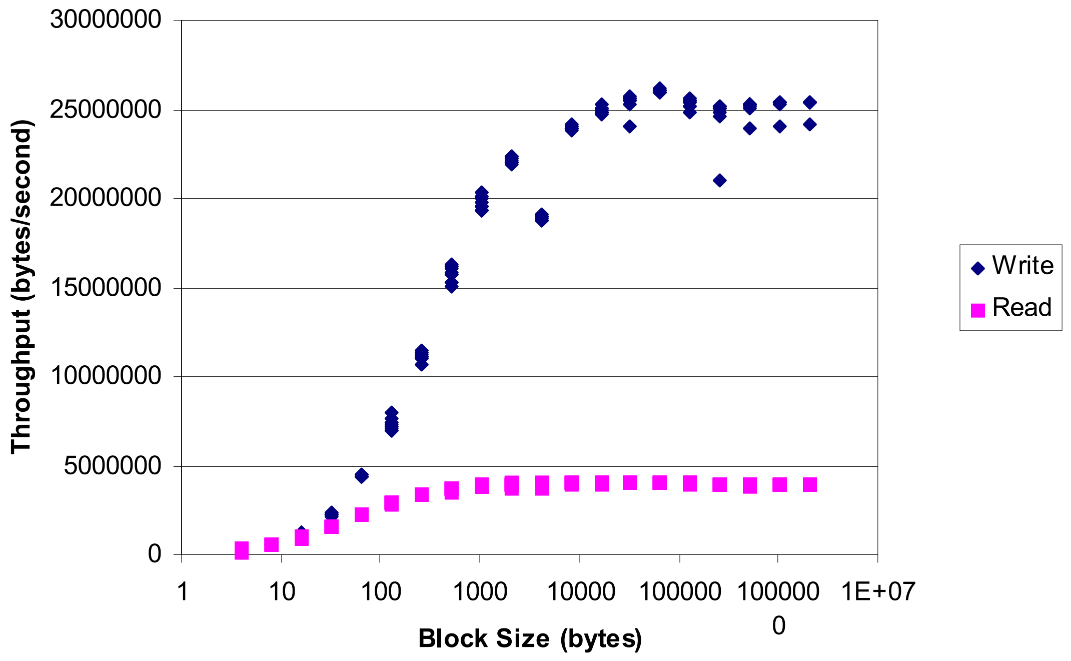

Figure 23: C Static Linked Library Throughput for Athlon

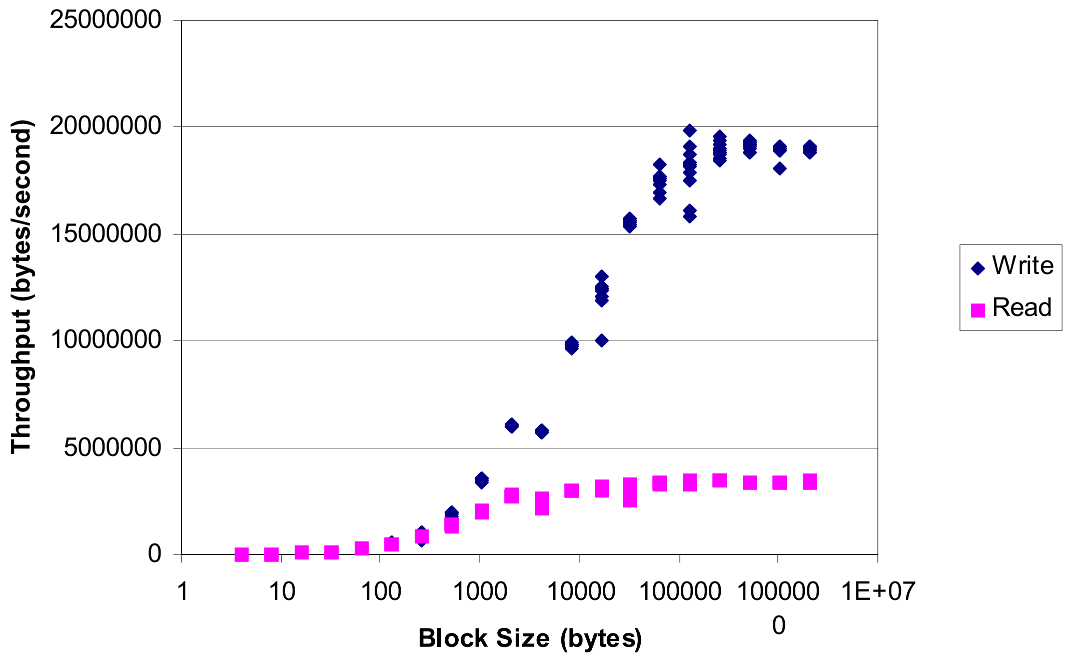

Figure 24: C DLL Throughput for Pentium

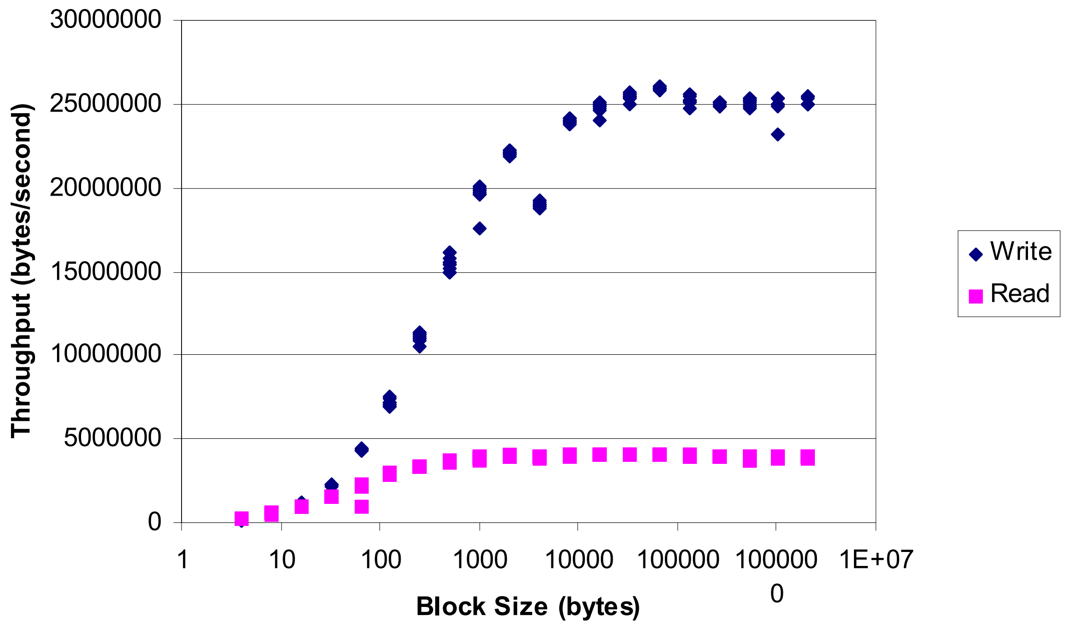

Figure 25: C DLL Throughput for Athlon

The AIL testing shows that driver interface library implementation has negligible impact on system performance. This is a positive result since it allows programmers to select the programming language and library types best suited to the application without incurring system performance penalties.

4.5 Bus Utilization Results

The purpose of the bus utilization test was twofold: to measure the fraction of time the PCI bus was used for SCC transactions and to determine how long the various block transfers take. Only the latter part of objective as met. Once testing was underway, it was concluded that the PCI bus analyzer traces were too detailed to analyze manually. The scope of the test was therefore limited to a single data set for each driver type with the block sizes ranging from 4 to 32 bytes. The Pentium system was used.

The times required to perform read and write transactions are shown in Figures 26 and 27. The transfer times can be compared to the theoretical PCI transfer time which is calculated as (1+Block_Size)*30 nanoseconds. This assumes a data word is transferred only every PCI clock cycle, which is unrealistic, but it does provide a baseline measurement. The graphs indicate that the actual transfer time is substantially greater than the theoretical PCI transfer time.

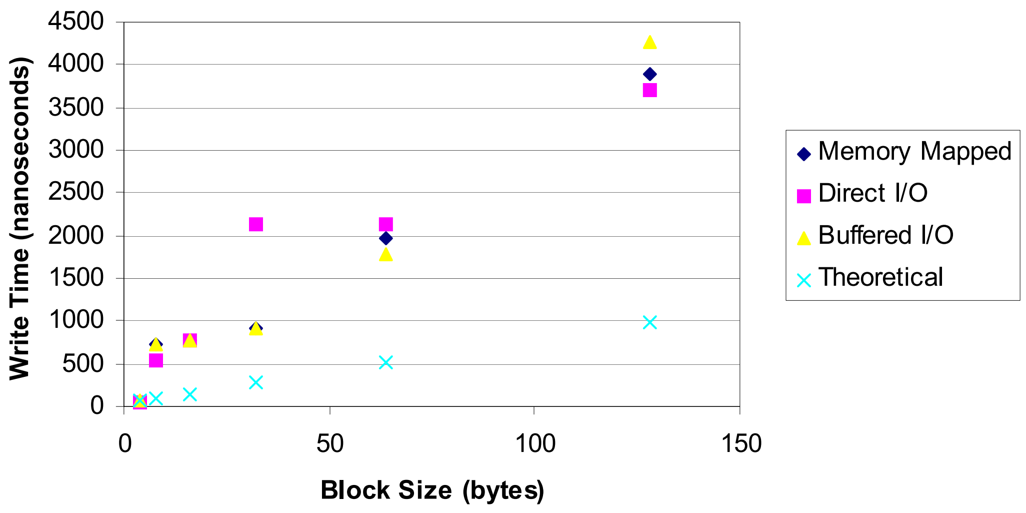

Figure 26: PCI Bus Write Time

The discrepancy between the actual and theoretical write performance illustrated in Figure 26 provoked a more detailed analysis of the logic analyzer traces. The host PCI interface uses PCI burst write cycles for data transfers larger than one word; however, it was found that the burst size is limited to nine words. It is presumed that this is done to avoid monopolizing the PCI bus.

The write trace analysis revealed that the i960 is negatively affecting throughput. It forces the host to retry burst writes an average of three times before the transaction completes successfully. This, combined with the host feature of using burst cycles for writes of over one word, leads to a write time of approximately 720 nanoseconds for a two word write, as opposed to the 128 nanoseconds the write should take. The effects of the retry overhead decrease as the burst length increases, but the host limits the burst to only nine cycles.

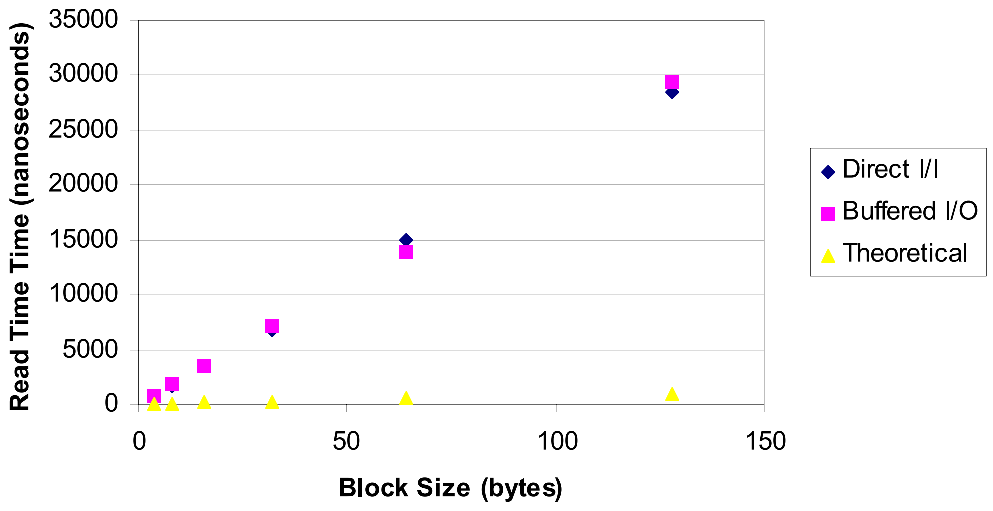

Figure 27: PCI Bus Read Time

Investigation of the logic analyzer traces further exposed the read performance bottleneck illustrated in Figure 27. First, it was confirmed that there is no read merging by the host; this was expected. The host issues individual PCI read cycles for every word read by the host. The trace also shows that, at least for small blocks, the reads are sequential on the bus; i.e., no other device interrupts the string of reads. This gives the illusion of being a PCI burst read, but it is not. In fact, a PCI burst read would be twice as fast as a string of individual reads because it only issues the address once during the address phase while the individual read generates an address cycle for every transaction.

The i960 adds an additional component to the sub-optimal read performance. It forces the host to retry reads an average of three times for every read transaction attempted. This makes the typical read time for a single word 816 nanoseconds or approximately 27 PCI bus clocks. The i960 performance problem can probably be attributed to one of two sources. First, the i960 DRAM access time may not be fast enough to respond in one clock cycle. This can be alleviated by installing faster memory. Second, the i960 effectively dual ports its DRAM so it can be accessed from the PCI bus and the i960. The i960 is executing the debug monitor from DRAM so it is possible that the i960 is contending with the PCI bus interface for DRAM access, thus forcing the PCI interface to generate a retry.

Only the host results have a direct bearing on the SCC design since the SCC will have a high performance PCI interface. The idiosyncrasies introduced into the system by the i960 will not be present in the SCC. However, these results further support the need for a DMA driven system as presented in Section 4.3.5.

Unfortunately, this test did not provide the desired bus utilization metric; however, it did explain part of the performance degradation experienced by the benchmarks. When the i960 was selected to be the SCC simulator, the assumption was that the i960 could accurately emulate the SCC. This assumption was proven false. These test results also identify pitfalls to be avoided in the SCC hardware design.

5 Conclusion

5.1 Discussion

The results are admittedly different than anticipated. The original hypothesis was that the memory mapped driver would have a significant performance advantage over the other two drivers. As the driver testing showed, the memory mapped driver had no advantage over the buffered I/O driver. This is completely counterintuitive and had these test not been run, the belief would still hold. Since there is no performance penalty, the better design choice is to use the direct I/O driver, which allows the NT I/O Manager to protect the kernel memory from errant memory accesses. It is important to note that the SCC memory is still mapped into kernel memory for all driver types. The manner in which data is passed to the driver is independent of this mapping.

At a macro level, the choice of library implementation was discovered to have no impact on system throughput. This is contradictory to the second hypothesis that DLLs and C++ code add overhead to applications. At the micro level, these programming methods generate a small amount of code overhead; however the test results proved this overhead to be negligible. In the case of DLLs, the results make sense. Their only overhead is loading them into memory. Once loaded, they function more or less identically to statically linked libraries. The same result vindicates the much maligned C++ language as well as since no measurable difference was detected between the C++ and C benchmark implementations.

Finally, the most surprising result is that the PCI throughput from the application perspective is significantly less than the PCI theoretical 132 MB/s peak. This is attributed to operating system overhead and the shared nature of the PCI system bus. The multitasking, preemptive nature of WinNT has the negative side effect that no application or device driver can be guaranteed exclusive access to the CPU or system bus. The best case is that the driver has uninhibited access to the system bus and the performance degradation is due solely to the WinNT Scheduler occasionally executing. This is only possible in a lightly loaded system. In the worst case scenario –and the more realistic- WinNT drivers will be preempting each other in response to repeated interrupts, the VM Manager will be paging to the hard drive over the PCI bus, and high priority tasks will preempt the user application. This latter scenario more closely models the environment in which the test results were collected.

As it turns out, the hypotheses posed in Chapter 4 were proven false. The implications of the test results are that the PCI bus throughput is seriously hampered in the proposed host environment and that the proposed software solutions are inadequate to solve the problem. The issue seems to stem from the nature of WinNT itself. It is a multitasking environment that must coordinate the use of all host system resources. This flexibility comes at the cost of system performance.

The main contribution of this thesis is a generic protocol for evaluating OS overhead on system performance. The testing methodology is flexible enough to be applied to other OS's and bus architectures. The significance of this is that most related work focuses on optimizing the system from outside the host; no effort was placed on evaluating the host itself. Their overall performance could be enhanced by applying the insights derived from this research.

Another contribution of this research effort was to create a generic programming environment for accessing devices on the PCI bus. Though the work was done in the context of the SCC project, there are no constraints to keep it from being expanded to support additional devices. The AIL allows SCC applications to be ported easily to new platforms and/or OS's; only the device driver (which is inherently OS dependent) would need to be reworked.

5.2 Future Work

The ideas and results presented by this research pose several new questions that could serve as the basis for future investigation. First, the test procedures/ methodologies presented are architecture independent. A natural extension to this research is to test other candidate host platforms, interconnect buses, and OS's. This would quantitatively identify optimal coprocessor host platforms. Second, an important piece of work paramount to using the proposed system is to augment the SCC compiler to use the PCI support developed here. This compiler would have to generate optimized code streams for both the host CPU and the SCC array. It would also have to schedule the loading of basic blocks and manage memory usage. Finally, it would be interesting to develop a method of networking SCC hosts. This would allow the SCC to be used to solve larger problems. A potential starting point would be to use Parallel Virtual Machine (PVM) developed for creating a MIMD network from UNIX workstations or the Resource Manager concept proposed by Jean et al. [12].

6 References

[1] Herbordt, M.C.; Cravy, J.; Honghai Zhang; Lin, C.; Hong Rao
Control for High-Speed PE Arrays
In Proceedings of IEEE International Conference on Application-Specific Systems, Architectures, and Processors, 2000. Pages 247-257.

[2] Peter G. Viscarola and W. Anthony Mason
Windows NT Device Driver Development
Copyright 1999 Open Systems Resources, Incorporated
Published by Macmillan Technical Publishing

[3] Tom Shanley and Don Anderson
PCI System Architecture, Third Edition
Copyright 1995 Mindshare, Incorporated
Published by Addison-Wesley

[4] David A. Solomon
Inside Windows NT, Second Edition
Copyright 1998
Published by Microsoft Press

[5] Hans-Peter Messmer
The Indispensable PC Hardware Book, Third Edition
Copyright 1997
Published by Addison-Wesley

[6] Edward N. Dekker and Joseph M. Newcommer
Developing Windows NT Device Drivers, A Programmer's Handbook
Copyright 1999
Published by Addison-Wesley

[7] Mink, A.; Salamon, W.; Hollingsworth, J.K.; Arunachalam, R.
Performance Measurement Using Low Perturbation and High Precision Hardware Assists
In Proceedings of the19th IEEE Real-Time Systems Symposium, 1998. Pages 379-388.

[8] Houzet, D.; Fatni, A.
Image Processing PCI-based Shared Memory Architecture Design
In Proceedings of 1997 Fourth IEEE International Workshop on Computer Architecture for Machine Perception, 1997 (CAMP 97). Pages 244-252.

[9] Cloutier, J.; Cosatto, E.; Pigeon, S.; Boyer, F.; Simard, P.
VIP: an FPGA-based processor for image processing and neural networks
In Proc. MicroNeuro, pages 330-336, 1996.

[10] Moll, L.; Shand, M.

Systems Performance Measurement on PCI Pamette
In Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines. Pages 125-133, Napa, CA, April 1997.

[11] Harbaum, T.; Meier, D.; Prinke, M.; Zitterbart, M.
Design of a Flexible Coprocessor Unit
IEEE Communications Magazine, 35(1). Pages 80-86, January 1997.

[13] Jean, J.; Tomko, K.; Yavagal, V.; Shah, J.; Cook, R.
Dynamic Reconfiguration to Support Concurrent Applications
In IEEE Transactions on Computers, Vol. 48, No. 6. Pages 591-602.

[13] PC 99 System Design Guide, A Technical Reference for Designing PCs and Peripherals for the Microsoft® Windows® Family of Operating Systems
Copyright 1998-1999 Intel Corporation and Microsoft Corporation
http://www.microsoft.com/hwdev/pc99.htm

[14] Herbordt, M.C.; Cravy, J.; Honghai Zhang; Lin, C.; Hong Rao
An array control unit for high performance SIMD arrays
In Proceedings of the Fifth IEEE International Workshop on Computer Architectures for Machine Perception, 2000. Pages 293-301.

Appendix A: SIMD Coprocessor Card (SCC)

The SIMD processing model is best suited for massively data parallel applications with relatively high compute to I/O instruction ratios. Candidate applications include computer vision, graphics, and DNA genome string matching. The power of this architecture can be seen when the number of processor elements (PE) is equivalent to the size of the data set. In this situation, component-wise operations on parallel variable elements can be done simultaneously. Even in the situation where the number of elements in the parallel variable is greater than the number of PE's, the speedup, compared to a sequential algorithm, is immense.

The SIMD architecture contains a single control unit with multiple PE's. The PE's are slaves to the control unit and cannot fetch or interpret instructions. The PE's are basically arithmetic logic units (ALU) capable of performing logic, arithmetic, and data transfer operations within their own memory space. They all perform the same operation in lockstep under the direct control of the control unit. Inter-PE communication is coordinated by the control unit as well. The advantage of this type of architecture is in the ease of adding more memory and PE's to the system. The disadvantage is the computing overhead of the control unit managing memory exchanges.

This appendix discusses several issues associated with the SIMD architecture and poses potential solutions. It then presents the design for the SIMD Coprocessor Card (SCC) based on work performed by Herbordt et al. [1].

A.1 SIMD Issues

SIMD PE's can be built with operating frequencies in excess of 1 GHz [1]. The main problem is keeping them busy, i.e., the host must determine which instructions to issue and issue them faster than the array executes them. The inherent mismatch between host and array execution frequencies makes it difficult to maintain high array utilization: either the instruction issue rate is very low or PE data locality is compromised. The PE data locality issue is further compounded by the fact that the data working sets for most useful SIMD applications exceed the physical resources provided by the PE array. The following issues must be understood and addressed in order to optimize the performance of SIMD systems:

1) Instruction distribution latency. The host must be able to issue instructions to the PE array at a sufficient rate to maintain high utilization. Otherwise, performance advantages are lost because of array idle time. This issue also pertains to data transaction targeting array memory.
2) Instruction issue latency. The host executes the main thread of the SIMD application. It controls program flow based on feedback from the array creating control hazards.
3) Application working set mapping to physical PE's or tiling. Data and instruction locality are a key issue.

A.1.1 Instruction Distribution Latency

Assuming that the array data needs can be serviced by a standard cache/ main memory hierarchy, the next challenge faced by SIMD systems it to maintain high array utilization. PE's can have operational frequencies up to 1 GHz [1] so the array host instruction issue frequency must be comparable or all performance improvements are negated.

The first approach would be to have a standard PC host issuing instruction directly to the array using an industry standard bus. The drawback to this approach is that the array would be idle most of the time since the bus bandwidth is significantly less than the PE execution frequency. Table 1 lists the bandwidth for the most common PC buses.

A significant improvement to the previous approach would be to have the host issue macro instructions that would be expanded in hardware to array micro instructions. This type of instruction expansion has been previously used and was shown to significantly reduce the bus bandwidth required for instruction issue. This effectively decouples the array and host operating frequencies. The problem with this solution is it still relies on the host issuing one instruction at a time which could leave the array idle if the macro instruction expansion ratio is low. In an ideal system, the host would determine which instruction to issue before the array completes the previous operation. This scenario is unlikely given the asymmetric, multithreaded nature of SIMD applications in which the host must issue instructions based on feedback from the array. Assuming a 1 GHz array and 1:10 macro instruction expansion ratio, the host would have to issue instructions at a rate of 100 MHz. None of the standard system buses in Table 1 can support this bandwidth requirement. Even a 1:50 expansion would require 20 MHz of system bus bandwidth which only PCI and VESA can easily accommodate. These measurements neglect to account for OS and/or application overhead. The instruction I/O bottleneck must be removed for the SIMD system to achieve optimal performance.

A.1.2 Instruction Issue

Given sufficient data availability and instruction distribution rates, the next SIMD issue is for the host to determine which instructions to execute. In the SIMD programming model, the host executes the main SIMD application thread so it must execute a certain amount of serial code to control program flow and to exchange data with the array. The issue is the array will idle while the host is evaluating branch conditions or computing scalars, i.e., data and control hazards exist.

Control hazards are created when the host must wait for runtime feedback from the array before issuing the next instruction. The one solution is to stall the array until the branch condition is evaluated. However, the exact stall time can be indeterminate if the host OS is multitasking and/or the array interconnect bus is multiplexed with other devices. The problem could be ameliorated by adding an instruction pipelining and cache to the PE's, but the control hazard is not completely removed. If there is no change in program flow, program execution is improved. However, if a branch is taken, the cache must be flushed and the instructions reloaded leaving the array idle. Also, a negative side effect of adding the cache and pipeline is reduced PE count per IC.

Scalar computations can lead to data hazards that manifest themselves in two distinct manners. In the first case, the host stalls waiting for feedback from the array. This really is not a significant problem as long as the feedback is not required for the array to continue operation. The second case is the array stalls waiting for data from the host. This is a significant issue as the exact amount of time required for the host to issue the scalar is indeterminate. The host compiler can help reduce the impact of this issue by scheduling the scalar calculation as early as possible in the instruction stream; however, this is only possible if the calculation is independent of feedback from the array.

One can attempt to apply standard compiler techniques to resolve these issues, but the success is limited by data dependencies between the host and array working sets. Assuming these issue can be resolved or at least minimized, SIMD designs must still contend with the issue of an application requiring more hardware resources than are physically available.

A.1.3 Tiling

The SIMD programming model abstracts the array hardware by providing Virtual PE's (VPE). The assumption is there is one VPE for each data element comprising the application data working set. In reality, this is not recognizable in hardware because highly data parallel applications like graphics or vision would require several thousand physical PE's. Not to mention the fact that the number of PE's needed to solve a problem is application specific. The solution is to map the data set such that each physical PE operates on several slices of a parallel variable called tiles. A major negative consequence of this approach is most data locality within the array data stream is lost. The extent of the performance penalty is determined by which tiling method is employed. The following examples illustrate this point. The following examples assume the array control unit performs macro instruction expansion.

The first method (Tiling Method #1) executes all instructions on all tiles before proceeding to the next instruction. The process represented in for loop notation is:

```
FORALL macro instructions
    FORALL tiles
        DO microcode expansion
```

The strength of this method is its simplicity. The host issues instructions to the array and the array control sequencer does a Table lookup to expand the macro instructions into sequences of array instructions. The array controller also controls tile selection for VPE emulation. The major drawback to this method is that it destroys data locality. When the inner FORALL loop completes, further PE cache accesses are misses. This negates the performance enhancements provided by using cache and significantly degrades system performance since all data accesses hit external memory.

The second tiling method (Tiling Method #2) is to execute all instructions on the same tile before moving to the next tile. The process depicted using for loops is:

```
FORALL tiles
    FORALL macro instructions
        DO microcode expansion
```

This method improves system performance since data locality is preserved, but it has two major flaws:

1) Tiling must be controlled by host. This significantly degrades system performance. The array must stall after completing an instruction stream until the host swaps the tiles and restarts instruction issue.
2) The most significant problem is: This method does not work! Mainly because it ignores inter-PE dependencies and reduction hazards. The problem becomes more visible in fine grain parallel applications.

These issues, while significant, are not insurmounTable. SIMD has been successfully employed in numerous high performance applications. The key issue is that they may not be performing to their fullest potential.

A.2 SCC Theory

In an attempt to solve the SIMD architectural issues, Herbordt et al. [1] have proposed a SIMD design based on an Array Control Unit (ACU) inserted between the host and the PE array (see Figure 28) that address the previously mentioned SIMD issues. This design is hereafter referred to as the SIMD Coprocessor Card or SCC. In the SCC design, the host is relieved of array control responsibilities other than running the main thread of SIMD application. The design also allows code segments called Basic Blocks (BB) to be preloaded into the array for execution.

A.2.1 ACU Concept

The purpose of the ACU is to handle macro instruction expansion, data tiling and PE control. The ACU effectively decouples the host operating frequency from that of the PE array by allowing the host to issue fewer instructions and by transparently swapping tiles. The ACU functionalities are completely implemented in hardware which allows it to operate at the same speed as the PE array. However, the host interface speed is fixed by the interconnect bus bandwidth. Alternate techniques must be applied to speed up this interface. A basic block diagram of the SCC is depicted in Figure 28.

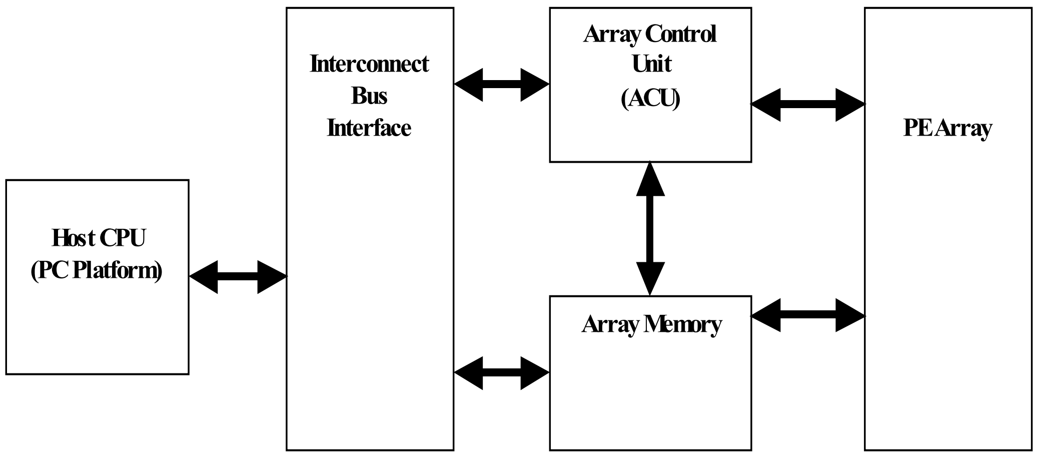

Figure 28: Block Diagram

The first interesting feature of this design is that there is no operating system running on the ACU; its functionality is completely hardwired. The significance of this is that there is no OS overhead to affect array performance and that the ACU operates at the same frequency as the array. Instead, the ACU operation is controlled by special array instructions -directives- issued by the host as part of the instruction stream. The directives are interpreted and executed within the confines of the ACU, i.e., they are not issued to the array. The ACU design incorporates directives to configure the ACU, control program execution and to transfer data to the array. A side effect of this design is that ACU directives may be freely interspersed with PE macro instructions. This allows the ACU (and thus the PE array program) to be reconfigured on the fly and program execution to be modified without using interrupts.

A.2.2 Macroinstruction Expansion

There are several issues related to instruction issue that must be overcome to achieve optimal array performance. The ACU employs macro instruction expansion to improve host to ACU communications bandwidth by reducing the number of instructions issued. The macro instruction to micro instruction expansion ratio is important. The more micro instructions derived from each macro instruction, the better the system performance. This mechanism also has the added benefit of reducing the penalty associated with control hazards. For example:

```
IntPlane (2,2) A,B;  // Parallel variables.  Assume a 2x2
                     // PE array
UINT temp;           // Temp storage for result

If( B.ANY())         // Check for any nonzero element in
     A = 7 + B;      // parallel variable B
Else
     A = B – 7;
```

This simple code segment translates into the following compiler tuple segment:

```
(ANY,temp,B)         // Host tells array to check for
                     // nonzero elements in B
(CMP,temp,0)         // Host waits for feedback value from
                     // array
(JZ, tuple6)         // Host must decide which instruction
                     // to issue next.
(+,A,7,B)            // The array is stalled for as long as
(J,tuple7)           // it takes to make the decision.
Tuple6: (-,A,B,7)
Tuple7:
```

Assume that array B has nonzero elements so the A = 7 + B instruction is executed. The macro instruction sequence issued by the host would be:

```
LD R0,B            # Instructs PE’s to load elements of B
GOR R0             # Global OR reduction
STO temp,R0

NOP                # The array stalls at this point as the
NOP                # host determines which instruction to
                   # issue next
LD R0,B
LD R1,#7
ADD R2,R0,R1       # Perform addition
STO A,R2           # Store result
```

Finally, the micro instructions actually issued by the ACU to the PE array:

```
R0_0 <- B           # Get the lower byte of B
R0_1 <- B           # Get the upper byte of B
R1 <- R0_0 OR
R0_1
R1 <- CMP (R1, 0)  # Check for nonzero data. Reduction
                    # operation.
temp_0 <- R1_0      # Store lower byte of result
temp_1 <- R1_1      # Store upper byte of result

NOP                 # The array stalls at this point as the
NOP                 # host determines which instruction to
                    # issue next.

R0_0 <- B           # Get the lower byte of B
R0_1 <- B           # Get the upper byte of B
R1_0 <- 7           # Get the lower byte of immediate data
R1_1 <- 0           # Get the upper byte of immediate data
ACC <- R0_0 +
R1_0
R2_0 <- ACC
ACC <- R0_1 +
R1_1
R2_1 <- ACC
A_0 <- R2_0         # Store lower byte of result
A_1 <- R2_1         # Store upper byte of result
```

If the host had to issue micro instructions, it would have to execute 16 instruction cycles not including the delay to evaluate the branch condition. In contrast, if the host issues macro instructions, it would only execute seven instruction cycles not including branch. Assuming a fixed instruction cycle time, it is intuitive that the host macro instruction performance is twice that of the micro instruction method. This was contrived example, but assuming a bus speed of 66 MHz, 15.2 nanoseconds were saved by not transferring the 11 extra instructions. This example did not account for OS overhead on the host. While difficult to quantify, host OS overhead can have a significant negative effect on the array.

A.2.3 ACU Control Directives

In order to maintain high-speed operation, all of the ACU functionalities are implemented in hardware. The implication of this is there is no operating system to control the array operation so the ACU must assume these responsibilities. Five special instructions (or directives) have been defined to control the operation of the ACU. The directives can be issued individually by the host or interspersed with the code stream. Table 4 describes the supported ACU directives.

Directive	Operand 1	Operand 2	Description
CONFIGURE	Number of tiles	Tile size in bytes	Instructs the ACU to configure the data access registers.
BASICBLOCK	Start address	N/A	Instructs the ACU to load Operand1 into the PC and start execution at that address.
IMMEDIATE	Data value	N/A	Instructs the ACU to fetch an immediate data value from the INFIFO.
FEEDBACK	N/A	N/A	The array is instructed to perform a reduction operation. The result is copied into the OUTFIFO.
SINGLESTEP	Array instruction	N/A	This directive is intended to system debugging. It instructs the MFU to fetch a single instruction from the INFIFO and execute it.

Table 4: ACU Directives

A.2.4 ACU Tiling

The ACU transparently controls tile swapping based on Tiling Method #3 which presented in the next section. The concept is not so different from the code relocation register concept employed by x86 based systems. The ACU uses three registers to keep track of tiles. The first two registers are the Tile Count Register (TCR) and the Tile Size Register (TSR). The TCR defines how many tiles are contained in a parallel variable while the TSR defines the number of elements in each tile. The third register is the Start Address Register (TSAR). It is initialized at runtime to the start address of the parallel variable to be operated on. At the completion of each instruction sequence, the TSR is added to the TSAR to compute the start address of the next tile. This operation is repeated TCR times until the instruction sequence has been applied to all tiles.

A.2.5 Basic Blocks

The SCC incorporates another system level performance enhancing feature. It applies the compiler concept of Basic Blocks [1] to further reduce the host/array bandwidth requirements. In this context, the term Basic Block refers to a sequence of PE instructions that can be executed within a tile before execution on a new tile must be initiated. The tile swapping is generally the result of an inter-PE communication or feedback operations.

The concept is to preload sequences of macro instructions (Basic Blocks) into array memory before they are required. This instruction caching approach allows BB's to be loaded while the interconnect bus would be otherwise idle and assures the array has a steady source of instructions accessible from high-speed memory. The host only has to issue Execute ACU directives at runtime to instruct the ACU where to begin execution. The preloading BB approach has several benefits:

1) The host can schedule the loading of BB's so the array will not stall waiting for instructions.
2) The host only has to run a single instruction cycle (send the Execute ACU directive) to execute a BB. This significantly reduces the runtime bus bandwidth requirements.
3) An offshoot of benefit 2 is that the BB's are easily re-executed as long as the host has not swapped the BB out of array memory.

4) Finally, the control hazard penalty is reduced. All of the BB's associated with a branch can be preloaded. The host only has to issue the appropriate execute ACU directive at run time after the branch condition is evaluated.

The BB concept gives rise to a third tiling method that addresses the issues present in Section A.1.3. In this tiling method (Tiling Method #3), all instructions are executed on a tile before it is swapped. This is similar to Tiling Method #2 in that data locality is preserved. The difference is the instruction sequence is broken into BB's by the compiler, i.e., instructions are executed on a tile until a communication or feedback operation is required. This is depicted as follows:

```
FORALL basic blocks
    FORALL tiles
        FORALL macro instructions
            DO microcode expansion
```

This addresses the issue of communication/ reduction hazards. It also has the same performance advantage as Tiling Method #1 where all instructions are executed on a tile before it is swapped. Tiling Method #3 has receives an added performance boost by not involving the host in tile swapping; this is done transparently by the ACU.

A.2.6 Software Model

The interesting point about the BB software model is that it works for both SIMD and MIMD systems. It has been previously shown to work for SIMD, but the fact that it works for MIMD is less obvious. In an MIMD implementation, The SCC architecture is modified to provide one PE per ACU; effectively creating a network of MIMD processors. The rest changes are limited to the compiler. It is the compiler's responsibility to schedule loading and execution of BB's based on data dependencies.

BB's are by definition independent of each other so they can be preloaded into different ACU/PE pairs and executed on demand as long as there are no data dependencies.

A.3 SCC Implementation

The coprocessor card as presented by Herbordt et al. [1] consists of an FPGA baseboard connect to a host via the PCI bus. The host platform is a standard PC. In order for the SCC design to be viable, it must have a host system to provide I/O capabilities and a software system to allow it to interface to the host. A system level block diagram of the SCC system design is shown in Figure 29.

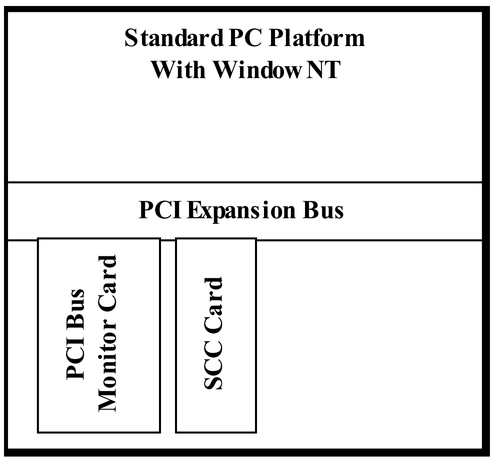

Figure 29: SCC System Level Block Diagram

A.3.1 Host Platform

Industry trends dictate that the SCC host be an Intel x86 based PC with a PCI expansion bus running the Microsoft NT operating system. This host platform was selected so that the SCC can be easily integrated into standard PCs. However, the SCC can be utilized in any host that uses the PCI bus assuming the proper software support is available. The PC host also provides the functionalities required for SCC control and I/O operations eliminating the need for costly custom hardware.

PCI was selected for its high bandwidth, ubiquitous presence in desktop PCs and most importantly, its acceptance as an open standard. PCI allows for system expansion since it can support eight cards without a bridge. This means multiple SCC cards can be installed in a single system to solve larger problems. The PCI burst mode data transfers can be used to move large data blocks with minimal system overhead. However, this is dependent on the North Bridge PCI implementation and not necessarily directly under software control.

The SCC memory, control registers and data queues appear as memory mapped registers on the PCI bus. Details of these registers are discussed in following sections. What is important to note is that the SCC memory and registers can be accessed as easily as any other PCI memory device. Memory mapped I/O will also reduce the system overhead required for user applications to access the SCC.

WinNT 4.0 was selected for its ubiquitous presence in the PC/ workstation arena. This high performance, multiprocessor enabled OS provides a sTable, extensible environment for hosting the SCC. Another positive effect of using a WINTEL host, is the plethora of development tools available. Compilers, debuggers and CASE tools are readily available; eliminating the need to develop tools suites for the SCC target.

The SCC design is implemented as a coprocessor card. The main reason for this is to avoid reinventing the I/O support systems provided by PC's. The PC's also provide an operating system environment for loading, executing and debugging user applications. SIMD systems are tuned for handling computation intensive applications. Their performance gains are realized while executing computationally intense algorithms; not performing I/O operations. The PC CPU will also be used to calculate scalars and evaluate branch conditions. It executes the main thread of the SIMD application. This frees the SIMD array from having to provide a dedicated processor for evaluating scalars.

A high-speed datapath is required between the host and the ACU. This is critical to achieving optimal array performance. It is equally important to observe industry standards. Therefore, the logic choice for the host/ array bus is PCI. PCI provides the highest bandwidth available on commercially available systems.

The coprocessor card concept itself has been proven successful by several research initiatives. For example:

- Houzet and Fatni [8] implemented the GFLOPS system for image processing.
- Cloutier et al. [9] implemented VIP: Virtual Image Processor.
- Harbaum et al. [11] implemented a reconfigurable computing card call FHiPPs based on the Intel i960 and FPGAs.

The driving factors cited in all cases were flexibility and low cost.

The coprocessor design lends itself to the possibility of easily expanding the SCC processing capabilities. It is possible to install multiple SCC cards in the same host. The number of SCC cards installed is limited by the number of PCI slots available in the host. Only the user application would have to be modified to take advantage of the additional SCC boards. It would also be possible to network SCC hosts together using standard LAN or WAN networking. This would require additional control software on the hosts, but this is not a significant issue as Harbaum et al. [11] has implemented a similar concept called the Hardware Manager in the FHiPPs project. It would also be possible to port the Parallel Virtual Machine (PVM) code from UNIX to WinNT to support networked SCC host.

A.3.2 Coprocessor Card Hardware

The hardware block diagram for the SCC card is shown in Figure 30. The SCC design is based on the Nallatech Ballynuey FPGA development board. The Nallatech board contains three Xilinx Virtex 1000 FPGAs which are programmed to implement the SCC system blocks depicted in Figure 31. The FPGA based implementation provides a high degree of flexibility for exploring different architectural concepts using the same hardware.

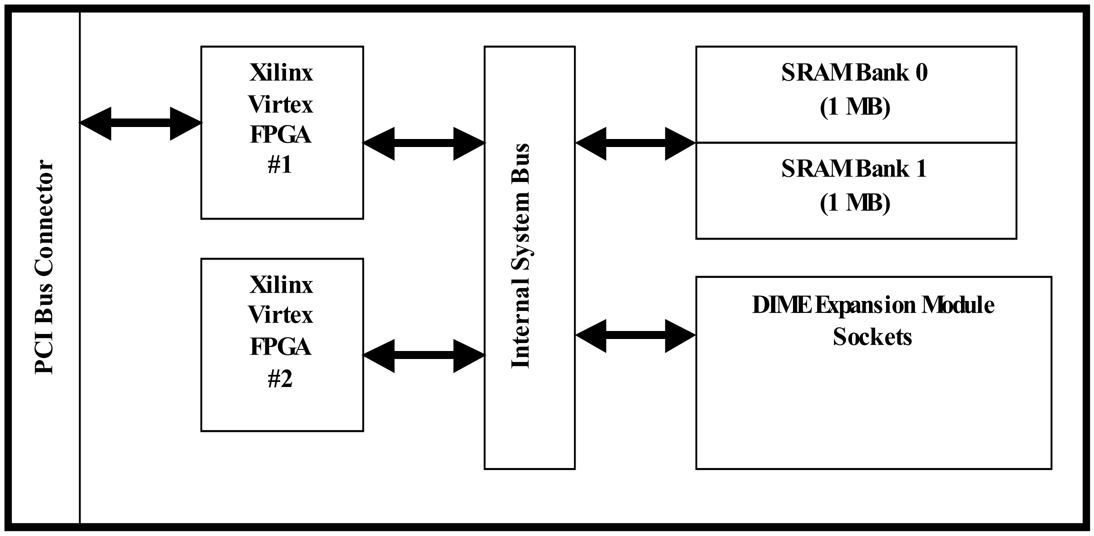

Figure 30: SCC Card Block Diagram

Ultimately, the FPGAs will be replaced with ASICs. This would allow the PE operating frequency to be increased along with the number of PE's per IC. This change would have no impact on the host or support software, as they are effectively isolated from the SCC hardware by the PCI bus.

The logic design for the SCC board FPGAs is shown in Figure 31. It consists of three main components: the PCI interface, the ACU and the PE array.

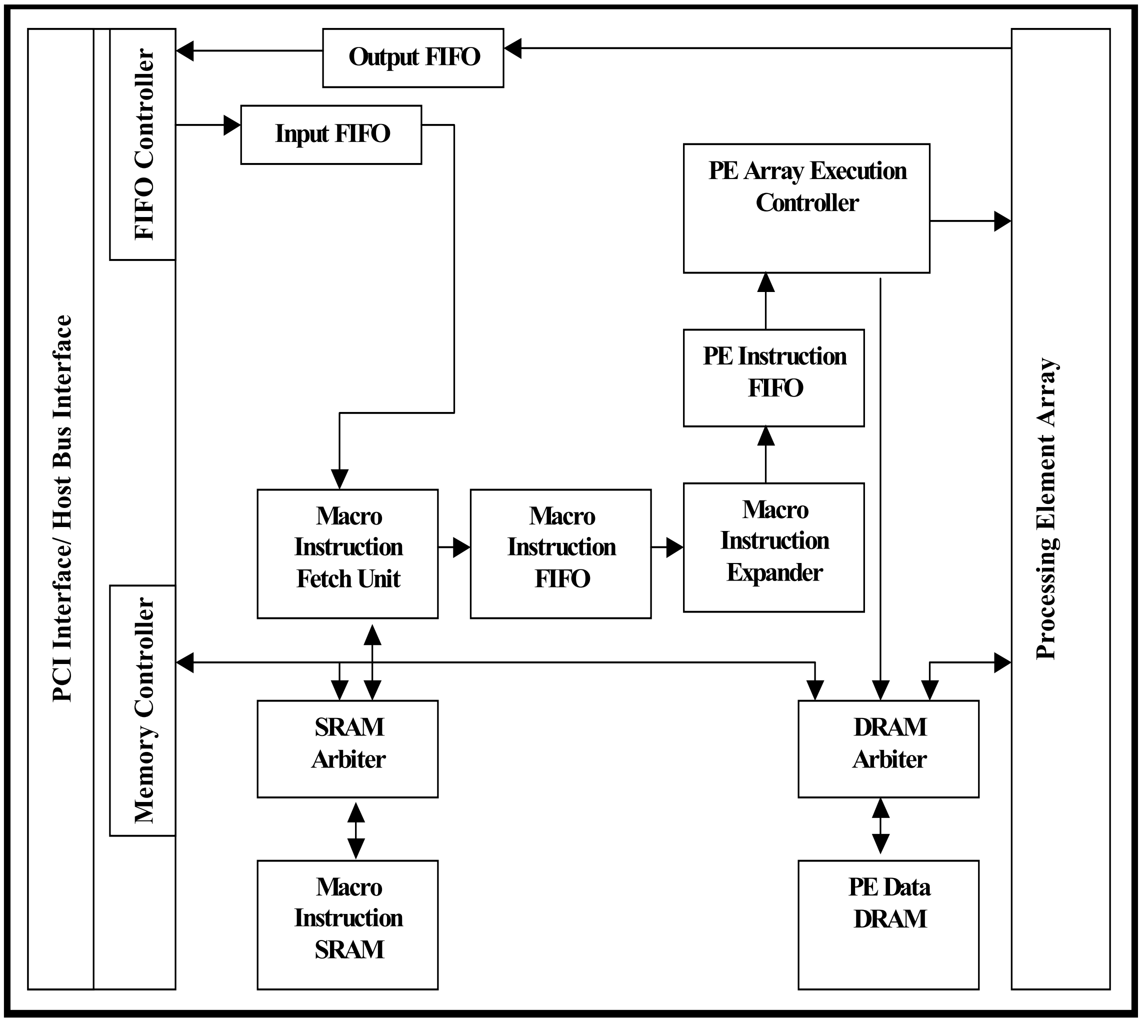

Figure 31: SCC Block Diagram

The first logic block is the PCI bus interface. The main purpose of this block is to provide the host access to the SCC high-speed macro instruction memory, data memory (DRAM) and the ACU I/O queues. The PCI interface contains a memory controller that interacts with the macro instruction and DRAM memory access arbiters to allow the host to access SCC memory. The arbiters effectively dual port the memories. This mechanism allows the host access to the SCC memory while the array is active.

The second block is the Array Control Unit (ACU). Its sole purpose is to keep the PE array running at as high a speed as possible which equates to issuing PE instructions at a high frequency. The ACU attempts to achieve this goal by implementing the concepts presented in Section A.2.1.

The most important functionality of the ACU is to issue instructions to the array. The Macro Instruction Fetch Unit (MFU) fetches macro instructions from the Macro Instruction SRAM based on its Program Counter (PC). The MFU analyzes the instruction and takes one of the following actions:

1) If the macro instruction is invalid, it is discarded.
2) If it is a simple macro instruction, it is inserted into the Macro Instruction FIFO.
3) If the macro instruction requires an operand, the MFU first inserts the macro instruction into the Macro Instruction FIFO then copies the operand from the INFIFO to the Macro Instruction FIFO. This way the instruction and operand travel through the system together.
4) Finally, if the macro instruction is determined to be an ACU directive, the MFU executes it. This process is described in the following sections.

The ACU implements macro instruction expansion to reduce instruction latency and issue frequency. The expansion takes place in the Macro Instruction Expansion Unit (MEU). Typically the ACU issues decoded instruction directly to the PE array. However, since the SCC does not have an operating system, it recognizes five instructions as internal directives. These directives are inserted into the PE array instruction stream by the host to control the initialization and operation of the ACU. In the absence of an operating system, the ACU must also control tile swapping. The ACU implements Tiling Method #3, i.e., all of the instructions are executed on a tile until a feedback or communication instruction is encountered. The CONFIGURE directive is used to configure the number and size of the tiles.

The final block is the PE array itself. The actual implementation of the PE array is unimportant in the context of this research, as it is implementation specific. It is more important to note that the ACU interface is generic enough to support a variety of array configurations, as long as the array design conforms to the PE instruction specification and is able to interface to the PE Data DRAM arbiter to load/store data. In the current SCC design, the PE array will be implemented in an FPGA. This method allows various PE array designs to be explored using proven host, PCI interface and ACU hardfware.