AFRL-RI-RS-TR-2012-259

CACHE HARDWARE APPROACHES TO MULTIPLE INDEPENDENT LEVELS OF SECURITY (MILS)

UNIVERSITY OF IDAHO

OCTOBER 2012

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2012-259 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

WILMAR W. SIFRE
Work Unit Manager

/ S /

PAUL ANTONIK, Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
OCT 2012	FINAL TECHNICAL REPORT	MAR 2010 – APR 2012

4. TITLE AND SUBTITLE

CACHE HARDWARE APPROACHES TO MULTIPLE INDEPENDENT LEVELS OF SECURITY (MILS)

5a. CONTRACT NUMBER
FA8750-10-2-0135

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
33140F

6. AUTHOR(S)

Robert Rinker

5d. PROJECT NUMBER
MILS

5e. TASK NUMBER
UI

5f. WORK UNIT NUMBER
02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Idaho
Center Secure and Dependable Systems
875 Perimeter Drive, MS 1008
Moscow, Idaho 83844-1008

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-259

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The purpose of this research is to explore possible security vulnerabilities in the cache memory systems of modern multicore processors, and to develop the tools necessary for exploring such vulnerabilities, including a model that allows for the simulation of the vulnerability and for the implementation of possible solutions that defeat the effectiveness of the vulnerability.

15. SUBJECT TERMS
Multi-core processors, Multi-level security, MILS, Multiple Independent Levels of Security, cache

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**WILMAR W. SIFRE**
U	U	U	SAR	39	19b. TELEPONE NUMBER (*Include area code)* **N/A**

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

1.0 SUMMARY

This document is the final report for the project "Cache Hardware Approaches to Multiple Independent Levels of Security (MILS)," contract FA8750-10-2-0135. The purpose of this research is to explore possible security vulnerabilities in the cache memory systems of modern multicore processors, and to develop the tools necessary for exploring such vulnerabilities. These tools include a model that simulates the vulnerabilities and implementations of possible solutions that overcome the vulnerabilities.

In recent years Central Processing Unit (CPU) vendors have reached a plateau in performance, finding it difficult to continue to increase clock speeds. Instead, the trend has focused on replicating CPU resources, providing multiple cores within a single system, and even within the same processor. These cores all access the same memory, so that the operating system is able to manage the multiple cores as a single system, and therefore can schedule a task on any of the available cores. In order to allow each core to observe the same memory values at all times and throughout a multilevel cache system, part of the memory system is dedicated toward keeping each processor's view of memory consistent with each other. Messages are sent between cores to insure that values that are shared between cores are kept consistent with each other. Unfortunately, sharing values between cores could constitute a security risk, if it were possible for an adversary to monitor the cache coherence messages and reconstruct the values in memory. The objective of this project is to identify possible hardware-based security vulnerabilities involving memory, and to provide tools that can explore such vulnerabilities and implement possible solutions.

2.0 INTRODUCTION

Modern net-centric concepts are based upon ubiquitous connectivity and standards based services. This is in fact the basis upon which the Department of Defense Information Enterprise Architecture is founded [1]. However, to realize the objective of secured availability requires that users and processes with various levels of trust and access share a common infrastructure. The emerging state of the art on this type of multiple level security is based upon the Multiple Independent Levels of Security (MILS) architecture. This architecture consists of a number of highly robust components that when combined appropriately can be trusted to enforce the primary principle of Information Assurance:

Information is only shared with those processes and users allowed by policy.

In the MILS architecture [2], made popular by the Air Force Research Laboratory High Assurance Middleware for Embedded Systems (HAMES) project [3], multi-level secure systems are implemented through separation and controlled information flow. The system is built on a foundation consisting of a separation-based infrastructure, the separation kernel (i.e., hypervisor) and secure inter-processor communication (i.e., the Partitioned Communication Systems, PCS). These components isolate individual applications and services and provide the pathways for secure communication. Supporting these pathways are additional components (i.e., guards, cross-domain services, encryption engines, routers, etc.) that implement the system's security policy. One approach to the MILS architecture is to develop a *guarded communication subsystem* (GCS) which is responsible for the "routing" of messages between applications, sending them through the appropriate filters, guards and access decision points.

Consider the exemplary system depicted in Figure 1. In this figure, we have three processors, each hosting a number of processes. Some of the processes (A, B, D, E and F) are applications that have not been fully analyzed with respect to security, and are thus considered untrusted. Assume application A needs to utilize services provided by application F. Requests from A are passed through the MILS Message Router (MMR) [4] which first sends the message through the appropriate guards (e.g., G_1 or G_2) and then passes it on to the PCS which securely transmits the message to Processor 2 and its MMR. Processor 2's MMR may pass the message through Guard G_5 first, and then to the F service engine. Along the way the guards may accept the message, modify it (e.g., add additional metadata, filter contents) or reject it.

The security policy of the system depicted in Figure 1 includes all of the specifications of authorized requests and communication between the untrusted applications. For example, the policy can specify the content and format of requests from A to F (A may be running a Secret level application).

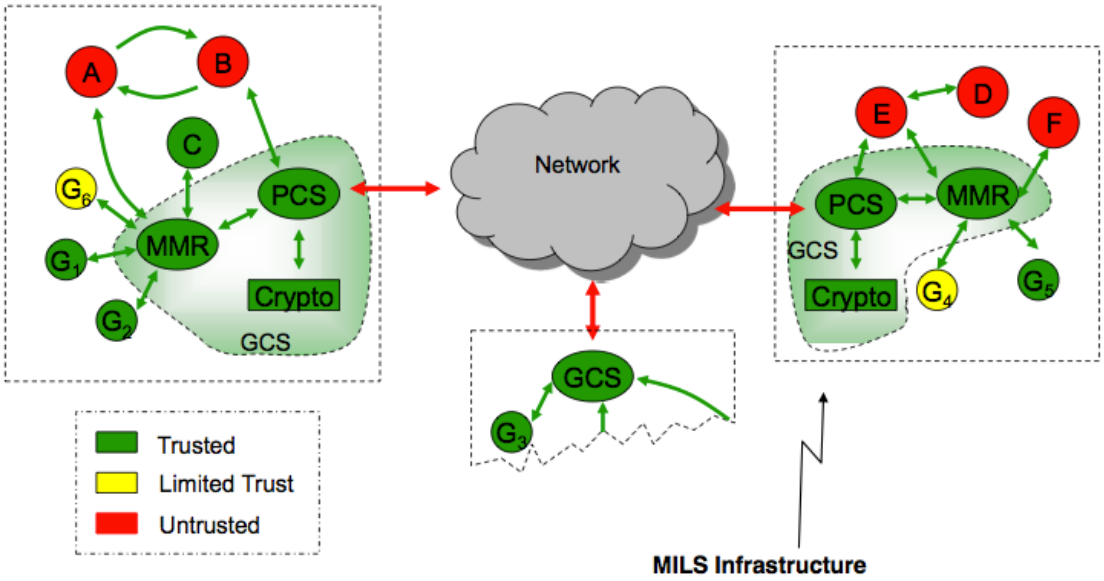

Figure 1- Secure Multi-Level Communication System, implemented using Guarded Communication Subsystems (GCS)

The security policy can be specified as a conjunction of predicates/operations performed on the messages as they travel between the processes. Each guard can be responsible for enforcement of one or more aspect of the policy. The system as a whole enforces the totality of the policy.

Moore's Law - number of transistors on integrated circuits doubles approximately every two years – has held true since it was first stated in the 1960's. The chip manufacturers have been able to achieve this result in part by increasing clock speeds. However, very recently this trend has not been able to keep pace due to limitations in silicon physics, in particular in power dissipation. As a result, the manufacturers have begun to produce multicore processors, where multiple CPUs are implemented on a single die. The usual architecture for such a system is for each core to have its own local level-one (and sometimes level-two) cache, and then have a shared main memory.

If we consider the system in Figure 1 in the context of a multi-core processor, we can deploy the processes (guards and applications) to different cores, or we could even map some of the physical processors to cores. In this MILS system, we still need to ensure separation and controlled information flow as depicted in the architectural representation of Figure 1. The open research question is: *Can we securely deploy MILS systems on multi-core architectures?* The answer to this question leads to many questions which we address in this and a companion research project. This project addresses the study of hardware issues pertaining to the MILS problem. In particular, multicore processors implement cache coherence mechanisms that use message passing between cores, over dedicated buses. This data sharing may pose a security risk if the data being shared needs to be secured.

2.1 Project Overview

Thus, the purpose of this project is to study the issues involved in extending the MILS approach to multicore processors. Specifically, the goals of this project were as follows:

- To investigate the impact of hardware-based cache models on the performance and security of MILS compliant separation kernels in multicore systems.
- Based on this investigation, explore new architectural solutions using hardware that enables improved performance and security of multicore processors while enforcing the data isolation and information flow properties of MILS separation kernels.

A computer system consists of both hardware and software. Since a companion project was charged with studying the software aspects of the problem, this project focuses on the hardware aspects of the problem. This focus was concentrated on the cache sharing characteristics of modern multicore systems that have become ubiquitous in our world. Similarly, since the x86 architecture is the dominant architecture being used, our focus initially was on this architecture. Both Intel and Advanced Micro Devices (AMD) produce processors that utilize the x86 Instruction Set Architecture; however, the underlying implementations are different. Each vendor implements a set of model (or machine) specific registers Model Specific Registers (MSRs) that are used, among other things, to configure the processors' connections with surrounding memory and Input/Output (I/O) functions. Some of these MSR's are common to both vendors, while others are unique to one particular vendor. Some are even unique within different processor models or families within a vendor's product line.

One area where processor implementations are particularly unique is in their interconnection with cache memory. Multicore implementations are especially so, since there are considerable differences in the way each core interacts with its first level cache, which is usually local to each core, subsequent level caches, which are often shared between cores, and main memory, which is almost always shared. In order for all cores to have a consistent view of memory, some sort of coherence mechanism must be employed. Usually, this mechanism involves the passing of messages between cores, to inform the non-shared memory elements of changes that other cores have made that affect the rest of the system.

As processors have become faster, and the trend toward systems having more cores has become more prevalent, the message passing mechanism has increasingly become a system bottleneck. Vendors have tried to mitigate the problem by incorporating more elaborate communication systems between cores.

Figure 2 shows the fairly simple arrangement that characterizes some of the first multiprocessor systems. These systems usually were composed of separate processor chips, connected together via physical buses implemented on the system motherboard. Memory operations and coherence between processors was largely handled via external circuits, called *chipsets*, designed specifically to work with the processor chips. The

chipset also provided the electrical interface to memory and I/O. The connection between processors and chipset was a true bus, called the front-side bus. This arrangement allowed for relatively easy implementation of coherence, since everything was connected to the front-side bus – messages could be received by all components simultaneously.

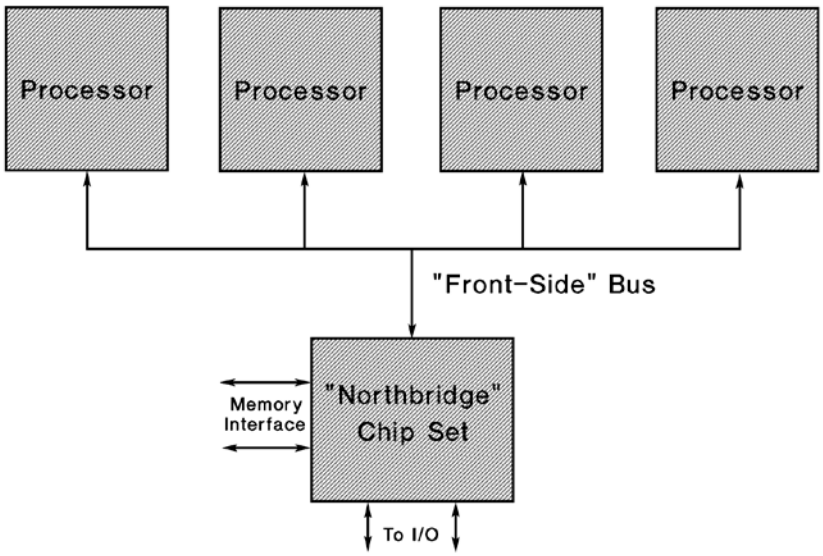

Figure 2 - Traditional multi-processor architecture, with a front-side bus.

With the advent of faster processors and more cores requiring more memory transactions and coherence messages, the front-side bus increasingly became more of a bottleneck. As a first solution to this problem, memory was moved to its own bus, sometimes called the back-side bus. The chipset circuits now had a more difficult task, since they needed to coordinate interactions between the front-side and back-side buses, including routing messages to the appropriate system modules. As a further enhancement, multiple memory and I/O buses were implemented, in an attempt to distribute all of the data traffic across multiple bandwidth elements. This made the cache coherence problem even more difficult.

Today, both AMD and Intel have come up with more sophisticated solutions. In 2003, AMD introduced the HyperTransport (HT) bus, and Intel followed a few years later with the Quick Path Interconnect (QPI). Figure 3 shows a diagram of the Intel QPI – the functionality of AMD's HT bus is similar.

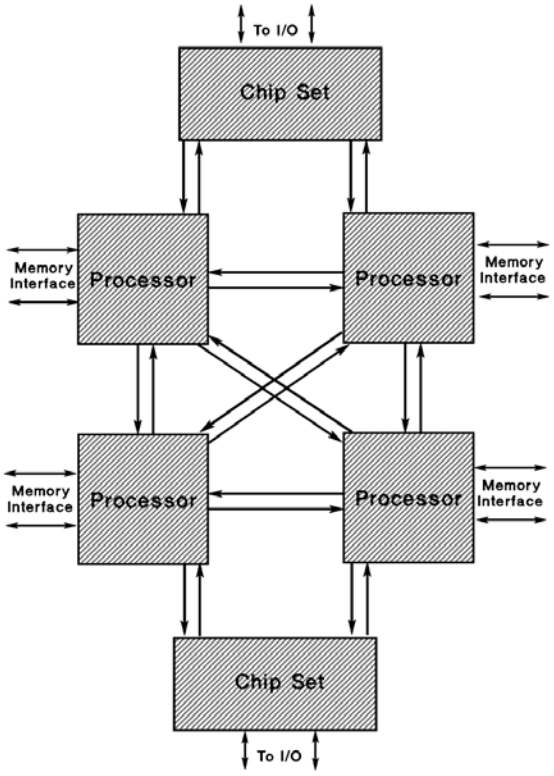

Figure 3 - High Speed point-to-point interconnect for multicore processors. This diagram is specifically for the Intel QPI, but the AMD HT bus is similar.

The former bus connections between processors have been replaced with configurable high-speed links to reduce contention that otherwise occurs in a bus-based system. The system can be partitioned such that processors that are sharing memory with each other can be connected directly to one another, and can be separated from other system elements that do not require such intimate sharing.

These new interconnect strategies are more complicated than previous solutions, and they allow and require more configuration options than the simpler bus solutions. Early phases of this project involved the study the mechanisms that are used to configure the interconnection systems, and the types of configuration that can be performed. Since the two interconnection architectures differ in detail, and are incompatible with one another, it was decided to focus on the AMD HyperTransport interconnect system first. The HT bus was conceived as an open standard, whose focus was to provide both high speed and I/O functionality to a variety of systems. It is an older and probably more established standard. As a result it was thought that the details on its operation and configuration would be easier to obtain than the Intel QPI, which is newer and was conceived from the start to be a proprietary design. We expect that much of the knowledge gained from the AMD design would also be common to Intel, and would be helpful in understanding both implementations.

Both AMD and Intel implement the x86 architecture, and even though they use different implementations, there is a motivation to maintain compatibility within the architecture. Nonetheless, some scheme must be employed that allows access to the underlying internal implementation details without significantly affecting the architecture itself. In other words, the architecture must provide a "standard means for being non-standard." The mechanism used to do this is a separate mode of operation, called System Management Mode (SMM). This mode is in addition to the more commonly known x86 modes – Real Mode and Protected Mode – and is the scheme used by both vendors to allow system dependent configuration of the underlying processor and the connection with its surrounding interconnect.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The original plan for the project consisted of six tasks:

1. Recruit and hire students, assemble team.
2. Study the cache systems of several existing Commercial Off The Shelf (COTS) processors.
3. Choose a target processor and develop a model platform to be used in demonstrating proposed solutions.
4. Implement one or more hardware-based solutions.
5. Test designs and collect data from experiments.
6. Write final report.

The project roughly followed the above schedule, and the rest of this report is organized to follow this order of tasks. Of the original six tasks, we completed Tasks 1 through 3, partially finished Task 4 and did not finish Task 5. Task 6 is this report. We were working to complete Task 4 when the project ended. We have delivered our nearly completed hardware model, the deliverable from Task 4, to AFRL personnel.

As is often the case with research projects, some of the tasks took some unexpected turns and took unexpected amounts of time. With Task 2, the focus turned to understanding the capabilities and limitations of the SMM. A possible security exploit was uncovered with SMM, and is described in the section on Task 2. Task 3 took significantly extra time, as the choice of a model platform ended up being a more difficult task, with a different outcome, than was originally expected. As a result of this delay, we were in the process of completing Task 4 when the project ended, and did not finish Task 5.

The project was originally scheduled to start in December, 2009. However, due to delays in budgeting, the contract wasn't signed until March, 2010, and a kickoff meeting was held in April, 2010. The original scheduled time for the project was two years, with an original ending date of December, 2011. Even though the start was delayed, the ending date was not moved, so the project duration was shorted by almost four months.

Funding was also adjusted. The first-year funds that were not expensed by December, 2010 were removed from the total budget.

The rest of this report is organized around those tasks that were completed, or nearly completed. Our findings and results are described in the task-based sections that follow.

4.0 RESULTS AND DISCUSSION

4.1 Task 1 - Recruit and hire students, assemble team.

The following graduate students of the University of Idaho (UI) Computer Science Department worked on the project:

1. Michael Wilder, PhD student in Computer Science, involved for the entire project.
2. Trevor Davenport, MS student in Computer Science, involved through September 2011.
3. Brandon Ortiz, PhD student in Computer Science, involved for the entire project.
4. Dallas Stinger, MS student in Computer Science, joined in June 2011, and continued for the remainder of the project.
5. Xin Mou, MS student in Computer Science, joined in September 2011 and continued for the remainder of the project.

The team was directed by Robert Rinker, Associate Professor of Computer Science, the Principal Investigator for the project.

4.2 Task 2 - Study the cache systems of several existing COTS processors.

The first thing the team needed to understand was how the cache memory system works, and how the processor is able to interact with it. For the most part, the processor is unaware of the existence of cache, except for when it tries to access memory. The memory access operation completes quickly if a cache hit occurs, but takes longer if there is a cache miss, in which case the deeper levels of cache/memory must work to service the miss. The cache system must do its job as quickly as possible, since memory accesses are the bottleneck in all processor designs – the operation of the hierarchical memory system is on the "critical path" of processor operation.

The cache system is an important performance enhancement to a computer system; however, it does not change the *functional* behavior of the system – the processor supplies an address, and the memory system either supplies the resulting value from memory, or it writes the value to memory. Except for the afore-mentioned timing differences, the processor is unaware of the presence of the cache. This is an important observation when the modeling of a processor system is taken into account; for this project, this fact will have an impact in later tasks.

The initial belief was that the processor had only limited control over the operation of the cache; this premise was borne out while studying the basic x86 processor instruction set. There are a few instructions (less than ten) that can be used to perform operations such as flushing the caches. However, a separate mode within the processor, the System Management Mode (SMM) mentioned earlier, allows manipulation of the memory mapping system.

The SMM was originally implemented to provide a mechanism to access the power management features of processors intended for laptop computers. These features were envisioned to be different from desktop processors, which did not require such sophisticated energy management techniques, and would also vary within various mobile processor models. Thus, these features were not "baked into" the architecture itself, but rather would be implemented using the MSRs mentioned earlier. All of the model dependent features could be isolated into a small group of software modules, and could be incorporated, usually into the Basic Input/Output System (BIOS) code, for the specific computer model.

Over the years, the role of SMM has been expanded to include a variety of system configuration duties, and today sometimes includes functionality that might not strictly be considered initialization. For example, software drivers that handle the Universal Serial Bus (USB) mouse and keyboard that are common on today's systems is typically set up to execute in SMM by the BIOS. These drivers cause these USB devices to mimic the older-style serial hardware, effectively hiding the actual mouse/keyboard type from the operating system. The use of SMM for this purpose is convenient and "innocent" in this case. However, it points out a potential risk that SMM poses – functions can be set up by

a malicious BIOS that can be hidden from even a supposedly security-aware operating system. This fact presents a potential security vulnerability.

As mentioned earlier, during this phase of the project we limited our focus to the AMD x86 architecture. We acquired two AMD Phenom II X4 810 systems. These systems contain four cores, run at 2.6 GHz, and have 6 GB of memory. While these systems were "commodity" systems, they represented the state of the art at the time of purchase. The BIOS resides on flash memory chips, and can be modified in situ. We obtained some blank flash chips so that we could physically replace the BIOS but preserve the original BIOS code.

Some observations from this study appear below:

The initial effort focused on understanding the AMD memory system, with a special focus on the memory controller. Much of the information that is available comes from the "BIOS and Kernel Developer's Guide" (BKDG), available from the AMD website www.amd.com. This guide consists of over 400 pages, and is quite thorough, although some modifications and additions have occurred for various versions of the processor. Several facts have emerged from the study of the BKDG:

1. Before the memory controller is initialized, the level 2 cache memory can be used as regular RAM memory. This provides a processor access to a memory area that is consistent among processors, and independent of the final main memory configuration. It can serve as a sort of "scratch pad" or alternate memory while main memory is being configured.

2. The memory controller can be manipulated via registers that are accessible to a privileged (level 0) user. SMM, or some other special processor mode, is not necessary to modify the behavior of the memory controller. Given a task that is running at this high privilege, it appears to be possible to modify the memory controller configuration at any point in time.

Of particular note is the usual role of the SMM, which operates similarly in both AMD and Intel implementations. Figure 4 shows the possible transitions between modes in the x86 architecture. A separate area of memory, called System Management Random Access Memory (SMRAM), can be configured for use by SMM. This memory can either be shared with the rest of the system, and in the rest of the processor modes, or it can be excluded from the rest of the main memory, visible only while executing in SMM. SMM can be entered from the other modes either programmatically, or via the assertion of an external signal called System Management Interrupt (SMI). Code in SMM memory can be executed, and then the processor can return to its former mode via the RSM (resume) instruction.

The potential vulnerability could be exploited as follows: A (malicious) BIOS could set up an area of exclusive SMM memory, and install an SMI handler that performs some unauthorized task. As mentioned earlier, SMM has access to the entire processor, so

virtually any information contained within the system could be accessed. Since this handler is in memory only visible to SMM, and the SMI interrupt can be generated external to the processor, the operating system would be unaware of its operation, or even its existence. While the SMI handler is running a few CPU cycles will be consumed, which could possibly be detected by the operating system, but this is not something that it usually monitors.

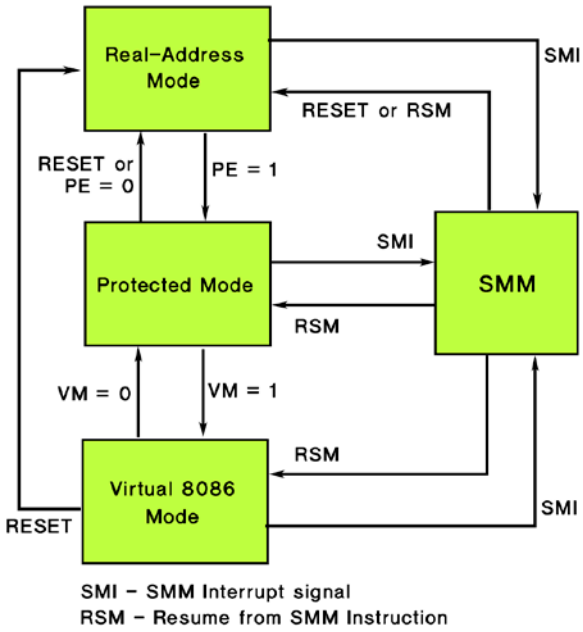

Figure 4 - State Diagram showing transitions to and from SMM from the other x86 operating modes

In our experiments with SMM we were able to modify the startup code of the Linux kernel so that we can execute an SMI after the kernel has changed to normal (protected) mode. Since SMM is a truly separate operating mode for the processor that is unknown to the "normal" mode and to the kernel, changes to system state can be made without the kernel ever knowing – a possible exploit path. This fact should be of significant concern, since it is possible for the BIOS to configure the memory system and enable an entry into SMM via an SMI from either external stimulus or from a normal user-mode process executing on the processor, without the operating system knowing about it.

It is possible to "lock down" SMM after it has been configured. It appears that the proper use of this feature is for the BIOS to configure the required SMM options, and then lock down the SMI handler by setting a bit in one of the MSRs. Once this bit is set, no further modifications can be made to the handler without doing a hard reset.

Surprisingly, the BIOS's that are delivered with both the Intel and AMD machines we tested do not perform this final step, so it is possible to modify the SMI handler even after

it has been configured by the BIOS. This modification could be performed by malicious code embedded and delivered in the BIOS, or by code that is executed during the start-up of the Operating System. It is strongly recommended that the BIOS of any commodity system where confidential information might be processed be replaced with a known, trusted one which sets the lockdown bit.

We replaced the BIOS of our AMD machines with our own, derived from the *coreboot* project (www.coreboot.org). In addition, we have developed a software tool that allows us to more easily create modules for the SMI handler. A description of this tool and the process is included in Appendix A.

By using our own code added to the BIOS, we were able to demonstrate that the HT bus can be configured and reconfigured. We also demonstrated that the HT bus can also be reconfigured using SMM, even after the operating system boots. While this capability does not appear to be used in any current system, the capability nonetheless exists. This could cause the entire system to be reconfigured, and using SMM it could be done without the operating system knowing about it.

We did not explore all the possibilities in terms of configuring the HT bus. We were able to modify the configuration enough to "crash" the system. The obvious next step would be to change the configuration in such a way to allow the system to continue to operate. However, because of our choice of models, described next in Task 3, we did not pursue this step.

4.3 Task 3 - Choose a target processor and develop a model platform to be used in demonstrating proposed solutions.

This task involved the selection of an experimental platform with which we could study possible exploit scenarios via the cache coherence protocols being used, and the new processor interconnect mechanisms that have been recently introduced.

Most commodity systems referred to as "multicore" systems are more strictly called Symmetric Multi-Processor (SMP) systems. The term *symmetric* refers to the fact that all the CPUs in the system are identical. One CPU is designated as the "master" processor during bootup, but usually once the system is started, all the processors are equal within the system. Another view is that the multiple CPUs in the system are simply system resources that can be allocated and scheduled, just like the other resources within the system, such as I/O devices and files. By contrast, an Asymmetric Multi-Processor (AMP) system contains different, usually specialized, CPUs. Examples include the Sony Playstation, which has a Cell processor made up of asymmetric cores. The main core is a PowerPC-based core that governs several slave Reduced Instruction Set Computer (RISC) based execution cores.

SMP systems are not new. However, a difference between traditional SMP system and the new multicore systems is that the term "multicore" usually refers to a processor chip that contains multiple CPUs (the "cores") on the same die, or at least in the same physical package. Older SMP systems utilized separate packages, connected together via actual wires on the motherboard. Today, the term *processor* is used to specify the actual package, whereas the term *core* is used to specify the CPU, along with its associated circuitry – usually including level 1 and sometimes even level 2 cache. Thus, traditional SMP systems had one core per processor, and multiple processors in a system, while modern multicore systems, built around the Intel CoreDuo or AMD Phenon processors, have a single processor containing multiple cores.

SMP systems, and specifically the COTS x86 processors (standard PCs) - are organized using a shared memory model. This model relies on a cache coherence protocol to keep the local cache memories attached to each core consistent. This protocol must be implemented in hardware, since it operates within the most time-critical part of the entire computer system – the processor to memory path. The hardware consists of a specialized high speed interconnect that is used to send messages between memory modules (local caches and main memory). This interconnect is the Intel QuickPath Interconnect or the AMD HyperTransport Bus discussed earlier.

Also mentioned earlier was our decision to focus on the HT bus. The HT Bus is promoted as an open standard, and is supported by a consortium of interested manufacturers, www.hypertransport.org. However, the cache coherence protocol AMD uses over this bus is proprietary – this fact would prove to be problematic.

During project conception, we did not know what types of experimental platforms would be useful, so we left open the possibility that the platform could be hardware-based, which would likely consist of an actual PC system that might be used to exploit any security vulnerabilities that might exist, or a software simulator of such a system. The hardware system would allow us to experiment with the actual hardware that could be used with such an exploit, but might not provide the flexibility to try new ideas. The software system, probably using one of the available processor simulators currently available, would probably be slower but would provide more flexibility to test different configurations.

After some discussion, we decided to first explore a hardware system. Our original plan for an experimental platform was to use our AMD-based Phenom systems as a platform, and then develop or acquire a Field Programmable Gate Array (FPGA) based hardware attachment to the HT bus, so we could monitor the cache coherence traffic on the bus. Analyzing the traffic, we could determine what a potential adversary could glean concerning memory contents. Once we understood the possible vulnerabilities, we could use the same system to try methods to defeat the exploit.

Initially, we pursued a scheme where we would use our AMD Phenom systems to monitor the cache coherence traffic on the HT bus using some sort of hardware attachment to the HT bus. We joined the HyperTransport Consortium, a primarily industrial group whose members have some interest in the HT bus. This membership allowed us to access numerous HT Bus documents and other resources. Boards that did similar functions to what we were hoping to do have been developed by other consortium members – in particular, an FPGA based board developed by the University of Heidelberg in Germany looked promising. AMD has a research and development facility in Heidelberg, and they have collaborated with the University in developing the board.

Upon further investigation, we encountered several obstacles towards the implementation of the hardware-based system. First, the motherboard in our AMD PC does not include a connector to the HT bus. In fact, there are currently few if any motherboards being manufactured with an HT bus connector. Motherboards that were designed around 2005 had such connectors, at a time when today's crop of multicore chips were not available. The only way to create a multiple CPU system was to use several single core processors and connect them together via an external HT bus. Additionally, the HT bus was intended to connect very high-speed I/O devices, such as Infiniband, to the system. With the advent of multicore processors, the cores all reside within a single processor component. Therefore, an external extension of the HT bus is no longer necessary. So, while the HT bus is still being used and new versions are being developed, it is now internal to the single chip processor itself, and not being made available outside of the processor package on today's motherboards. Several motherboard manufacturers are planning to introduce external HT connectors, to accommodate systems that require that several multicore processors be connected together in a single system. However, no such boards were available on the market at the time of the study. It is still possible to purchase some of the older motherboards from secondary channels such as eBay, but they represent

older technology. This problem could have been solved if we had not encountered the other issues.

The second issue is the cost of the hardware necessary to monitor the HT bus activity. We have been in contact with the University of Heidelburg in Germany, who has been designing and testing FPGA-based HT boards. These are very sophisticated designs, utilizing Low Voltage Differential Signaling (LVDS) techniques to transmit HT bus signals to the FPGA, and as a result they are expensive – we were quoted a price of 5500 euro, somewhere around $7500, for one of their boards. Even at that price, the limited speeds of FPGA technology do not allow full speed operation of the HT bus. This cost is more than we had anticipated, although some adjustment in the budget might have allowed us to pursue the purchase.

The final issue and the ultimate show stopper for this bus monitoring approach, is that AMD will no longer provide licenses for their cache coherence protocol. Obtaining such a license was a prerequisite for purchasing the FPGA board from the University of Heidelberg, since their FPGA core requires that we have such a license. AMD no longer supports the development of any sort of coherent bus hardware, such as the cache monitoring system we were hoping to build.

The final hardware monitoring approach we investigated is to use a logic analyzer to monitor HT bus activity. Some general purpose logic analyzers have attachments and software available which allows one to monitor bus activity of the HT bus; one such model is the Agilent 16900. These analyzers are able to monitor the HT Bus at full speed. However, these logic analyzers are even more expensive than the previous solution, they require that the motherboard of the monitored system have an HT connector, and as before they cannot decipher the proprietary AMD cache coherence protocol. It was decided that the hardware bus monitoring approach was not feasible. We then turned our attention to software and hardware simulator approaches.

Several software simulators for the x86 are available, both proprietary and open source. One problem with all of the simulators we considered is that they do not actually simulate the operation of the cache memories, or if they do, they do not do it very faithfully. As mentioned earlier, cache memories are included to improve hardware performance, and do not add any additional functionality as far as the CPU operation is concerned. For a software simulator, doing a simulation of cache behavior would not add any functional detail to a processor simulator, and it would slow down an already very slow simulation process – the simulators often run hundreds or thousands of times slower than the processor they are simulating. Thus, none of the simulators we considered were particularly well-suited for our needs. Since we would need to develop our own software model for cache coherence, it was important that the interfaces between the CPU module and memory, where we would need to add software to simulate the cache coherence protocol we wanted to study, be well-defined. For the open source ones, at least theoretically it would be possible to find this interface; for the proprietary ones, the information needed might not be available or accessible.

AMD has a proprietary simulator called SimNow. After some review, it was decided that this one would not serve our needs. It consists of modules that, based on the particular model of processor being explored, can be linked together into an executable. Information regarding the internal workings of each module was not available, and in fact there was no guarantee that the resulting simulator was actually doing a hardware-level simulation, as we would require. The modules themselves were simply "black boxes" that were selected for use by a configuration tool. A description of the modules being selected was minimal. This simulator was ruled out for our purposes.

A couple of open source simulators were also evaluated. Quick EMUlator (QEMU) (www.qemu.org) is a popular simulator whose modules are supplied by a community of developers. Since it an open source project, information is available concerning each module. However, the availability of modules seems to significantly lag the time when the corresponding hardware hits the market. Basically, the software module can only be implemented after the hardware can be analyzed and simulated. It does not appear that a module that implements cache coherence was available, at least at the time, so it would need to be created from scratch.

Another open source simulator is called Bochs (www.bochs.org). Its target appears to be more at the instruction-level, rather than at the hardware level, and its development appears to be less strongly supported, and if anything is behind QEMU in its development.

Another problem with all of the software simulators is that the expertise of our research team is not so much in the software simulator realm, but rather is slanted toward the hardware. This was on purpose, since the project was originally envisioned to be a hardware-oriented one. The learning curve for any software simulator is fairly steep, since they all must simulate a complicated system. It would have taken our team a significant amount of time to learn how the simulators work, and even longer to begin to adapt one of them for our purposes. Nonetheless, we were on the verge of choosing QEMU as our experimental platform.

Just as we were making our decision, the research team at AFRL was choosing a hardware simulator based on the Open-SPARC T1 project. As a result of their decision, we also decided to go with the Open-SPARC T1. This would allow us to be more compatible with the efforts of the AFRL team, who are designing a multicore processor from the ground up with MILS in mind. The FPGA-based solution also fit the expertise of the University of Idaho team better – it was hoped that this would help us make up for the time lost in determining the experimental platform. The SPARC T1 system that is being emulated is a different architecture than x86, which is a disadvantage of using this approach. However, the SPARC T1 also uses a cache coherence protocol, called MOESI (Modified-Owner-Exclusive-Shared-Invalid, the names of the legal states within the protocol), similar to the AMD processor. While the actual instruction sets are different, it has been shown that the actual mix of instructions that are executed to solve a given task are largely independent of the instruction set itself, even for widely different instruction set architectures [5].

Parts of this processor have been implemented on Xilinx FPGAs, which allows us to modify the design to incorporate the cache coherence features we need for the project goals. It also would allow us to have a common platform with AFRL researchers, providing the potential for future collaboration.

Finally, the OpenSPARC system appears to be the only full-scale open source multiple core architecture available for FPGAs. It is derived from the actual Verilog that was used to implement the original commercial chip. Therefore, it is more than just a simulator - it implements almost all of the details of the actual processor.

This choice also had several drawbacks, which became more apparent as we worked with the model. While not being a perfect solution, we ultimately decided to pursue this option. A description of the original SPARC T1 processor, along with its FPGA implementation, is provided in the next section, Task 4.

4.4 Task 4 - Implement one or more hardware-based solutions

The Sun SPARC T1 processor, upon which the FPGA-based simulator is based, was introduced in 2005, and is specifically designed to be a server. A block diagram of the processor is shown in Figure 5. Its architecture is optimized toward handling a large number of short jobs, as is typical of server loads. It utilizes parallelism at several levels to achieve high throughput for short jobs – it includes up to eight processors, each of which can execute up to four threads at once via hyperthreading. Within each processor, fine-grained thread scheduling is utilized – threads are switched at each clock cycle, with any idle threads being skipped. Thus, a processor is never idle unless it truly has no thread available to execute.

The memory configuration exhibits similar parallelism. Each core has its own local Level 1 cache. Four level 2 caches are included, connected to the cores via a cache crossbar, so that multiple L2 cache accesses can occur in parallel. Latency due to an L1 cache miss is hidden due to the fine-grained scheduling of tasks.

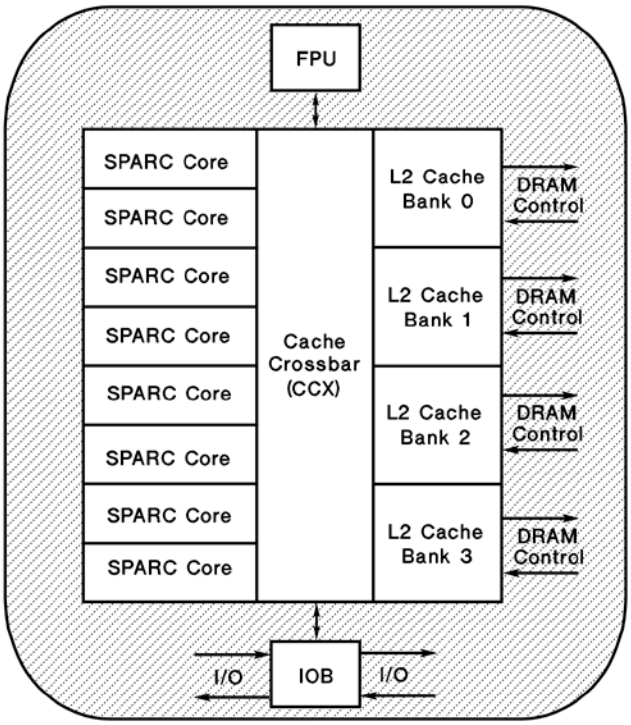

Figure 5 - Block Diagram of the SPARC T1 processor

Shortly after the processor was introduced, Sun Microsystems made the Hardware Design Language (HDL) description of the design – the Verilog code used to implement the processor design - available to the public, in a project it calls OpenSPARC T1. This is one of the few commercial processor designs, and certainly the most complicated, available in “source code” form.

A portion of this HDL description has been adapted for implementation on an FPGA. In a project jointly supported by Sun and Xilinx, a single processor core has been ported to a Xilinx Virtex 5 FPGA, attached to a circuit board that includes memory and peripherals to create a complete system, the Digilent OpenSPARC Evaluation system, shown in Figure 6. A block diagram of this board, showing the FPGA and array of peripherals, is shown in Figure 7. Since the actual HDL code from the original SPARC T1 processor is used in the FPGA implementation, the system is more than just a simulator – it is executing the actual HDL description that was used in the real processor.

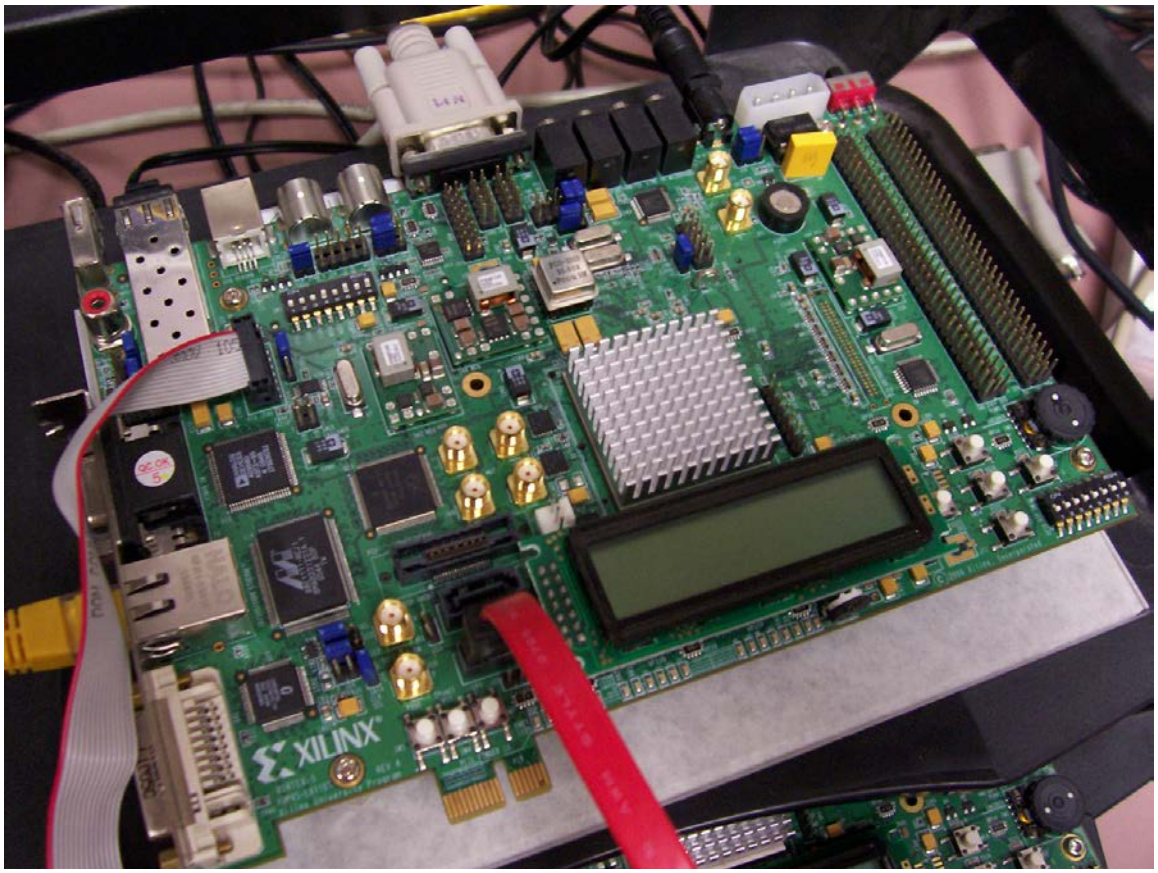

Figure 6 - The Digilent OpenSparc Evaluation Board used to implement the OpenSPARC processor core. The red cable is the SATA cable that connects to a second identical board, part of which can be seen underneath the top board.

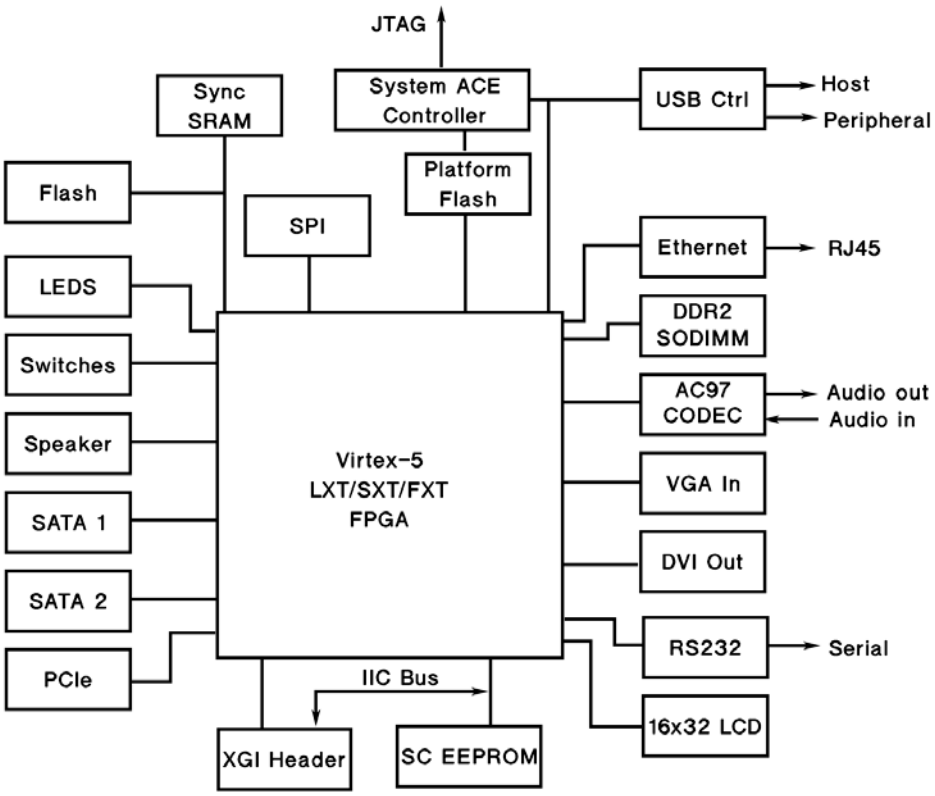

Figure 7 - Block Diagram of the Digilent OpenSPARC Evaluation board. The board includes a Xilinx Virtex 5 FPGA, RAM memory and an extensive array of peripherals and I/O to support an entire computer system.

This project has since been extended to dual (and more) cores, by connecting two or more individual FPGAs, each implementing a single core, with a simplified version of the cache crossbar that exists in the original T1 design called the ccx2mb ("cache crossbar to microBlaze"). Communication between the two cores is implemented using a Xilinx MicroBlaze soft processor. A goal of this project was to demonstrate the utility of the MicroBlaze processor in implementing this high speed data link. The link itself utilizes the SATA interface that is available on the FPGA board. It is this specific implementation that serves as the basis for our hardware simulator. A block diagram of the resulting two core system is shown in Figure 8.

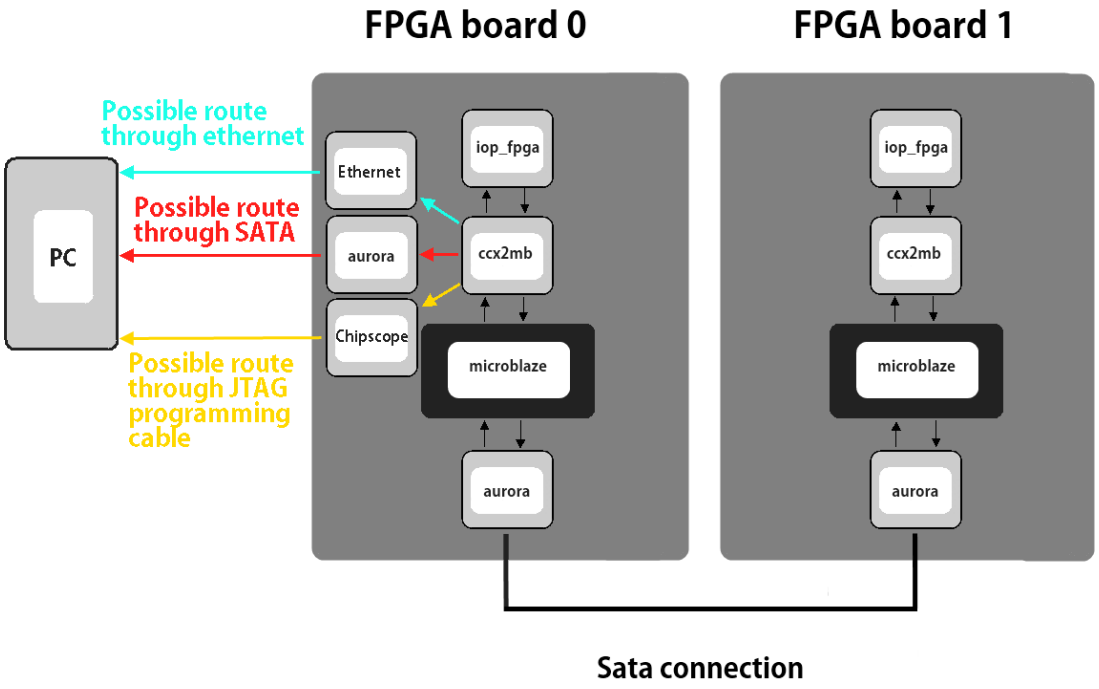

Figure 8 - Block diagram of a two core, FPGA-based OpenSPARC system, including the attached PC that serves as the console terminal, the host for the FPGA programming tools, and data collection device.

With the decision made concerning the hardware, we began to assemble the necessary software tool support to proceed. Programming the FPGA requires the Xilinx Integrated Software Environment (ISE) Software tool set, including a license for the base software and the optional Embedded Development Kit (EDK). The Xilinx software tools are very complex, and are non-trivial to install and use. Upgrades are not necessarily upward or backward compatible. The original OpenSPARC Project was developed using version 10.1 of the tool set, but the latest version is 13.1. So the first order of business was to adapt the OpenSPARC system to the newer version.

Implementation of the experimental platform included several challenges. The SPARC T1 project was originally undertaken by Sun Microsystems to create a new processor using full custom Very Large Scale Integration (VLSI) – it was not intended to be implemented on an FPGA. As such, it is a very large and complex system – the learning curve has been very steep. The complete OpenSPARC FPGA project actually includes several different reference design variations; the reference designs appear to have been chosen to demonstrate different capabilities of the Xilinx FPGAs. So, in addition to having to understand the overall system, we also had to determine which design was best to start with. During our investigations, we discovered several issues:

- Only a small portion of the entire OpenSPARC T1 processor system has actually been ported to the FPGA – even though the Open SPARC architecture consists of

eight cores, only a single core will fit on a single FPGA. One of the projects in the OpenSPARC project has implemented a dual-core system, but it requires two Xilinx FPGAs on two separate (but identical) circuit boards. It appears that at least part of the motivation for this particular Open SPARC project was to demonstrate the use of the Xilinx Intellectual Property (IP) – the micro Blaze softcore and ancillary logic modules. An advantage to this particular reference design for us is that it was relatively easy to isolate that part of the model where the communication traffic between cores was being processed. The actual cache coherence traffic travels over a standard SATA bus, which makes it relatively easy to identify and capture for analysis. Ultimately, it was this reference design that we used.

- The cache crossbar that is present in the original SPARC chip was replaced in the FPGA version by a much simpler version. This version utilizes the Xilinx MicroBlaze soft core, and is therefore called the CC2MB ("Cache Controller to MicroBlaze"). The SPARC processors communicate with this simpler version in a similar way to the Application-Specific Integrated Circuit (ASIC) version, so much of the original documentation is still valid. Nonetheless, the design is different from that of a "real" SPARC T1, so any documentation on the T1 is suspect.
- The FPGA version runs around 20 times slower than the ASIC version. While this is not particularly important to our work, since we are not concerned with performance issues, it nonetheless means that the FPGA-based processor is quite slow. For example, it takes more than an hour to boot the operating system.
- The FPGA design does not actually implement any Level 2 cache. As stated earlier, caches improve performance, but do not change the functionality of the system. Implementing memory on an FPGA requires extensive programmable resources within the FPGA. Even with the large capacity FPGAs present on the boards, just implementing the core stretches the resources that are available on the FPGA. Since performance was not a primary focus, L2 cache is not implemented. While this does not impact normal processor function, it is not clear how it affects some of the other functionality we are interested in that is related to the L2 cache operation – in particular, the actual cache coherence protocol that is being used.
- The processor runs a modified version of the Solaris Operating System that appears to be missing many of the standard system utility programs. There is no documentation concerning this minimal version of Solaris – we have had to discover which pieces are included. While it appears that the kernel for the most part is complete, we still do not know precisely which features are included. While Solaris normally comes with a rich set of multithreading and multicore tools, most of these utilities do not seem to be present in this minimal version. Also, tools such as editors and compilers, necessary to produce new executables for the system, are not included.
- With Oracle taking over the former Sun Microsystems, many of the links to the online documentation about OpenSPARC are broken. This makes it difficult at times to obtain the necessary documentation of the system.
- In particular, documentation on the cache coherence protocol is very minimal at best. The original SPARC T1 documentation specifies that a MOESI protocol is

used in the actual processor – our initial impressions are that a simpler protocol is implemented on the FPGA ported version, as part of the MicroBlaze firmware. It has been very difficult to find out the specifics of this protocol, and in particular the messages that are produced as the protocol transitions from one state to another. None of the reference designs in the OpenSPARC project appear to deal with this aspect of the processor, so minimal information appears to be available on this aspect of the system. Unfortunately, this is a primary area of interest for this project.

These issues presented several challenges in implementing our experimental platform. First, in order to better understand the capabilities of the Solaris system, we used an old SPARC V1 system, owned by the UI Computer Science Department. While this system is not multicore, the operating system is complete, and the usual program development environment and the operating system multithreading resources are available. Using this system we are able to create SPARC executables, debug them, and then move them to the FPGA system, thereby allowing us to write test codes to exercise the cache coherence system. This has proven to be invaluable in allowing us to observe a cause and effect relationship between shared memory manipulations and the cache coherence messages produced as a result.

The primary purpose of the port to an FPGA was to demonstrate the utility of a Xilinx soft processor, the MicroBlaze, in implementing a high speed interconnect. The project did not focus on that part of the system; of most interest in our investigation was the cache coherence protocol. This protocol is only explained in general terms within the OpenSPARC-Xilinx system documentation. Since the Verilog code used to formulate and transmit the messages is available, it should be possible to glean the details of the messages by examining the code.

One advantage of the SPARC cores being implemented on separate boards, and connected by a single SATA cable, is that it has been possible to identify the design sections that actually transmit cache coherence messages between the cores. Each cache coherence message is 124 bits long, and consists of several fixed fields within those bits. Since the original SPARC T1 cores also produced 124 bit messages and sent them to the cache crossbar and lower levels of the memory hierarchy, we believe that the messages being used are the same as in the original T1 design.

Since the SATA bus that connects the cores in the FPGA system is only 32 bits wide, the long messages are split into four parts, and then transmitted in sequence. The primary purpose of the MicroBlaze processor is to receive the long messages, break them into 32 bit chunks, and then handle the framing and synchronization involved with the transmission of the four chunks to the other core. Another Xilinx module, called Aurora, is responsible for the low-level management of the SATA interface.

By studying the Verilog code, we believe that it is possible to decipher these messages, and determine the details of the protocol being used. So far, we have been able to intercept the messages. Then by modifying the Verilog code within the T1 that handles

the messages, we have been able to utilize one of the Xilinx ISE development tools called ChipScope to capture the messages produced during a specific time period, and write them to a file on the console PC. Then we can analyze this message trace off line, to determine the types of messages included in the trace, along with the reason for each message or set of messages. The amount of raw information these traces contain is on the order of tens of megabytes, and it appears that the majority of this data does not pertain to messages involved in the cache coherence protocol. If these other messages are filtered out, the size of the trace can be significantly reduced. We were not able to complete work on parsing these messages, or to determine the meaning of the messages as they relate to the cache coherence protocol.

Virtually the only documentation concerning the messages that are generated for cache coherence is in the Verilog code that implements the protocol. Thus we have had to infer the protocol by observing the messages.

In parallel with this message deciphering task, we have also developed a set of thread programs that we believe will cause cache sharing messages to be produced, so we can observe the cause and effect relationship between memory sharing operations and the messages they produce. We were not able to complete this part of the work in the reduced timeline for the project.

5.0 CONCLUSION

The project produced several useful results, in spite of not accomplishing all of the tasks that were originally proposed. The highlights from these results follow:

- The processor architectures of both Intel and AMD x86 processors were studied, with an emphasis on the AMD implementation. The capabilities of the intercore buses (QPI for Intel, HT Bus for AMD) are not currently being fully utilized in either processor.
- SMM exists in both processors, and allows access to all of the system-specific registers of the two x86 implementations. Usually, SMM is configured by the BIOS. A mechanism exists for "locking down" SMM after it has been configured by the BIOS, but this mechanism doesn't appear to be utilized by the standard BIOSs in either type of system. It could be possible to use SMM to compromise data within a system as a result. Therefore, it is highly recommended that the standard BIOS of any system involved in an application involving secure data be replaced with a trusted BIOS that locks the functionality of SMM.
- The details of the AMD cache coherence protocol that is implemented on top of the open HT Bus is not publicly available. The good news is that it is difficult to obtain the information necessary to monitor cache coherence activity on the HT Bus, thereby making it difficult to "snoop" memory contents via the cache coherence messages. The bad news is that we were not able to proceed with our initial idea of using a hardware attachment to the HT bus to both monitor cache coherence transactions, or to study ways in which these messages could be used to compromise secure data, or to develop ways to prevent such attacks from occurring.
- Similarly, virtually all motherboards being produced today lack the HT bus connections necessary to attach external monitoring hardware on the bus. Currently, this makes the snooping of the bus for malicious purposes more difficult to do. However, the HT bus signals are still present on the processor pins, so they are still accessible externally to the processor package. Thus, it is still possible to design motherboards with such hardware built in. Also, as systems grow to include more cores than can be fabricated in a single package, the HT bus will once again be brought out on the motherboard; the current situation is almost certainly a temporary one.
- We investigated the use of software simulators, and concluded that since they generally do not implement a cache mechanism, they would be inadequate for use in this study, at least not without a significant amount of work.
- We were able to use the FPGA-based OpenSPARC T1 platform to produce an experimental platform for further work in this area. We modified the platform so that we can intercept cache coherence messages between cores. We did not finish the work involved in deciphering and interpreting the messages, due to the termination of the project. However, it should be a relatively easy task to interpret the cache messages, and complete the necessary platform functionality. The

platform, as is, is a reasonable starting point for future work involving the understanding of cache coherence, and its possible role in cyber attacks.

We have delivered our version of the OpenSPARC T1 FPGA model, including the modifications necessary to capture and store the messages being passed between cores on a host PC, to AFRL personnel.

Two students who worked on the project are completing their graduate degrees, using the results from this project as a basis for their theses. One of these students is reporting on the software tool he developed to manipulate SMM. Another student is continuing the work to decipher the cache coherence messages produced by the intercore communication of the SPARC T1, and relating those messages to the implementation of the MOESI cache coherence protocol.

Even though we were not able to complete the experimental platform, nonetheless the work we have done should prove useful for the work that is continuing at AFRL. We have delivered the modifications we made to the two-core OpenSPARC reference design that allows it to be implemented using the latest version of the Xilinx development software, and allows for capture of cache coherence messages.

6.0 REFERENCES

1. Department of Defense Information Enterprise Architecture Version 1.1, May 2009.

2. J. Alves-Foss, W. S. Harrison, P. Oman and C. Taylor. "The MILS Architecture for High Assurance Embedded Systems", *International Journal of Embedded Systems*, 2(3/4):239-247, 2006. DOI: 10.1504/IJES.2006.014859.

3. Air Force Research Laboratory, High Assurance Middleware for Embedded Systems.

4. B. Rossebo, P. Oman, J. Alves-Foss, R. Blue and P. Jaszkowiak, "Using Spark-Ada to Model and Verify a MILS Message Router", In *Proc. International Symposium on Secure Software Engineering*, March 2006.

5. John L. Hennessey, David A. Patterson, "Computer Architecture: A Quantitative Approach", Fifth Edition, Morgan Kaufmann, 2012.

6. Intel Corporation, "An Introduction to the Intel QuickPath Interconnect," January 2009, Document number 320412-001US.

7. Intel Corporation, "Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1".

8. L. DuFlot, D. Etiemble, and O. Grumelard, "Using CPU System Management Mode to Circumvent Operating System Security Functions," Direction Centrale de la Securite des Systemes d'Information, 2006.

9. Sun MicroSystems, "OpenSparc T1 DataSheet," Part No. 819-5015-10, 2006.

APPENDIX – System Management Mode Description

System Management Mode on AMD64 Family 10h Processors

System Management Mode (SMM) is a separate operating mode found on X86 processors that allows system level execution outside of the operating system's control. SMM is entered through special System Management Interrupts (SMI) to perform tasks considered out of the operating system's control such as performing low level power management tasks or emulating a PS2 keyboard device via alternative connections such as USB. In response to an SMI, the system executes an SMM handler to perform the required action.

This work looked at modifying the SMM handler on AMD Phenom II processors to gain an operating environment that could be used to monitor the operation of the system.

X86 processors include multiple modes that provide different capabilities for the processor. These modes include Real mode, Protected mode, Unreal mode, Virtual 8086 mode, Long mode, and System Management. Each mode provides its own abilities and limitations that determine when a specific mode should be used. For example, Real mode makes use of a 20-bit segmented memory model which allows access to 1 Megabyte of memory which can freely accessed by any running software. Protected mode offers paging based access to memory which includes limiting where software is allowed to access as well as providing a significantly larger 32 bit (as compared to Real mode's 20 bit) address spaces allowing 4 Gigabytes of addressable memory. Long mode allows access to 64 bit extensions that allow larger registers and a 64 bit address space with the same memory access and protection mechanisms provided by protected mode. Traditionally, an x86 processor begins executing in Real mode upon power-on until software is executed to change to another execution mode. Most mainstream x86 based operating systems switch to protected mode, or long mode if supported by the processor, early during the boot process. Only early execution software such as boot loaders or other pre-operating system software typically uses Real mode today.

A special version of real mode often referred to as "Unreal mode" can be entered by loading a Local Descriptor Table (LDT) and Global Descriptor Table (GDT) while in real mode and setting the segment limits to the full 32 bit range instead of 64 KB segments initially used to create a flat memory model. No additional memory protection is provided as is normally found within the typical 32-bit protected mode operation but the additional memory available over real mode provides a much more capable environment.

System Management Mode (SMM) is a separate operating mode first introduced in the INTEL 386SL present in the modern X86 line of instruction architectures that provides an execution environment which is typically outside the control of the operating system.

SMM provides an execution environment that is outside of the operating system's control for the purpose of performing system level tasks such as handling hardware errors,

emulating hardware or power management tasks. SMM can only be entered by signaling a System Management Interrupt (SMI). Traditionally, SMIs are signaled through a specific pin on the processor, though more recent APIC (Advanced Programmable Interrupt Controller) based processors allow more methods such as most interrupt sources that come through the local APIC on a CPU. Upon receiving an SMI, the processor core saves the current execution state to a separate address space call SMRAM (System Management Random Access Memory). The SMRAM region is a specific region of RAM (Random Access Memory) specified for the storage of the processor state as well as the SMM handler and anything else it requires. After saving the current state to the SMRAM region, the program counter is set to begin executing at the beginning of the SMM handler.

The system firmware (typically BIOS or EFI implementation) specifies where the SMRAM region is located and installs the SMM handler to this region prior to beginning to execute any boot software. The SMRAM region is intended to be in a protected region of memory so that it is protected from modification or replacement. The SMM handler may or may not be vital for the operation of the system. For example, without a handler, the processor will still function in the other operating modes but functionality that may have been provided by the SMM handler may not be present, such as emulation of a PS2 keyboard from a USB keyboard that would require a USB HID driver instead of a simple PS2 driver. If the SMM handler has been used to correct errors the system may not function properly. Traditionally, the SMM handler has been located in the A segment (ASEG) region which specifies a region of memory from the physical address 0xA0000 to 0xBFFFF. This region is commonly used to provide the MMIO (Memory Mapped Input Output) region to the primary video device. This allows the SMM handler to be hidden from view since access to this region of memory will instead operate on the video device instead. Access to this region is allowed by the memory controller which the processor programs to allow access when entering SMM. SMM is not required to be in this ASEG region.

The SMRAM region has a specific structured layout. SMRAM begins at the value indicated by the SMBASE (or SMMBASE) register. The default value on system boot for the base value is 0x30000. Relative to the SMBASE register within the SMRAM region are the SMM handler and SMM state save area. The SMM handler begins executing at SMBASE+0x8000 upon receiving an SMI. The save state area is located at SMBASE+0xfe00 though SMBASE+0xffff. The SMRAM region can optionally exist in a different memory location other than main RAM. The SMRAM saved state includes the current processor state at the time the SMI was received.

SMM provides an execution environment much like Real mode. The initial execution state is a 16-bit operand sized, 20 bit segmented memory model. Unlike Real mode, which uses 64 Kilobyte segments, SMM is configured to allow 4 Gigabyte segments which allow access to a full 32 bit address space (4 Gigabytes) by manually specifying a 32 bit operand prefix. This mode is analogous to Unreal mode. SMM is not limited to operating in this operating mode. The SMM handler is free to configure a protected mode including paging or long mode to enable access to additional registers or functionality.

Once configured, SMM can only be entered via an SMI. However, multiple sources within the system can generate an SMI, including both external sources by asserting the SMI pin on the processor, or internal ones by executing certain instructions, including certain input and output instructions. For example, reading from the keyboard I/O pin could be trapped to allow an SMM handler to provide that data from another source. While in SMM, interrupts are disabled by masking the appropriate enable bits, including additional SMIs which gives SMIs highest system priority in regard to interrupts. For example, if a non-maskable interrupt, maskable interrupt or debug exception occurs at the same time as an SMI, only the SMI will be serviced. If an SMI occurs while the processor is in SMM, the first SMI to occur while the processor is in SMM will be serviced when the processor exits SMM. Only one SMI will be latched while the processor is in SMM. Additional SMM attempts will be ignored. Interrupts can be enabled while in SMM by enabling the appropriate bits.

SMM management mode is exited through a special RSM (Return from System Management) instruction. Upon encountering an RSM instruction, the processor restores the saved state that was initially stored to the SMRAM region and continues executing in the condition prior to the SMI. Other software, such as the operating system, is unable to know an SMI has been received and handled. Only by detecting lost CPU cycles or through the use of a logic analyzer monitoring the SMI pin can one be sure SMM has been entered.

The AMD Phenom II processor is part of the AMD family 10h processors that belong to the AMD64 architecture. Family 10h AMD processors include, in addition to the execution core, an embedded memory controller and Hyper Transport interconnect. In multiple processor machines, each CPU is connected to the coherent fabric. Events that generate SMIs do not do so directly. Instead, a message is generated on the coherent fabric and delivered to the appropriate CPU.

Configuration is performed through MSRs (Model Specific Registers). MSRs are a feature of x86 processors that allow access to vendor specific extensions through a generic mechanism. A register number is loaded into a general purpose register and the result is either read or written into another two registers respectively. This allows a wide range of possible registers.

The AMD64 architecture extends the SMRAM region to accommodate the 64 bit architecture. The specific SMRAM storage details are specific to the processor implementation though there is a large amount of overlap between Intel and AMD implementations.

MTRRs (Memory Type Range Registers) are used to provide access control to specific regions of memory. MTRRs primarily allow control over memory access cache policies. Memory access types include Uncacheable, Write-Combining, Write-Through, Write-Protect, Write-Back. Two types of MTRRs exist in the AMD 10h architecture, fixed-range and variable-range. Fixed-range MTRRs cover the first one megabyte of memory

with each MTRR register controlling a fixed size including 64K, 16K, 4K. Variable range MTRRs allow a variable range by combining a base and mask to specify the range. In addition to the primary access types, the AMD 10h architecture supports Extended Fixed-Range MTRR Type-Field Encodings which allow further control of access to DRAM by explicitly specifying if an access to the region covered by the MTRR will be routed to MMIO or to DRAM. Read and write access can be independently specified for the region. In addition to the protection provided by the MTRRs, the Phenom II processors also contain additional access bits for the ASEG and TSEG region. As part of the SMMMask MSR, the first two bits correspond to AValid and TValid respectively. When set, they indicate that the ASeg or TSeg are enabled for SMM use. If the respective bit is zero, access to the region is controlled entirely by the MTRR configuration. When set to one, if the CPU is in SMM access is dependant on the AClose and TClose bits of the SMMMask register. AClose and TClose allow the SMM handler to send data to the MMIO region which may be shared with the SMRAM region. By setting the appropriate close bit, memory accesses are directed to MMIO instead of DRAM for their respective regions. Finally, if the CPU is not in SMM while the valid bit is set, access is directed to MMIO.

The Phenom II processor provides two potential protected region of memory that can be used to store an SMM handler, the ASEG and TSEG. The ASEG region is located in the last 128 KB of the first megabyte of physical memory (physical address 0xa0000-0xbffff) and is protected by the fixed-range MTRRs. This region of memory is typically used for MMIO to the primary graphics device. As such, the default access type for this memory is Uncacheable (the recommended type for MMIO) and the extended fields are set such the accesses to this region will be mapped to MMIO instead of DRAM. The TSEG region analogous to the ASEG only resides at the top of memory which is also often used for MMIO. The TSEG region is covered by the variable-range MTRRs which do not support the extended field encodings supported by the fixed-range MTRRs though they can be controlled through the IORRs which provide these control bits. Thus, the ASEG provides more options in regard to access control.

The SMRAM region of memory's write access is controlled through the lowest bit of the HWCR (Hardware Configuration Register MSRC001 0015) known as the SMM Lock bit. This bit, when set, prevents modification to SMM BASE, SMMAddr, SMMMask (except TClose, AClose), and SMM CTL, which are read only, and SMIs are not intercepted in SVM. This bit can only be cleared by a system reboot unless an SMMkey has been specified. SMMkey is a 64 bit MSR that allows a key to be to be set by the write to the SMMKey MSR. Subsequent writes to the SMMKey MSR will clear the SMMLock bit if the same key is written. Incorrect keys are ignored. Once the bit is cleared, the SMM variables can again be modified.

Initial exploration attempts on the test Phenom II systems where very limited due to unexpected results and few methods available to learn what had happened or how to resolve the problem. To address these problems, three primary methods where used to explore the use of SMM on the AMD Phenom II processors. Initially, it was unknown if SMM was in use on the test systems though it was suspected. The methods included

replacing the vendor provided firmware with an open source version provided by the Coreboot project, writing a minimal operating environment to use as testing and finally a kernel driver to use from within protected mode linux operating system.

Coreboot provides a replacement system firmware for a number of supported chipsets and system boards, primarily AMD based since AMD contributes directly to the project. Our test systems included chipset that where known to work but the exact board was not supported. First attempts to replace the system firmware resulted in non-POSTing system. Our main boards included a ROM chip in a DIP socket which was easily removed and replaced with another working ROM. This method was used extensively while getting Coreboot onto the test system. In addition to swapping ROM chips, an EHCI debug port was used. Coreboot includes support for using the debug port provided by the SB700 south bridge as a serial port. This allows early debug output while the system is booting, before any other method, such as displaying on the screen, is available.

Coreboot does not provide an SMM handler for AMD chips. It does include some basic SMM support for older Intel chips but currently no SMM handler is available for Phenom II line of processors though it does provide some initial work to provide the framework to upload a handler however. With the vendor provided firmware, it was impossible to know all that the firmware had done as part of the boot process. This included whether an SMM handler was currently being used or not and what, if any functionality would be lost due to the absence of an SMM handler. With Coreboot working in place of the original firmware, it was possible to see that the system was functional without any SMM handler.

The second method used to explore SMM was a minimal operating environment that would provide a way to ensure no other operating system related changes affected the operation of the system firmware SMM handler. The minimal environment consisted of a small Multi-boot compatible executable which could be booted from grub. This environment would configure and enable the local APIC timer, configure a GDT and LDT to enable a flat memory model consistent with unreal mode as well as that of the SMM operating environment as discussed in chapter two and install basic interrupt handlers in required locations.

The final method was a kernel module and userspace tool that communicates with the kernel module that can be used from within the Linux operating system. The combination of these two; allows access the SMM handler memory region and can be used to modify the currently installed SMM handler. Sufficient operating system permissions are required to perform these modifications from within Linux system since access to model-specific registers is limited to protected code running in ring-0.

This work found that an SMM handler can be modified by an operating system and that not all protection mechanisms are being used by commercially available computer systems such as SMMLock bit and SMMKey where not being used by the BIOS vendor. In addition, it was possible gain access the protected regions of memory by modifying the MTRR's.

ACRONYM LIST

AMD	Advanced Micro Devices
AMP	Asymmetric Multi-Processor
APIC	Advanced Programmable Interrupt Controller
ASEG	A SEGent
ASIC	Application-Specific Integrated Circuit
BIOS	Basic Input/Output System
BKDG	BIOS and Kernel Developer's Guide
CC2MB	Cache Controller to MicroBlaze
COTS	Commercial Off the Shelf
CPU	Central Processing Unit
EDK	Embedded Development Kit
FPGA	Field Programmable Gate Array
GCS	Guarded Communication Subsystem
GDT	Global Descriptor Table
HAMES	High Assurance Middleware for Embedded Systems
HDL	Hardware Design Language
HT	HyperTransport
HWCR	Hardware Configuration Register
I/O	Input/Output
IP	Intellectual Property
ISE	Integrated Software Environment
LDT	Local Descriptor Table
LVDS	Low Voltage Differential Signaling
MILS	Multiple Independent Levels of Security
MMIO	Memory Mapped Input Output
MMR	MILS Message Router
MOESI	Modified-Owner-Exclusive-Shared-Invalid
MSR	Machine Specific Registers
MTRR	Memory Type Range Registers
NDA	Non-Disclosure Agreement
OS	Operating System
PCIe	PCI Express
QEMU	Quick EMUlator
QPI	QuickPath Interconnect
PCS	Partitioned Communication Systems
RAM	Random Access Memory
RSM	Return from System Management
SMI	System Management Interrupt
SMM	System Management Mode
SMP	Symmetric Multi-Processor
SMRAM	System Management Random Access Memory
UI	University of Idaho
USB	Universal Serial Bus
VLSI	Very Large Scale Integration