AN ABSTRACT OF THE THESIS OF

Nirut Chalainanont for the degree of Master of Science in Electrical & Computer Engineering presented on September 24, 2004.

Title: Design and Implementation of Configuration Modules in a Programmable Hardware-Assisted Cache Emulator (PHA$E) .

Abstract approved: _____

Alexandre Ferreira Tenca

Memory hierarchy design is becoming more important as the speed gap between processor and memory continues to grow. Investigations of memory performance have typically been conducted using trace-driven emulation, which could take tremendous resources (e.g. long emulation time, large storage requirements for traces, and high overall cost). Recent research has proposed the use of hardware for performing cache emulations. Such an approach is advantageous as it can be done in real-time, which eliminates the need for large storage for traces, reduces the emulation time, and improves the accuracy of the results. This thesis discusses the preliminary work with the Programmable Hardware Assisted Cache Emulator (PHA$E), a system for emulating cache in real-time. PHA$E is implemented using Field Programmable Gate Array (FPGA) chips, so it is flexible and configurable. Once configured, PHA$E can emulate various sizes of cache, different cache organizations, and many replacement algorithms. This thesis describes the design and implementation of Very High Speed Integrated Circuit Hardware Description Language (VHDL) modules that were programmed into PAH$E to make it capable of emulating off-chip shared level 3 caches with varying sizes and set-associativities. Furthermore, the emulation results from SPEC benchmarks (SPECcpu2000 and

SPECjAppServer2002 [13]), and a large vocabulary continuous speech recognition (LVCSR) system [24] are presented to verify the functionality of PHA$E. Lastly, future research directions are identified.

Design and Implementation of Configuration Modules in a Programmable Hardware-Assisted Cache Emulator (PHA$E)

by

Nirut Chalainanont

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 24, 2004
Commencement June 2005

Master of Science thesis of Nirut Chalainanont presented on
September 24, 2004

APPROVED:

Major Professor, representing Electrical & Computer Engineering

---

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Nirut Chalainanont, Author

ACKNOWLEDGMENTS

First of all, I would like to thank my mother, father, and sister who have been always supportive during my study here in the US. Further, I would like to thank Dr. Alexandre F. Tenca for his guidance throughout this thesis work. Finally, I would like to thank Dr. Shih-Lien Lu for giving me such a great opportunity to be a part of this research. This thesis could not be completed without his help.

This research is supported by Microprocessor Research Labs, Intel Corp.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDIX TABLES

Table Page

Remove this page. Since we couldn't fix a bug, this is still in. If you remove this part, you will see that the first page is not filled with text to the normal bottom edge of the text box. It stops about 1/2 inch higher than the other pages.

REMOVE THIS PAGE FOR YOUR OWN BENEFIT

DESIGN AND IMPLEMENTATION OF CONFIGURATION MODULES IN A PROGRAMMABLE HARDWARE-ASSISTED CACHE EMULATOR (PHA$E)

1. INTRODUCTION

With the continuously widening gap between CPU and memory speeds, obtaining a thorough understanding of memory hierarchy performance and behavior has become exceedingly important. Today's memory systems use layers of caches with varying sizes and speeds positioned between the CPU and the main memory to mitigate such differences in speed. These caches store frequently used data or instructions. By placing the faster, but consequently smaller, caches closer to the CPU, accesses to the slower caches and main memory can be reduced since frequently used data or instructions are made available in these faster caches.

In order to achieve the highest performance with the most efficient design, emulation is usually employed to assess memory design decisions. There are several ways to evaluate memory system performance [8]. Backend emulation using address reference traces is one of the methods [15]. Traces can be collected either through hardware or software means. After the traces have been collected, they are stored in some form and used by a backend (trace-driven) emulator to evaluate the effectiveness of different cache organizations and memory hierarchy arrangements. This method is repeatable and efficient. However its usefulness is limited by the size of the traces. Another common method used to evaluate memory hierarchy is through complete system emulation. In this approach, every component of a complete system is modeled, including the memory hierarchy. Full

system emulation is flexible and provides direct performance of benchmarks [6, 10]. However, full system emulation is less detailed and slower, which may limit the design space explored. It is also much more difficult to set up benchmarks on a emulated machine.

The ideal scenario is to emulate the memory hierarchy with hardware and perform a measurement of the actual system while executing real workloads. However, the real time emulation hardware is usually very costly and inflexible. It is difficult to explore all design spaces of interest using emulation. We employed an alternative method called *hardware cache emulation*.

Hardware cache emulator is a real-time hardware-based emulation environment. It watches the front-side bus and collects traces of an actual system while the system is running real application software. Additionally, it processes the traces with hardware in real time and stores only the statistical information.

That is, instead of collecting the traces and processing them with a software emulation program later on, the hardware cache emulator emulates the traces immediately. Since traces are processed directly and not stored, we are able to run for a much longer period of time and to observe long-term behavior of benchmarks. Hardware cache emulator also permits us to explore larger design spaces, including large cache effects of multi-processor systems. It removes some artifacts from software emulation having limited trace length.

Hardware cache emulator provides a complementary tool to full system emulation with software. Since it is running in real-time, it allows us to scan different workloads with multiple configurations. When the hardware cache emulator indicates the possibility that a larger cache is significantly beneficial, we can zoom in with the current hardware tracing methodology or full system emulation. Moreover, since we are implementing this emulator with FPGAs, we have the flexibility to modify the coherency protocol and study migratory sharing behavior. If there

are particular behaviors that can benefit from pre-fetching strategies, we could also add these pre-fetching algorithms into the emulator to study their impact on performance.

1.1. Basic Concept of Cache

Cache is the name given to some of the level in the memory hierarchy that a processor can access more quickly than it can access main memory. The purpose of cache is to speed up memory accesses by storing recently used chunks of memory. Cache memory is also described in levels of closeness and accessability to the processor. A Level 1 cache, called L1 cache, is the closest cache to the processor. It has the shortest access time, but is smallest in size. Second level cache, or L2 cache, is placed farther away from the processor. It has longer access time, but more capacity than L1 cache. Most processors have only L1 and L2 caches. However, this work explores the effectiveness of a third level (L3) cache which is used to bridge the gap between a small L2 cache and a slow main memory.

Since the cost of cache memory is much more expensive than the cost of main memory, it is impractical to replace the entire main memory with cache. However, we can have a small amount of cache that is sufficient to store some of the data. We choose data that has been most recently used and place them in the cache. This is a policy that takes advantage of spatial and temporal locality.

Spatial and temporal locality of memory accesses refers to the idea that if a certain chunk of memory has been accessed, that same chunk of memory, or memory near it, is likely to be accessed again in the near future. For example, when the processor executes consecutive instructions in memory, the next dozen of instructions are probably contained in a given contiguous block of memory locations, so the next instruction would be accessed close by. Similarly, if the

processor is accessing elements of an array, they will be located consecutively, so it makes sense to store a block of data that has recently been used.

When the processor wants to access memory at a certain address, it looks in the cache to see if the data is there. If the data is there, the processor gets it from the cache instead of going all the way to the main memory. This situation is called a *cache hit*. If the data is not in the cache, then it will be transferred from the main memory to the cache. This process takes longer because access time to the main memory is much longer than access time to the cache. This situation is called a *cache miss*. The delay from accessing main memory and transferring data between main memory and cache is called the *miss penalty*.

A very important factor in determining the effectiveness of the L2 and L3 caches relates to how the cache is mapped to the system memory. There are three different ways that the mapping can generally be done [5].

- *Direct Mapped:* This is the simplest way of mapping. Each memory block has only one place it can appear in the cache. The mapping is usually

  (*Block address*) MOD (*Number of blocks in cache*)

- *Fully Associative:* A memory block can be placed anywhere in the cache.

- *Set Associative:* A memory block can be placed in a restricted set of places in the cache. A *set* is a group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set. The set is usually chosen by bit selection:

  (*Block address*) MOD (*Number of sets in cache*)

  If there are n blocks in a set, the cache placement is called *n-way set associative*. Figure 1.1 shows an example of these three cache mapping techniques.

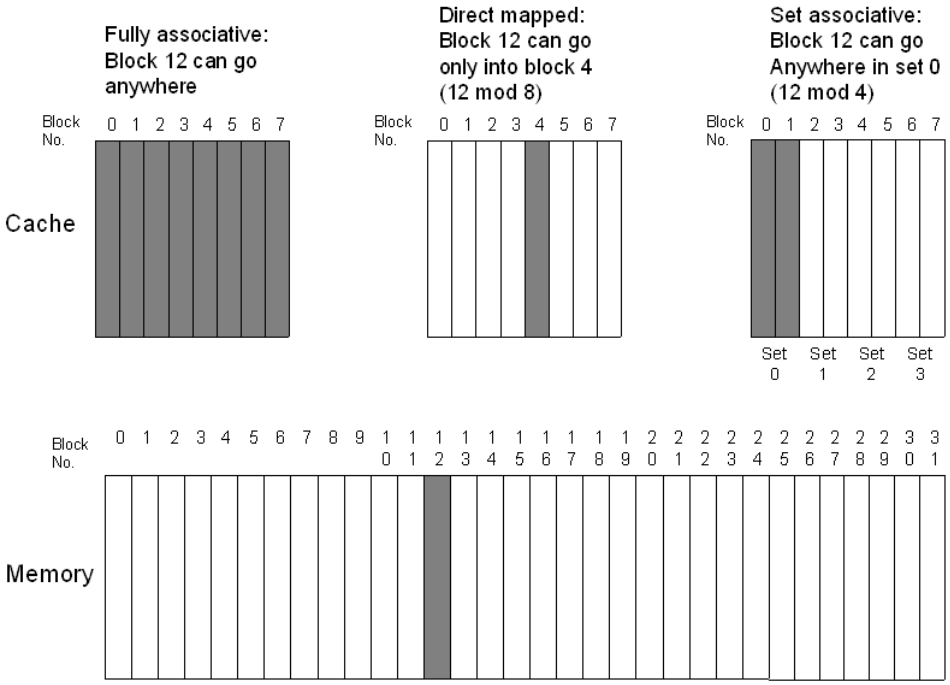

FIGURE 1.1: Example of cache mapping techniques [5]

There is a critical tradeoff in cache performance that has led to the creation of the various cache mapping techniques described in the previous section. In order for the cache to have good performance, we try to maximize both of the following:

- *Hit Ratio:* The goal is to increase as much as possible the likelihood of the cache containing the memory addresses that the processor wants. Otherwise, we lose the benefit of caching because there will be too many misses.

- *Search Speed:* Determine as quickly as possible if it is a cache hit or a cache miss. Otherwise, we lose a small amount of time on every access, hit or miss, while searching the cache.

The direct mapped cache is the simplest form of cache and the easiest to check for a hit. Since there is only one possible place that any memory location

can be cached, the line either contains the memory information the processor is looking for, or it doesn't. Unfortunately, the direct mapped cache also has the worst performance, because again there is only one place that any address can be stored. The hit ratio will be very low for direct mapped cache.

The fully associative cache has the best hit ratio because any line in the cache can hold any address that needs to be cached. This means the problem seen in the direct mapped cache disappears, because there is no dedicated single line that an address must use. However, this cache suffers from problems involving searching the cache. The searching process has to look through the entire cache to find the memory block that the processor wants. This penalty occurs for all accesses to memory, whether a cache hit occurs or not, because it is part of searching the cache to determine a hit. In addition, more logic must be added to determine which of the various lines to use when a new entry must be added (usually a *Least Recently Used* replacement algorithm is employed to decide which block in the cache will be used). All this overhead adds cost, complexity and execution time.

The set associative cache is a good compromise between the direct mapped and set associative caches. For example, a 4-way set associative cache has four times more space to store a memory block than a direct mapped cachce. As a result, the hit ratio of 4-way set associative cache will be much higher than the hit ratio of direct mapped cache. In addition, for every memory reference, the searching process has to search only four blocks in a set instead of searching the entire cache, as in the case of fully associative cache. Thus, the search speed of 4-way set associative cache will be a lot faster than the search speed of fully associative cache.

Table 1.1 summarizes the different cache mapping techniques and their relative performance.

Cache Type	Hit Ratio	Search Speed
Direct Mapped	Worst	Best
Fully Associative	Best	Worst
N-Way Set Associative	Very Good, Better as N Increases	Good, Worse as N Increases

TABLE 1.1: Different Cache Mapping Techniques and Their Relative Performance

1.2. Objectives of this Thesis

This thesis discusses the preliminary work with the Programmable Hardware Assisted Cache Emulator (PHA$E). Since PHA$E is an FPGA-based programmable hardware, it needs to be configured in order to work as a cache emulator. This thesis is focusing on the design and implementation of *Cache Update* modules that are programmed into FPGA chips to make PHA$E works as an L3 cache emulator. These modules are implemented using Very High Speed Integrated Circuit Hardware Description Language (VHDL). Once configured, PHA$E will be capable of emulating shared L3 cache sizes of 4KB up to 4GB. It is also capable of emulating multiple set-associativities (e.g. 1-way, 2-way, 4-way, and 8-way) in parallel. To verify that the Cache Update modules are working correctly, we emulate off-chip L3 cache with varying sizes on dual-processor and quad-processor systems running SPECjAppServer2002 benchmark, SPECcpu2000 benchmark [13], and a Large Vocabulary Continuous Speech Recognition (LVCSR) system [24]. The emulation results are presented and analyzed.

1.3. Organization of this Thesis

The rest of this thesis is organized as follows: chapter 2 presents several cache emulation methods that have been used before PHA$E. Chapter 3 shows the hardware architecture of PHA$E and details about the design and implementation of the Cache Update modules. Chapter 4 presents and analyzes the emulation results. Lastly, chapter 5 concludes this work and elaborates on future research directions.

2. PREVIOUS WORK ON CACHE EMULATIONS

There are many ways to investigate the performance and behavior of memory hierarchy. One of the most used approaches is using trace-driven emulation. There are three stages in trace-driven memory emulation: trace collection, trace reduction, and trace processing [8]. In the trace collection stage, the exact sequence of memory references of the workload under test is determined. In the next stage, unnecessary or redundant data is taken out. Only traces needed for the emulation are kept. Lastly, these traces are inputted to an emulator for evaluation of the memory design of interest.

Various techniques that can be used in each stage of trace-driven emulations are clearly described in a survey conducted by Uhlig and Mudge [15]. External hardware probes, microcode modification, instruction-set emulation, static code annotation, and single-step execution are some methodologies that can be used in trace collection. Additionally, ready-to-use traces are also available from sources such as the Brigham Young University's online trace repository [14] and New Mexico State University's TraceBase [9]. For the trace reduction stage, techniques such as trace compression, significant-event traces, trace filtering, and trace sampling may be utilized. Finally, the traces can be processed using software either by distributing the emulation in parallel (e.g. using multiple workstations) or by using multi-configuration emulation algorithms (e.g. an algorithm considers various design parameters in a single pass of the traces). Alternatively, full system emulators such as Simics and SimOS [6, 10] can perform all of the emulation stages described above. These emulators are capable of performing a very detailed emulation of virtually all aspects of a system.

Most of the techniques described above, however, suffer from various kinds of slowdowns and distortions. Collecting traces using hardware probes, for example,

is limited greatly by the size of the buffer available on existing logic analyzers. Thus, the traces collected suffer from a discontinuity problem. Other collection techniques would experience time and/or memory dilation, which may be caused by extra overhead induced into the system by the collection mechanism that alters the behavior of the system from the original state of the system without the collection mechanism in place. Such distractions eventually would reduce the accuracy of the collected trace and increase the collection time significantly. Processing the traces using software may also take a tremendous amount of time. Parallelizing the emulation work can reduce the emulation time. However, the network infrastructure used to distribute the work may be unreliable and can cause additional concerns [15].

To address the aforementioned issues, recent work has proposed the use of hardware for performing cache emulations in real-time. Traces are collected, reduced (if necessary), and processed by hardware as the system under test runs a workload of interest. The hardware generates the results of the emulation in the form of statistics of the parameters being investigated (e.g. misses, miss ratio, misses per instruction, etc). There are several real-time hardware cache emulators that have been reported in publications [3, 4, 11, 16, 17].

BACH [3, 4] is a hardware monitoring system capable of collecting traces of a running system. It interfaces with the System Under Test (SUT) via a logic analyzer. In spite of the buffer size limitation of the logic analyzer, BACH is capable of obtaining long traces by issuing a high priority interrupt each time the buffer is filled. Once the content of the buffer is transferred to a disk, the interrupt is de-asserted. Then, the SUT continues its run and the proceeding traces are collected. Unfortunately, such an interrupt mechanism requires alteration of the original SUT and, therefore, introduces minor dilations. Flanagan et al. [3]

reported that the dilations added 1.125% more references to the total references of an otherwise untraced machine.

In RACFCS [17], instead of using a logic analyzer to obtain the traces, a Latch board (with a FIFO buffer) that directly connects to the output pins of microprocessors is used. The Latch board is connected to a Trace board, which has SRAMs for trace data storage. Additionally, RACCFS provides the Flying Cache emulator to process the collected traces on the fly. The emulator interfaces with the trace board. As the board collects traces from the system bus, the emulator emulates the traces using the cache structure of interest in real-time and counts the number of cache hits and misses. EPLDs (Altera MAX7000) are used to implement the control logic of the system. Thus, RACCFS can be reconfigured depending on research needs.

HACS [16] was built as an improvement of BACH. HACS uses a FIFO board connected directly to the system bus of the SUT for collecting traces. This board is then connected to an FPGA board sitting on a host system via the SCSI interface. The FPGA board processes the traces coming out of the FIFO in real-time. The FPGA board also has a PCI interface, in which the emulation results can be obtained by the host machine. A limitation of HACS is that it cannot process at the maximum possible rate of traces generated by the system. Thus, there is a possibility that an emulation needs to be interrupted in the middle of a workload run. Regardless, emulation works using HACS showed that the maximum trace generation rate had never occurred.

MemorIES is a real-time hardware emulator developed by Nanda et al. at IBM [11]. Unlike the emulators discussed previously, MemorIES is capable of processing traces at the same speed as the bus, and, thus, does not require the use of any interrupts for halting the system in the case of buffer overflow. It uses 7 FPGAs to implement the programmable cache controller functions and SDRAMs

(e.g. 1 GB) to store the cache tag and state tables during the emulation. MemorIES connects to the 6xx memory bus of IBM's S70 class RS/6000 or AS/400 servers. A unique feature of MemorIES is that it can emulate L2 in addition to L3 caches by turning the actual L2 cache off.

3. PHA$E HARDWARE SETUP AND VHDL MODULES

This chapter presents the reasons to consider the development of another cache emulation system and provides the most important information about its architecture and features. Since PHA$E is an FPGA-based emulation system, we first present the hardware design and then identify the hardware components that were mapped to the FPGAs.

3.1. Why PHA$E?

PHA$E is unique in comparison to the previously discussed emulators in several ways. First, PHA$E operates at up to the 133 MHz bus speed, thus addressing the possible effects of dilations when using the interrupt to handle buffer overflow as in the case of BACH, RACCFCS, and HACS. It does so by distributing the trace processing over four Xilinx XC2V1000 FPGAs. Each FPGA consists of two cache controllers that process the traces. More details on this will be discussed later.

The second uniqueness of PHA$E is that it supports different hardware platforms and cache parameters compared to existing emulators. Currently, PHA$E works on Intel IA32 architecture (e.g. Pentium Pro, Pentium II, Pentium III) and is capable of emulating shared L3 cache sizes of 4KB up to 4GB. Since PHA$E is designed with two modules, only the front-end module needs to be changed when a different platform is emulated.

Lastly, PHA$E is capable of performing emulation of multiple set-associativities (e.g. 1-way, 2-way, 4-way, and 8-way) in parallel. This is useful not only to reduce emulation time, but also to partially address the lack of repeatability [16] . It is

difficult for real-time cache emulators to recreate an exact emulation run, unless the trace of the emulation is stored and reused. By parallelizing the emulation, however, the same traces are used during a run. Therefore, PHA$E enables a more accurate comparative study of cache with differing set-associativities. However, due to large memory tags storage requirement, parallelizing emulation of a reasonably wide range of cache sizes is not implemented at this time and is left for future study. Emulation of cache with varying sizes is done by running multiple runs, each with its desired cache size parameter.

PHA$E also implements features offered by other emulators. First, FPGAs are used for trace filtering and processing. Thus, PHA$E can be re-programmed easily by modifying the necessary logic behavior in the FPGAs. Second, PHA$E can support up to eight 512MB SDRAMS, which offers a maximum of 4GB total memory for storing data during emulation. For this work, however, is configured with eight 256MB SDRAMs.

3.2. PHA$E Hardware Setup

There are five Xilinx XC2V1000 FPGAs on the PHA$E board. Four of these FPGAs are connected to eight SDRAM DIMMs (two SDRAM DIMMs per each FPGA). These SDRAM DIMMs are used as storage space for cache address information (TAGs) of the emulated cache. The PHA$E board is connected to the System Under a Test via Tektronix Logic Analyzer Interposer (LAI). PHA$E gets memory reference information from the SUT's processor(s) through this interface. The board is connected to the host system via a PCI slot. The FPGAs on the PHA$E board can be programmed to send out any information we need to the host system through the PCI bus. There is also a Xilinx Progamming Interface which is used to download programming information from the Xilinx ISE tool to

the FPGAs. Lastly, there is a reset switch which provides a global reset to every FPGA on the board. Figure 3.1 shows the block diagram of PHA$E board and how it is connected to the SUT and the host system. The PHA$E hardware board and computer systems used in this research are provided by the Microprocessor Research Labs (MRL), Intel Corporation, Hillsboro, OR. All the work for this thesis was done at the same location.

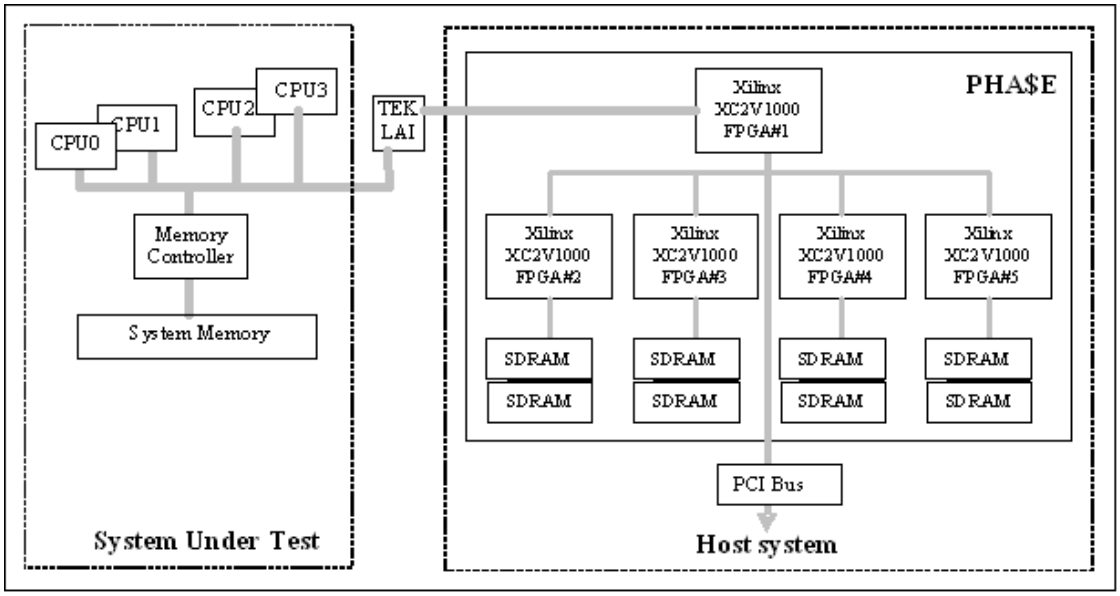

FIGURE 3.1: PHA$E Hardware Setup

3.3. VHDL Modules for PHA$E

The work done in this thesis consists of programming the FPGAs to create an off-chip shared L3 cache emulator and using this system to obtain experimental

data to evaluate the behavior of these L3 caches. This section provides description and functionality of each VHDL module that was programmed to the FPGAs.

3.3.1. Request Filter

FPGA#1 in Figure 3.1 is programmed to be a Request Filter FPGA (RF). This module is responsible for interfacing to the front-side bus of the processor and filtering out transaction requests that are not emulated. Transactions are identified and the proper information is queued in the RF according to the bus protocol. Once all of the information for a transaction has been collected the transaction will be stored in the FIFO.

Figure 3.2 depicts the major blocks within the RF. The Request FIFO in this figure is used to buffer the bus activities. There can be up to eight outstanding transactions in the front side bus at the same time. We have eight distinct state machines to keep track of these outstanding transactions. Transactions may be completed out-of-order if they are deferred. Once the transaction is complete, it will be pushed to a FIFO. The RF dispatches a request from the FIFO and sends it to the other four FPGAs when it is requested. The details of how to design and implement the RF module is outside the scope of this work.

3.3.2. Cache Update

FPGA#2, 3, 4 and 5 are programmed to be Trace Processor FPGAs. These four FPGAs are identical in functionality, which is to process the filtered memory requests stored in the Request Filter FPGA. The reason why there are four FPGAs to process the memory requests is because the rate of memory requests

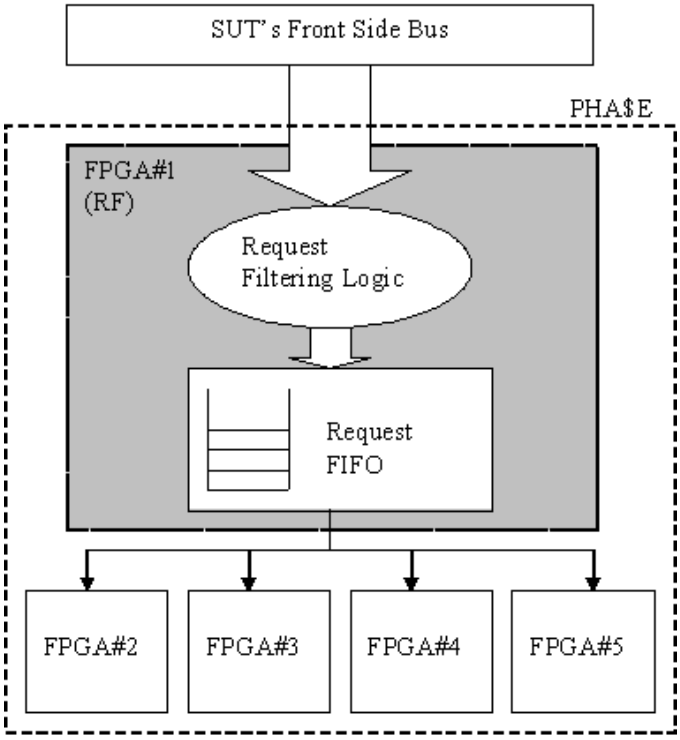

FIGURE 3.2: Block Diagram of Request Filter FPGA

generated by the SUT's processors is faster than the processing speed of one Trace Processor FPGA. Figure 3.3 depicts a block diagram of a Trace Processor FPGA. The Trace Processor FPGA consists of: two Cache Update modules, two SDRAM Interface modules (one for each SDRAM DIMM), an Event Counters module, and a PCI Interface module. The Cache Update is a VHDL module that is the core of PHA$E. It can be written to make PHA$E emulate several different kinds of cache organizations, set-associativies, replacement algorithms, and cache update protocols. This section will explain the design and implementation of the Cache Update module that makes PHA$E able to emulate shared L3 cache with several set-associativities (e.g. 1-way, 2-way, 4-way, and 8-way) in parallel and using a Least Recently Used (LRU) replacement algorithm.

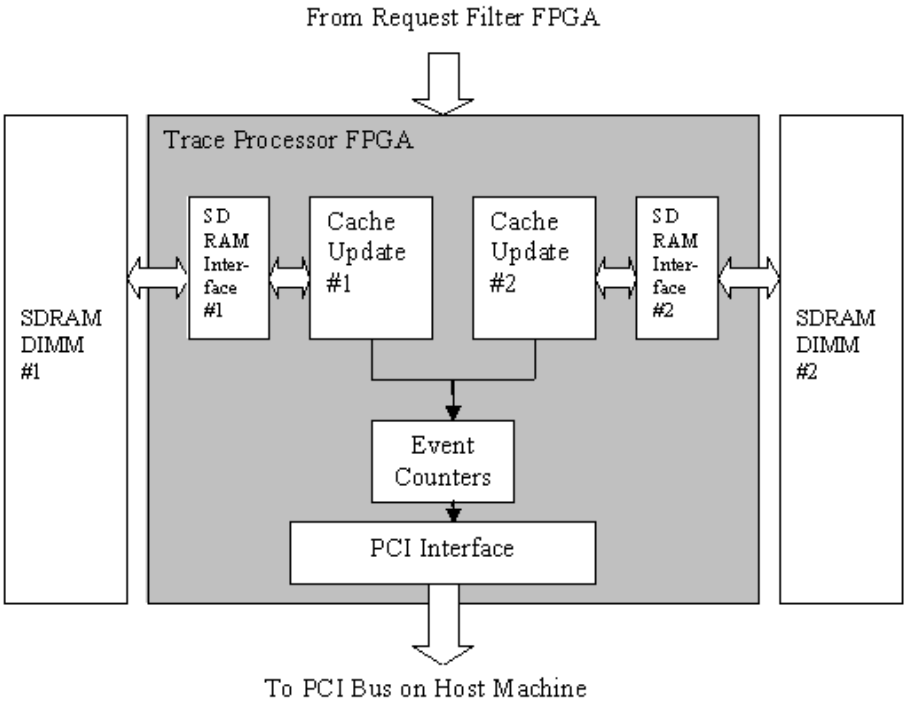

FIGURE 3.3: Block Diagram of Trace Processor FPGA

The Cache Update keeps track of the request queue. Whenever the queue is not empty, the Cache Update reads one request from the queue to process. Once the Cache Update gets the request it performs:

- Determine a set address in the request and send it to the SDRAM Interface module along with a *read* signal to read a cache line from the SDRAM. This cache line contains several TAG addresses. Since we emulate 1-way, 2-way, 4-way, and 8-way at the same time, the total for one cache line is 15 ways or, in other words, 15 TAG addresses. These TAGs will be compared with the TAG from the request to determine cache hit or cache miss.

- Wait for the cache line to be available from the SDRAM. The width of the cache line is 60 bytes (4 bytes per TAG), but the data width of the SDRAM

DIMM is 8 bytes. As a result, to read or write a cache line to/from the SDRAM needs 8 cycles to complete.

- Compare all TAGs in the cache line with a TAG from the request. The Cache Update compares all 15 TAGs in parallel and determine which one is a cache hit or a cache miss. Note that higher set-associativities have more chance of cache hit than the lower set-associativities. For instance, it could be a cache miss in 1-way and 2-way but be a cache hit in 4-way and 8-way.

- Shift and update the cache line using the Least Recently Used algorithm. To understand how LRU algorithm works, here is an example of a 4-way set-associativity. Figure 3.4 shows the block diagram of 4-way shift register. Register A is the Most Recently Used (MRU) position and Register D is the Least Recently Used (LRU) position. On cache miss, the TAG from the request will be written to register A which is the MRU. A TAG that was previously in register A will be shifted to register B, and a TAG in register B will be shifted to register C and so on. A TAG in register D which is the LRU TAG will be replaced. Everytime the LRU TAG is replaced the *Replaced* signal will be asserted. Moreover, if the replaced TAG has a *modified* status the *Write_back* signal will be asserted too. On cache hit, the hit TAG will be moved to register A and all the other TAGs will be shifted down. For example, a hit TAG is in register C. This TAG will be moved to register A, a TAG that was in register A will be shifted to register B, and a TAG in register B will be shifted to register C. TAG in register D will not be shifted. In case of cache hit, no TAG will be replaced. If the hit TAG is already in register A, no shifting is required.

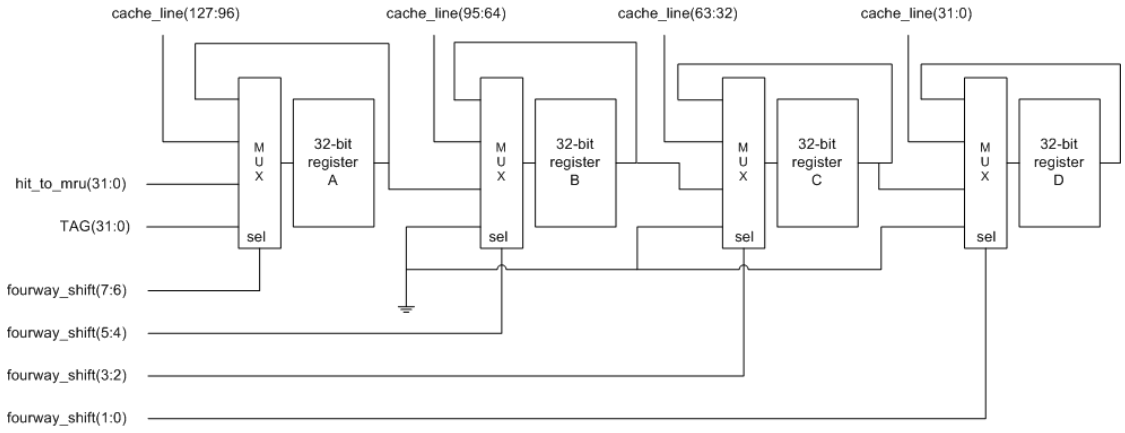

FIGURE 3.4: Block Diagram of 4-way Shift Register

In addition to shifting the TAGs, The Cache Update module needs to update the TAG status too. There are three TAG status used in this emulator: *Valid*, *Invalid*, and *Modified*. Upon initialization, all status in the SDRAM are marked as *Invalid*. If the request type is *read*, the TAG that associates with the request will be marked as *Valid*. On the other hand, if the request type is *write*, the TAG will be marked as *Modified*. However, *Modified* has a priority over *Valid*. Once the status has been changed to *Modified* it will stay *Modified* until the TAG is replaced even if there are more read requests refer to this TAG.

- Write modified cache line back to the SDRAM by sending the set address, data and *write* signal to the SDRAM Interface module.

- Send signals to update statistics counters in the Event Counters module. These signals are cache hit, cache miss, cache line replaced, and cache line write back for all set-associativites.

- Wait until the write operation is done then read the next request from the request queue.

The state diagram of the Cache Update is shown in Figure 3.5. Descriptions of each state are presented in Table 3.1.

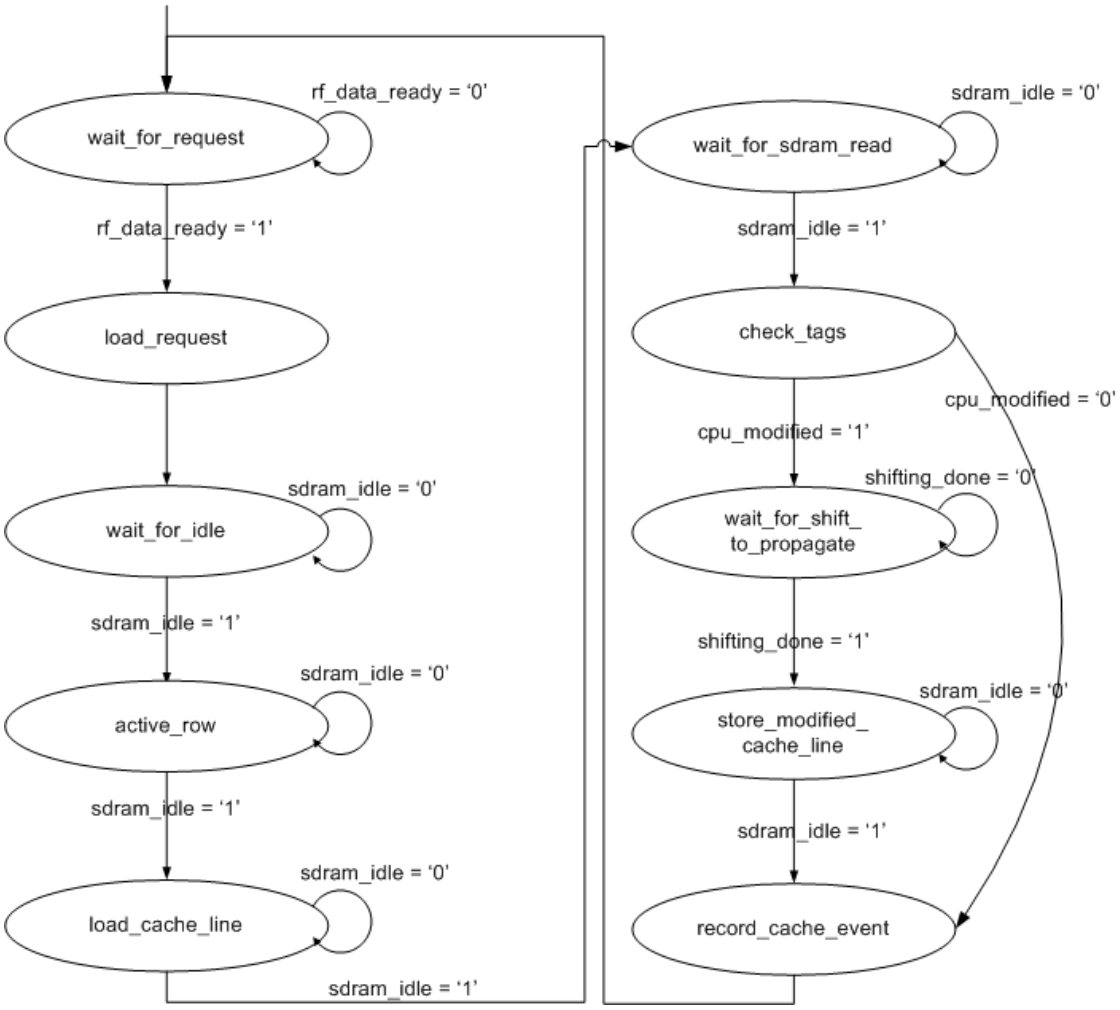

FIGURE 3.5: Cache Update State Diagram

Figure 3.6 shows input and output ports of the Cache Update module.

State	Description
wait_for_request	Check for request in the queue. If the request is ready, go to next state. If the queue is empty, stay at the same state.
load_request	Send *read* signal to read one request from the request queue.
wait_for_idle	Wait for the SDRAM to be idle before moving to next state.
active_row	Provide a memory address to the SDRAM Interface module to activate the row in the SDRAM that contains the cache line.
load_cache_line	Send *read* signal to the SDRAM Interface module to read the cache line from the SDRAM.
wait_for_sdram_read	Wait until read operation is done and the cache line is available to the Cache Update module.
check_tags	Compare all TAGs in the cache line with a TAG from the request. If there is a cache hit on non-MRU position, or a cache miss, or any of the TAG status needs to be changed, the cache line needs to be modified. In this case *cpu_ modified* signal will be '1' and the state machine moves to *wait_for_shift_to_propagate* state. On the other hand, if there is a cache hit on MRU position and none of the TAG status needs to be changed, the cache line does not need any modification and the state machine will go directly to *record_cache_event* state.

State	Description
wait_for_shift_to_propagate	Wait while the cache line is being modified (shifted).
store_modified_cache_line	Send *write* signal along with the modified cache line to the SDRAM Interface module. The modified cache line will be written back to the SDRAM.
record_cache_event	Activate counters in Event Counters module to record all statistics for this memory request. Then go back to *wait_for_request* state and wait for the next request.

TABLE 3.1: Description of Cache Update States

3.3.3. Other Modules

3.3.3.1. SDRAM Interface

The SDRAM Interface module is responsible for interfacing the Cache Update module with the SDRAM DIMM. For *read* operation the SDRAM Interface gets the column address, row address, and bank address from the Cache Update module. Then, it activates a state machine for multiple cycles (bursts) read. For each read burst the SDRAM Interface gets 8 bytes of data from the SDRAM DIMM and put it in a register. As mentioned before, the width of one cache line is 60 bytes, so the state machine has to go on for 8 cycles to read one cache line out of the SDRAM DIMM. Once all 60 bytes of the cache line are ready in the register, the Cache Update module will read the cache line from the register and the SDRAM Interface

will go to standby state waiting for the next operation. The *Write* operation is similar to *read*. The SDRAM Interface sends the column address, row address, and bank address to the SDRAM DIMM. Then, the state machine will write 8 bytes of data per cycle to the SDRAM DIMM. The *Write* operation is done when all 60 bytes of data is written to the SDRAM DIMM. The SDRAM Interface also provides setting values to the SDRAM controller such as CAS latency, RAS to CAS latency, refresh period, burst type, operating mode, etc.,

3.3.3.2. Event Counters

This module consists of several counters. These counters are 32-bit registers that are activated by signals from the Cache Controller state machine. Each counter keeps track of one statistics value according to the results from memory reference. In a private cache design, these counters keep track of all values for every CPU. This helps us to see behavior of each CPU individually. Since PHA$E can perform emulation of multiple set-associativities in parallel, in shared cache design, these counters will store values for each set-associativity seperately. As a result, we can see the difference in each one of the cache associativity easily. These are some examples of the values we collect.

- Total references.
- Number of hits, misses, replaced, and write back for all set-associativies.
- Type of memory references (e.g. read code, read data, write, write back, read invalidate).
- Memory addresses.
- Bus Cycle.

3.3.3.3. PCI Interface (Console Interface)

This module passes on emulated statistics from the Event Counters module to the PCI bus on the host machine. There is a C program running on the host machine to collect these statistics from the PCI bus every one second. These statistics will be written into a text file as a emulation result.

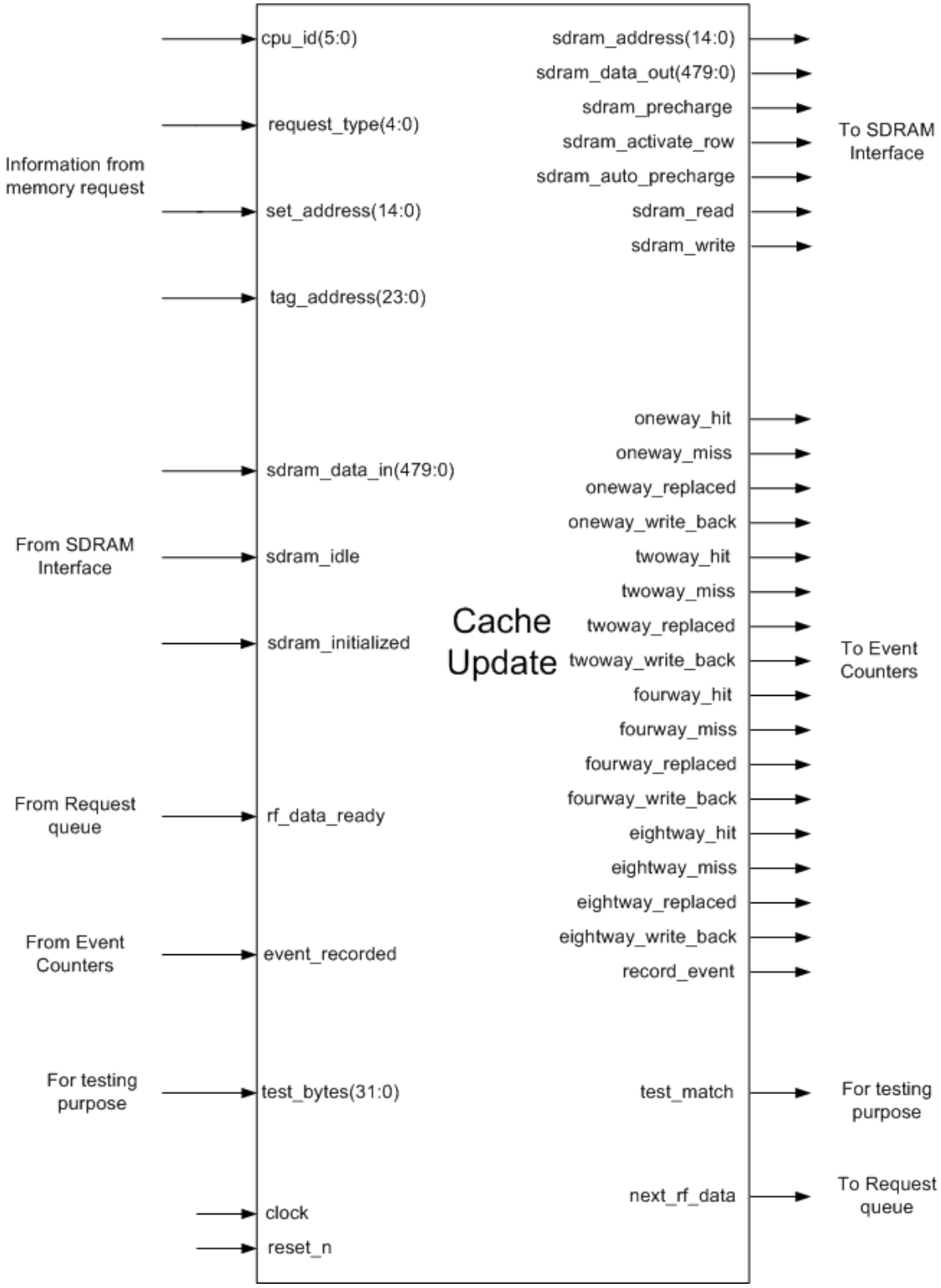

FIGURE 3.6: Cache Update Input and Output Ports

4. EXPERIMENTAL RESULTS AND ANALYSIS

PHA$E was used to emulate several shared external L3 cache configurations on dual-processor and quad-processor systems using three different workloads. The processors used in this research are Intel Pentium III Xeon^{TM} 550 MHz. Each processor has 32KB L1 data and instruction caches, and a 2MB unified L2 cache. PHA$E is used to obtain statistics (e.g. number of reads, writes, writebacks, read invalidates, and hits) from the emulated L3 cache for varying cache sizes with 1-way, 2-way, 4-way, and 8-way set-associativities. Miss ratio is derived based on dividing the number of misses by the number of references seen by the emulated L3. Even though PHA$E supports up to a 133 MHz bus, the SUT uses a 100 MHz bus only. Eight PC100 256MB SDRAMs are used as the memory tag storage. Additionally, we utilize the Pentium performance counters [12] to collect instruction and clock cycle counts. The unnecessary background services are disabled to reduce the perturbation of the system. Further, instead of logging in to the system, the benchmarks and PHA$E control scripts are executed remotely.

4.1. System Validation

Several validation tasks are performed to evaluate the correctness of the functionality of PHA$E. For the back-end validation, a VHDL module called "test request generator" is devised to evaluate each of the trace-processor FPGAs. When the trace-processor FPGA uses the test generator, the real filtered traces sent by the request-filter FPGA are ignored. Instead, the generator creates test traces based on a programmed pattern. Three patterns of trace generation are used in our evaluation. (1) The increasing address pattern accesses the memory from the

lowest to the highest possible address. This pattern evaluates whether PHA$E can correctly access all of the memory range available. (2) The Hot-spot emulating pattern intensely accesses a group of memory locations. (3) The Multiple hot-spots emulating pattern hops around among several groups of frequently accessed memory locations. These patterns are designed to yield particular output statistics (e.g. miss ratio, number of reads and writes, etc). Thus, the outcome of PHA$E should match the designed parameters when running the patterns. PHA$E performed as expected for all of these patterns. Thus, we are confident that the trace-processor FPGAs are working properly.

For the front-end validation, performance counters are used to check whether the request-filter FPGA properly captures the memory references from the system bus. Two validation methods are employed for each of the workload runs. (1) The number of references on the bus is derived from the performance counters and matched with the number obtained from PHA$E. We found that there is a maximum of 4% difference from all the workloads we run. (2) The number of references is plotted in relation to time. The plot from the performance counters is compared against the plot from PHA$E. The shapes of the plots for all of the workloads are virtually identical. Therefore, based on these two testcases, it could be concluded that the request-filter accurately captures the number of references and the timelines at which they occur.

4.2. Java Application Server

4.2.1. SPECjAppServer2002

SPECjAppServer2002 [20] is an industry standard benchmark designed to measure the performance of Java 2 Enterprise Edition (J2EE)^{TM} [19] application servers. The workload emulates a heavyweight manufacturing, supply chain man-

agement, and order/inventory system representative of one in use at a Fortune 500 company. It stresses the ability of EJB containers to handle the complexities of memory management, connection pooling, passivation/ activation, caching, etc.

There are four parts of the SPECjAppServer2002 benchmark. (1) The driver models customers and manufacturers that induce transactions to the system. The driver is implemented as a Java application. It creates threads that emulate the transactions. (2) The supplier emulator models the supplier domain of the system, which is implemented as a collection of Java servlets. (3) The database provides the repository capabilities for the corporate, orders, manufacturing, and supplier transaction data. (4) The application server models the middle-tier that handles the presentation logic (servlets) that serve the driver and supplier emulator, and implement the business rules (EJB) that prepares the information for the presentation logic and communicates with the database.

4.2.2. Emulation Parameters

In this research, we emulated the SPECjAppServer2002 workload running on shared cache design with varying cache sizes from 4MB to 1GB with doubling increments. The set associativity of 1 to 8 is used for all the emulated caches. The size of the application server heap is 1.5GB. The 1.5GB heap size is chosen due to the 2GB physical memory boundary. We use Bea Weblogic Platform JRockit JVM version 8.1 with Xgc:parallel option for parallel garbage collection method. Details of garbage collection methods can be found in [20].

In order to see the memory behavior of multi-processor systems, we run SPECjAppServer2002 workload on dual-processor and quad-processor systems. We set the injection rate for the dual-processor system to 60, and set it to 200 for the quad-processor system. These injection rates were selected because they

give approximately 98% of the observed average CPU utilization during steady state of the workload run. We believed that such CPU utilization is sufficiently high for our cache analysis. Several tuning techniques [21, 22] can be used to further improve the performance of the workload but that was outside the scope of this study. The trigger time is set to be 60 seconds for the dual-processor system, and 230 seconds for the quad-processor system. Lastly, for both systems, ramp-up time is set to 120 seconds, steady-state time to 300 seconds, and ramp-down time to 60 seconds. Data collection is performed during these three stages.

4.2.3. Emulation Results and Discussion

First, we take a look at the programming phase of the workload. Figure 4.1 illustrates the phase of a SPECjAppServer2002 run on an emulated 32MB 8-way L3 cache. The figure shows a very small amount of memory references seen by the emulated L3 in approximately the first 230 seconds during the trigger phase. The ramp-up, steady-state, and ramp-down phases happen in the next 480 seconds. We can see that the number of memory references as well as misses jumps drastically, which consequently increases the miss ratios (e.g. at about 20% level). Several spikes in the graph indicate garbage collection operations. During garbage collection, miss ratio jumps up due to the cleaning up of memory space. Once the garbage collection is done, miss ratio comes back down to its normal rate. Finally, the workload enters the phase where it collects statistics from the run, which last for approximately 100 seconds. At this stage, the number of memory references and misses drops.

Next, we look at miss ratios collected from shared emulated L3 cache. Figure 4.2 and Figure 4.3 show miss ratios of SPECjAppServer2002 running on shared

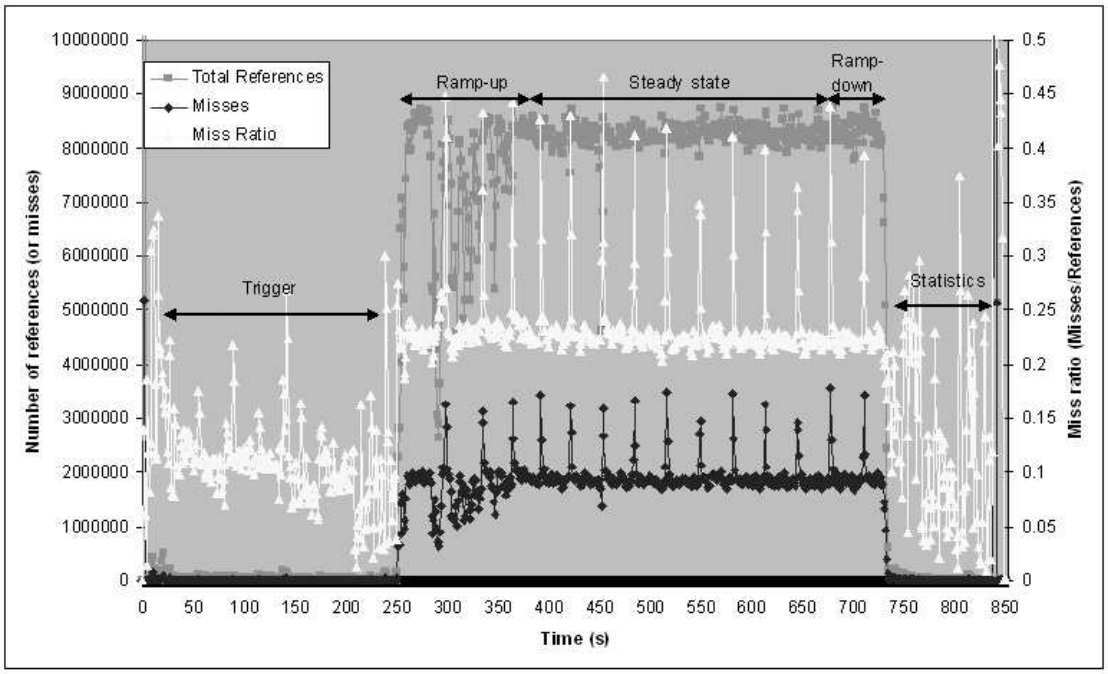

FIGURE 4.1: The Phase of a SPECjAppServer2002 Run on a emulated 32M 8-Way L3 Cache

emulated L3 cache with various cache sizes and set-associativities. In general, the bigger cache and higher set-associativity provides a lower miss ratio. These data show us that by increasing cache size and set-associativity, we can reduce capacity misses which results in improvement of overall miss ratios. These results confirm that PHA$E is working correctly.

4.3. SPEC CPU2000

SPEC CPU2000 is a CPU intensive benchmark suite, provided by Standard Performance Evaluation Corporation (SPEC) as a tool for measuring the perfor-

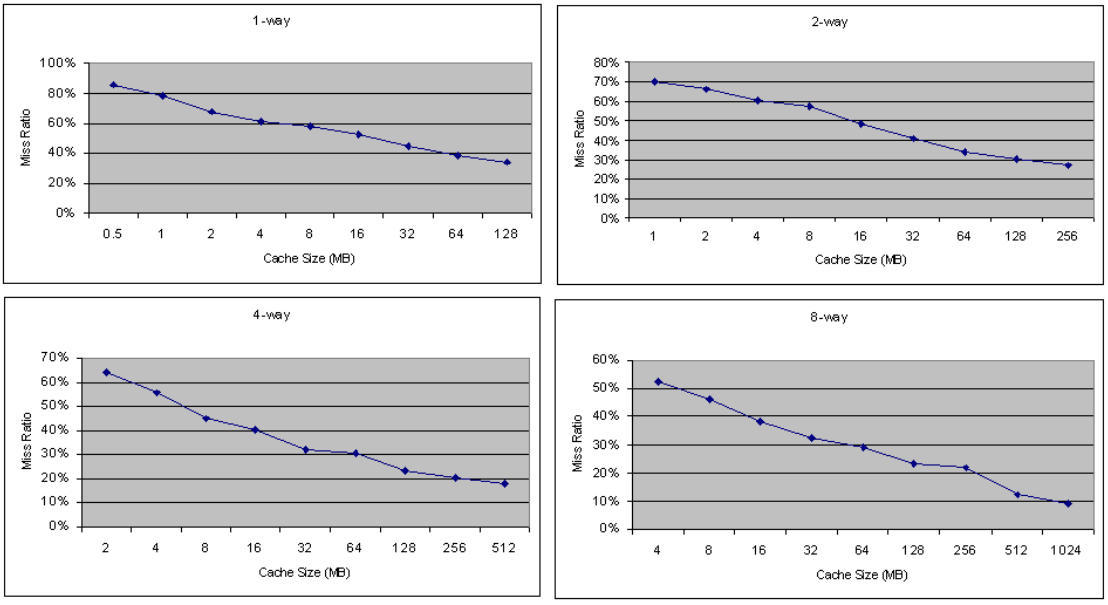

FIGURE 4.2: Miss Ratios of SPECjAppServer2002 running on a dual-processor system for Various L3 Cache Sizes and Set-Associativities

mance of the processor, memory and compiler of a system. We vary the cache size by changing the number of the set-address bit (e.g. from 11 to 19 bit). We start at an 11-bit set address, which is the smallest cache size, and increase one bit at a time until we reach 19-bit, which is the biggest cache we emulate. For 11-bit set address, the emulated cache size is 0.5MB for 1-way, 1MB for 2-way, 2MB for 4-way, and 4MB for 8-way. Increasing one set-address bit will result in double in emulated cache size. For example, using 13-bit set address will result in the cache sizes of 2MB for 1-way, 4MB for 2-way, 8MB for 4-way, and 16MB for 8-way. We do a single run for each of the set address bit numbers. The miss ratios for a run of four different benchmarks for different set-associativities and varying cache sizes are presented in Figure 4.4. In general, the miss ratio decreases as the cache size is increased. The saturation point seems to be at 18-bit set address.

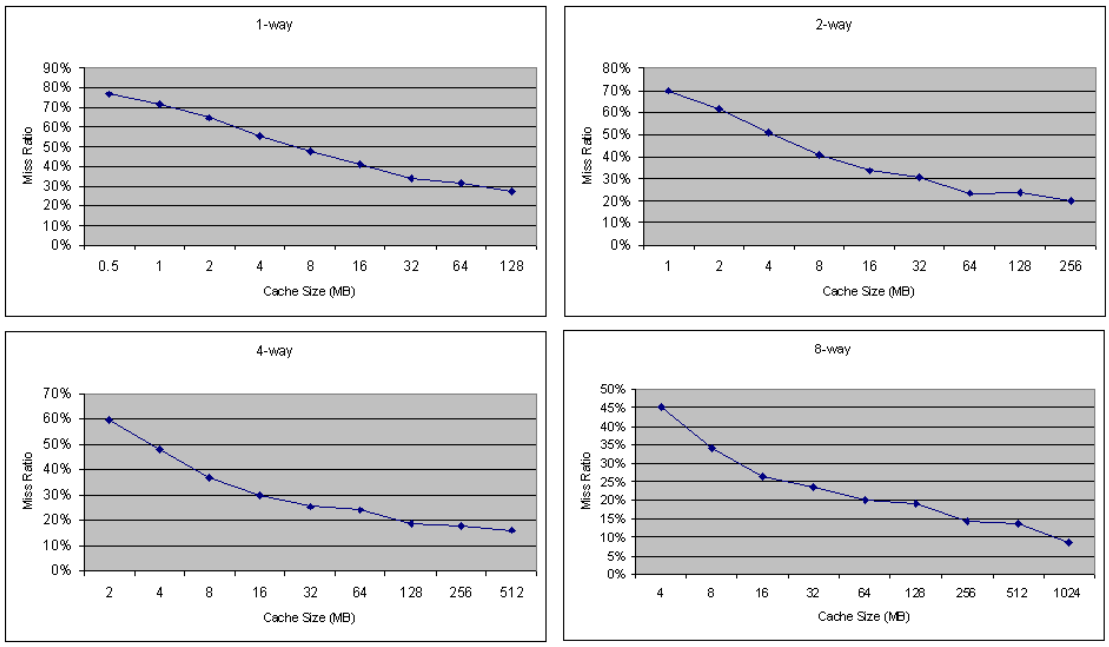

FIGURE 4.3: Miss Ratios of SPECjAppServer2002 running on a quad-processor system for Various L3 Cache Sizes and Set-Associativities

4.4. Large Vocabulary Continuous Speech Recognition

We analyzed the L3 cache behavior of the large vocabulary continuous speech recognition (LVCSR) system using PHA$E. LVCSR is a software capable of recognizing Chinese (Mandarin) utterances. The system was developed by Intel Labs, Intel China Research Center based on the LVCSR engine that was originally licensed from Oregon Graduate Institute (OGI) [23, 18]. Lai et al. in [24] had investigated the behavior of L1 and L2 caches in detail. However, L3 cache behavior was not explored. Furthermore, our SUT has a different configuration than theirs.

Figure 4.5 shows the miss ratios of LVCSR for various L3 cache sizes and set-associativities. The rate of the miss ratio reduction starts to slow down as the cache size increases above 256MB (see the graph for 4-way and 8-way). Furthermore,

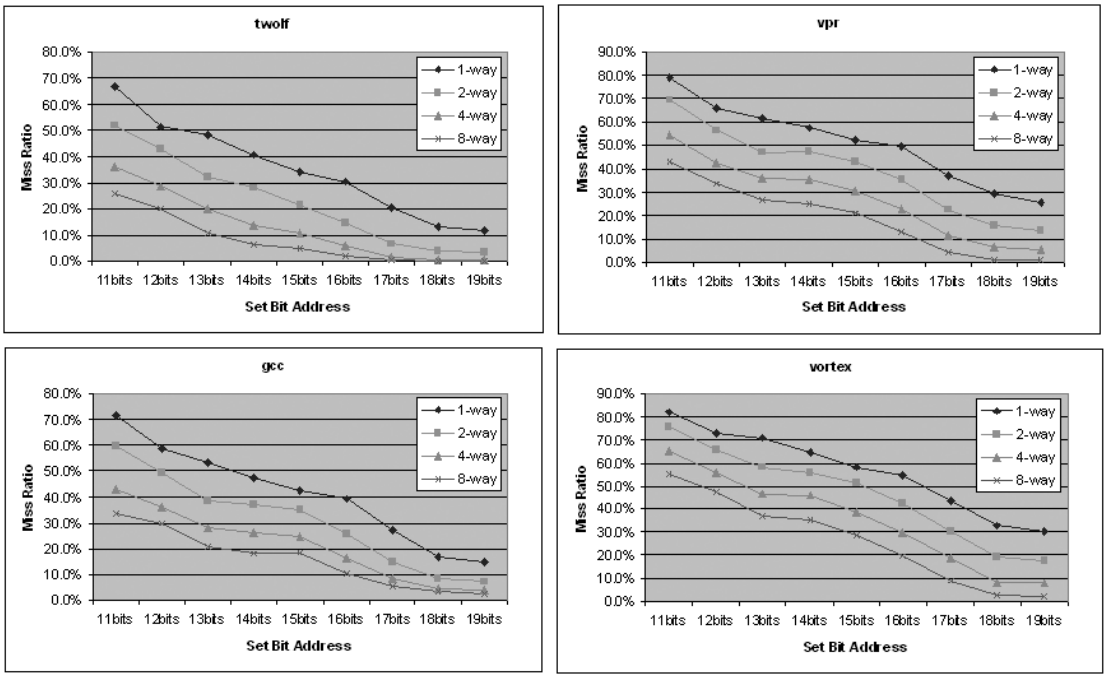

FIGURE 4.4: Miss Ratios of SPEC CPU2000 for Various Cache Sizes and Set-Associativities

at 512MB cache size with 4-way and 8-way set-associativity, the miss ratios have dropped to under 5% (e.g. 4.3% and 2.2%).

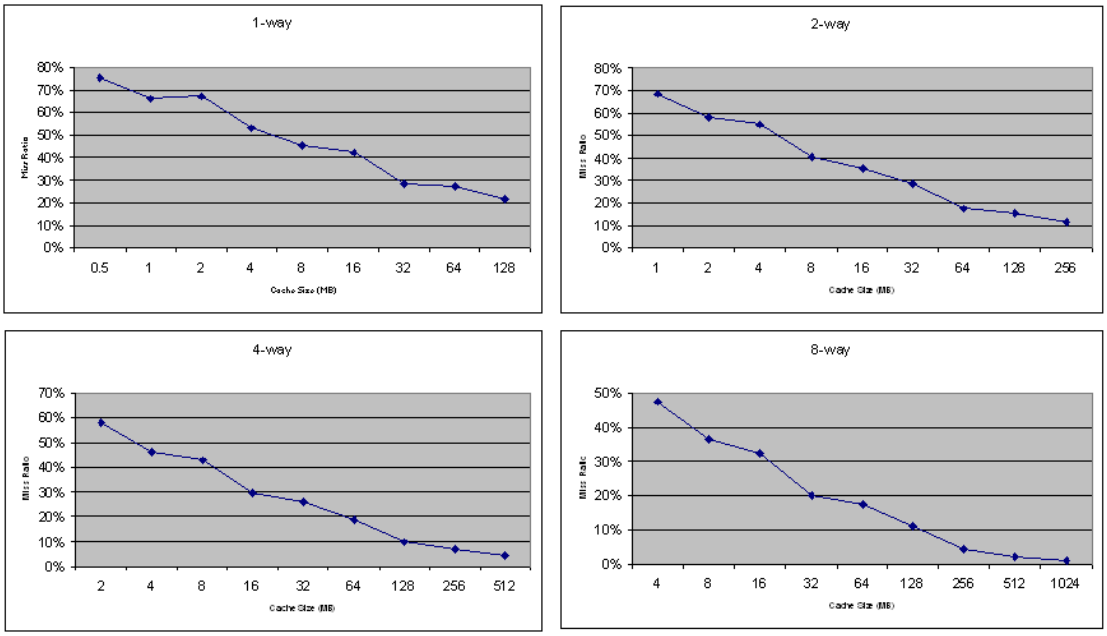

FIGURE 4.5: Miss Ratios of LVCSR system for Various L3 Cache Sizes and Set-Associativities

5. CONCLUSION AND FUTURE WORK

PHA$E is useful for investigating cache hierarchy efficiently and effectively due to its real-time capability and programmable feature. The work done in this thesis consists of programming the FPGAs to create an off-chip shared L3 cache emulator and using this system to obtain experimental data to evaluate the behavior of these L3 caches. Details of VHDL modules that were programmed to the FPGAs were described. Results from several SPEC workloads show that the system is working correctly. From Figure 4.2, 4.3, 4.4, and 4.5 we can see that the behavior of these emulated L3 caches are consistent with the conceptual behavior of cache memory. For example, the miss ratio reduces as the cache size is increased, and high set associative caches have lower miss ratio than low set associative caches.

There are many possibilities for further investigation with PHA$E. For example, we can change the configuration of L3 cache from shared cache to private cache. This will be useful in the study of memory behavior in multi-processor systems. We can collect all cache statistics from each CPU separately and see the behavior of private L3 cache on each CPU. Another area we can investigate is replacement algorithms. There are many replacement algorithms besides LRU algorithm. For example, we can replace a block in the cache randomly instead of replace the least recently used one. This replacement algorithm is simpler to implement than LRU algorithm. As a result, the area of the design will be smaller and the speed of trace processing will be faster because the delay from shifting the TAGs will be eliminated. Then we can run some workloads and see the trade-off between performance and complexity of these two replacement algorithms. More

results from different workloads, different cache configurations, and cache behavior analysis can be found in [25, 26].

BIBLIOGRAPHY

1. A. R. Alameldeen and D. A. Wood, "Variability in Architectural Simulations of Multi-threaded Workloads," *Proceedings of the 9th Ann. International Symp. on HPCA*, Anaheim CA, February 2003.

2. W. G. Cochran, "Sampling Techniques," *John Willey & Sons*, third edition, 1977.

3. J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, "BACH: BYU Address Collection Hardware, the Collection of Complete Traces," *Proceedings of the 6th International Conferences on Modeling Techniques and Tools for Computer Performance Evaluation*, September 1992, pp. 128–137.

4. K. Grimsrud, J. Archibald, M. Ripley, K. Flanagan, and B. Nelson, "BACH: A Hardware Monitor for Tracing Microprocessor-based Systems," *Microprocessors and Microsystems*, Vol. 17, No. 8, Elsevier Science, Amsterdam, October 1993, pp. 443–458.

5. J. L Hennessy, D. A Patterson "Computer Architecture: A Quantitative Approach, Second Edition" *Morgan Kaufmann Publishers, Inc.*, 1996.

6. S. A. Herrod, "Using Complete Machine Simulation to Understand Computer System Behavior," *Ph.D. Thesis*, Stanford University, February 1998.

7. M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989.

8. M. A. Holiday, "Techniques for Cache and Memory Simulation Using Address Reference Traces," *International Journal in Computer Simulation*, 1991.

9. E. E. Johnson, "PDATS II: Improved Compression of Address traces," *In Proceedings of IEEE Internaltional Performance, Computing, and Communications Conference*, February 1999.

10. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and W. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, February 2002, pp. 50–58. (http://www.simics.com)

11. A. Nanda, K. Mak, K. Sugavanam, R. K. Sahoo, V. Soundararajan, and T. B. Smith, "MemorIES: A programmable, Real-Time Hardware Emulation Tool for Multiprocessor Server Design," *Proceedings of the 9th Int. Conf. on Arch. Support for Prog. Lang and Operating Systems*, Cambridge MA, November 2000, pp. 37–48.

12. K. D. Safford, "A Framework for Using the Pentium's Performance Monitoring Hardware," *MS Thesis*, University of Illinois at Urbana-Champaign, 1997.

13. Standard Performance Evaluation Corporation's Web Site, http://www.specbench.org.

14. N. C. Thornock and J. K. Flanagan, "A National Trace Collection and Distribution Resource," *Computer Architecture News*, Vol. 29, No. 3, June 2001.

15. R. A. Uhlig and T. N. Mudge, "Trace-driven Memory Simulation: A Survey," in *ACM Computing Surveys*, J29(2), 1997, pp. 128–170.

16. M. Watson and J. Flanagan, "Simulating L3 Caches in Real Time Using Hardware Accelerated Cache Simulation (HACS): a Case Study with SPECint 2000," *14^{th} Symposium on Computer Architecture and High Performance Computing*, October 2002.

17. H. Yoon, G. Park, K. Lee, T. Han, S. Kim, and S. Yang, "Reconfigurable Address Collector and Flying Cache Simulator," *Proceedings of High Performance Computing Asia '97*, Seoul Korea, April 1997, pp. 552–556.

18. S.-L. Lu and K. Lai, "Implementation of Hardware Cache Simulator (HW$im) - A Real Time Cache Simulator," *Proceedings of FPL 2003*, Portugal, September 2003.

19. Java 2 Platform Enterprise Edition (J2EE), http://java.sun.com/j2ee/.

20. Standard Performance Evaluation Corporation's Web Site, http://www.specbench.org/.

21. K. D. Safford, "A Framework for Using the Pentium's Performance Monitoring Hardware," *M.S. Thesis*, University of Illinois at Urbana-Champaign, 1997.

22. Kingsum Chow and Gim Deisher, "SPECjAppServer2002 Performance Tuning," *WebLogic Developer's Journal*, September, 2003.

23. Y. Yan, X. Wu, J. Schalkwyk, and R. Cole, "Development of CSLU LVCSR: The 1997 DARPA HUB4 Evaluation System," *In Proceedings DARPA '98 BNTUW*, 1998.

24. C. Lai, S. Lu, and Q. Zhao, "Performance Analysis of Speech Recognition Software," *5th Workshop on CAECW*, February 2002

25. N. Chalainanont et. al., "Real-time L3 Cache Simulations Using the Programmable Hardware-Assisted Cache Emulator (PHA$E)," *Proceedings of wwc-6*, Austin, TX, 2003

26. N. Chalainanont, E. Nurvitadhi, K. Chow, and S. L. Lu, "Characterization of L3 Cache Behavior of Java Application Server," *Seventh Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2004