ABSTRACT

Title of Thesis: PERFORMANCE CHARACTERISTICS OF AN INTELLIGENT MEMORY SYSTEM

Justin Stevenson Teller, Master of Science, 2004

Thesis directed by: Associate Professor Charles Silio
Electrical and Computer Engineering

The memory system is increasingly becoming a performance bottleneck. Several intelligent memory systems, such as the ActivePages, DIVA, and IRAM architectures, have been proposed to alleviate the processor-memory bottleneck. This thesis presents the Memory Arithmetic Unit and Interface (MAUI) architecture. The MAUI architecture combines ideas of the ActivePages, DIVA, and ULMT architectures into a new intelligent memory system. A simulator of the MAUI architecture was added to the SimpleScalar v4.0 toolset. Simulation results indicate that the MAUI architecture provides the largest application speedup when operating on datasets that are much too large to fit in the processor's cache and when integrated with systems using a high performance DRAM system and a low performance processor. By coupling a 2000 MHz processor with an 800 MHz DRDRAM DRAM system, the Stream benchmark, originally written by John D. McCalpin, completed 121% faster in simulations when optimized to use the MAUI architecture.

PERFORMANCE CHARACTERISTICS OF AN INTELLIGENT MEMORY SYSTEM

by

Justin Stevenson Teller

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2004

Advisory Committee:

Associate Professor Charles Silio, Chairman/Advisor
Associate Professor Bruce Jacob
Assistant Professor Donald Yeung

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1: Introduction

Processor performance has enjoyed enormous performance increases in recent years. Historically, processor performance has increased about fifty-eight percent annually since 1994. Unfortunately, the memory system's performance has not increased as quickly as the processor's performance. Dynamic Random Access Memory (DRAM) latency has only decreased about seven percent annually, and DRAM bandwidth has increased about fifteen percent annually. The performance gap between the memory system and the processor has become a performance bottleneck to total computer system performance. The memory-processor performance gap is increasing as time progresses, only making the performance bottleneck worse [9].

An intelligent memory system is one architectural feature which shows promise in overcoming the performance bottleneck associated with memory accesses. Any intelligent memory system builds computational ability into the memory system. The goal of intelligent memory systems is to improve the performance of memory-bound applications and operations by moving some of the computation closer to the data stored in memory. Intelligent memory systems fall into one of two categories: either they migrate computational power into the DRAM system, or they migrate DRAM into the main processor [2].

Several intelligent memory systems have already been proposed, and their performance characteristics explored. The Active Pages architecture [19] and the Data IntensiVe Architecture (DIVA) [8] represent two intelligent memory system architectures that take the former approach of migrating computational power

into the DRAM system. The Active Pages project in particular has been shown, through simulations, to improve performance of some applications by a factor of about 1000 times [19].

The Intelligent RAM (IRAM) architecture [25] represents an intelligent memory system architecture that takes the latter approach of migrating DRAM into the processor. The IRAM architecture integrates a simple processor with several banks of DRAM. One IRAM architecture that has shown promise is the Vector IRAM (VIRAM) architecture. The VIRAM architecture integrates a vector processor with DRAM onto a single chip. Simulation results indicate that the VIRAM architecture is significantly faster than conventional cache based machines on truly memory system limited benchmarks. For instance, simulations indicate that the the VIRAM architecture is able to compute the transitive closure of a directed graph in a dense representation more than twice as fast as an Intel P4 1.5GHz workstation [6].

Despite impressive simulation studies, none of these proposed intelligent memory system architectures have gained popular support for consumer computer systems. One reason may be that the integration of logic and DRAM onto a single silicon die and moving away from commodity DRAM has proven to be expensive. There has already been one intelligent memory system proposed that does not require the integration of logic and DRAM onto a single silicon die. The User-Level Memory Thread (ULMT) [26] architecture builds additional computational power into the memory controller, avoiding the merging of DRAM and processing logic onto a single die and allowing the use of a commodity DRAM system. However, the ULMT architecture is not explicitly controlled by the application running in the processor, and is used specifically to aid in prefetching.

Despite this inflexibility, the ULMT architecture has been shown, in simulations, to provide up to a 58% speedup for some applications.

This thesis presents a new intelligent memory system architecture named the Memory Arithmetic Unit and Interface (MAUI) architecture. The MAUI architecture combines traits from the Active Pages, DIVA, and ULMT architectures to create a new computational model. Like the Active Pages and DIVA architectures, the MAUI architecture migrates computational power into the memory system. Furthermore, the MAUI architecture is explicitly controlled by the application running in the host processor, much like the Active Pages and DIVA architectures. Like the ULMT architecture, but unlike the Active Pages and DIVA architectures, the MAUI architecture does not require logic and DRAM to be integrated onto a single silicon die. The MAUI architecture integrates additional computational power onto the same chip as the memory controller. The MAUI architecture is further split into two separate components: the Memory Arithmetic Unit (MAU) and the Memory Arithmetic Unit Interface (MAUI). The MAU performs the actual arithmetic performed by the MAUI architecture, while the MAUI coordinates the data flow through the MAUI architecture.

Because the MAU is located on the same chip as the memory controller, it has a higher bandwidth, lower latency connection to memory than the host processor. Because the MAU has a more efficient connection to memory than the processor, the MAUI architecture completes memory-bound operations more quickly than the processor could complete the same memory-bound operation. Additionally, because the MAUI architecture is a separate processing element from the main processor, further application speedup is possible by exploiting parallel execution using the MAUI hardware and the host processor.

For the purpose of testing the performance of the MAUI architecture, the SimpleScalar v4.0 simulator was modified to include a MAUI enhanced memory system. Then, three benchmarks to test the MAUI enhanced memory system were created. The first two benchmarks, *MAUI-one* and *MAUI-two*, are "artificial," in that they do not represent real-world applications are were designed only to determine under what circumstances the MAUI hardware performs well. Simulations of *MAUI-one* and *MAUI-two* have shown that the performance of the MAUI hardware increases as the memory system's performance increases, the problem size increases, and the processor speed decreases. Simulations of *MAUI-one* have shown that the MAUI hardware can perform a single vector operation up to 103% faster than the processor, and simulations of *MAUI-two* have shown that, by using the MAUI hardware and the host processor in parallel, applications can run about 80% faster than by using the processor alone.

The final benchmark, *Stream*, is a well accepted benchmark used to test total memory system performance. Originally written by John D. McCalpin [16], *Stream* performs four vector operations on three extremely large arrays. Performing three of *Stream*'s vector operations using the MAUI hardware resulted in a 121% speedup compared to the unoptimized version in simulations. Because this approach exploited both the fact that the MAUI performs vector operations faster than the processor, as well as some parallelism, *Stream* performed better than what was predicted from the *MAUI-one* and *MAUI-two* simulations.

The remainder of the thesis is divided into four chapters. Chapter 2 begins by outlining the background and motivation for intelligent memory system architectures and concludes with a summary of previous research done on intelligent memory system architectures. Chapter 3 introduces the new intelligent memory

system, the MAUI architecture, details the MAUI's architectural features, and concludes with a description of the simulation environment used to test the MAUI architecture's performance characteristics. Chapter 4 presents the simulation results of the MAUI architecture. Chapter 5 concludes the thesis by summarizing the conclusions and suggesting areas of further research.

Chapter 2: Background, Motivation, and Previous Work

Computer system performance is greatly increasing as time progresses. In general, Moore's law predicts that the performance of a computer system will double every eighteen months. Unfortunately, the performance of the memory system has not increased nearly as fast as the rest of the computer system. The performance gap between the memory system and the rest of the computer system has become a performance bottleneck to total computer system performance. The processor-memory performance gap is increasing as time progresses, only making the performance bottleneck worse.

Conventional cache-based computer systems use latency hiding techniques to alleviate some of the performance bottleneck caused by memory system performance. However, latency hiding techniques such as caching, out-of-order execution, and prefetching are becoming less effective at improving the performance of the memory system and are exposing the bandwidth limitations of the memory system. Intelligent memory systems seek to improve the performance of the memory system by merging computational power into the memory system. The remainder of the chapter covers the background and motivation for intelligent memory systems and concludes with a summary of some previous intelligent memory system research. The following sections provide the background and motivation for the Memory Arithmetic Unit and Interface (MAUI) architecture, a new intelligent memory system architecture. The MAUI architecture is introduced and detailed in Chapter 3.

2.1 Conventional Cache-Based Computer Systems

Modern computer systems can be partitioned into two pieces: a processing system to perform mathematical and logical operations, and a memory system to hold all the data and instructions involved with those operations. Most general purpose computers consist of a single processor and a memory system consisting of several levels. The lowest levels are cache memory, and one or more levels of cache are usually integrated on the same die as the main processor. The caches are built from semiconductor Static Random Access Memory (SRAM). The next level of memory is main memory, and it is usually constructed from several chips of semiconductor Dynamic Random Access Memory (DRAM). The highest level of the memory system is non-volatile. Non-volatile memory is usually built from magnetic disks. Semiconductor non-volatile memory, such as floating-gate or Flash memory, could be used as non-volatile memory for smaller data requirements [9].

In order for a processor to perform operations on data, the data must be first transferred from main memory into the processor. In modern operating systems, copying data from the memory system to the caches is a very time expensive operation. Because increases in processor speed out-pace the performance increases in memory systems, the relative time the processor waits on memory accesses is increasing. Figure 2.1 illustrates the need for a processor to wait for the data transfer to complete before continuing execution.

While the performance of both the processor and memory are increasing exponentially as time progresses, processor performance is increasing at a much faster rate than memory performance. The transistor density possible for logic chips, which include processors, increases by about thirty-five percent annually, while

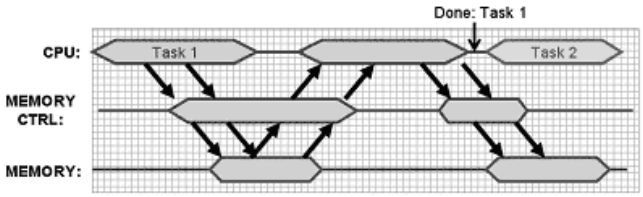

Figure 2.1: Timing diagram illustrating DRAM latency.

die area increases from ten to twenty percent annually. These factors combine to increase the transistor count possible on a single chip about fifty-five percent annually [9]. The increase in transistor count is one factor that accounts for the approximately fifty-eight percent increase in processor performance annually since 1994. SRAM performance follows the performance trend, which means that caches run approximately the same speed as processors.

While the density of semiconductor DRAM increases by between forty and sixty percent annually, the performance of DRAM is also increasing much more slowly. DRAM latency has decreased by only about one-third in ten years averaging just a seven percent decrease per year. Bandwidth per DRAM chip has improved only about twice as fast as DRAM latency [9]. Although DRAM density increases have been impressive, the performance improvements of the DRAM system hasn't followed the DRAM density increases. The performance gap between the processor and the memory system is only increasing as time progresses. The growing processor-memory performance gap is illustrated in Figure 2.2.

Read-latency is defined here as the shortest possible response time of a single DRAM chip. The shortest possible response time is the amount of time following a Column Address Strobe (CAS) read command before the DRAM chip responds. This definition for read-latency is extremely optimistic, as it does not take into

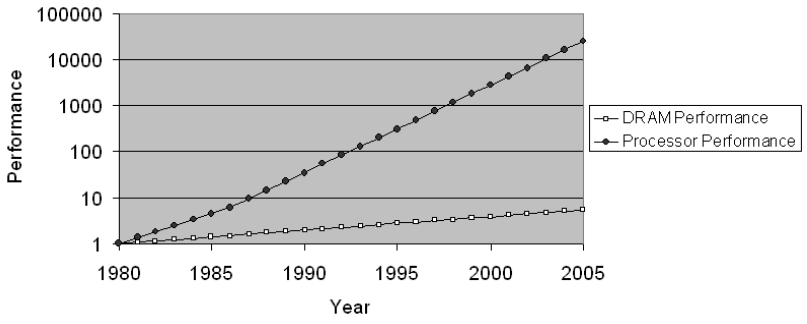

Figure 2.2: With 1980 performance as a baseline, the performance gap between memory and processor performance is plotted over time [9].

account the precharge and Row Address Strobe (RAS) commands which are typically required in order to read data out of the DRAM chip. Currently, the read-latency of a typical DRAM chip is approximately fifteen nanoseconds [17]. Assuming the optimistic read-latency of only fifteen nanoseconds, a processor running at 3.0GHz (e.g. Intel's Pentium 4 or AMD's Athlon) needs to wait at least forty-five cycles for data to return from a read operation on a single memory chip. In reality however, overheads introduced by the caches and memory controller force many computer systems to wait several hundred cycles or more for a single read operation to return data from memory. As shown in Figure 2.2, the performance gap between processor and DRAM speed is increasing at fifty percent annually [24].

Modern architectures employ three schemes in an effort to hide or reduce the apparent latency of reading and writing to memory. The first scheme is out-of-order execution. Out-of-order execution allows instructions which aren't dependent on one another to finish out of order. With out-of-order execution, an instruction blocked on a memory operation need not delay the completion of in-

structions which don't depend on the memory operation. Out-of-order execution also has other performance advantages. It allows for non-memory instructions with differing latencies to finish out of order and exploit the Instruction Level Parallelism (ILP) of the program.

The second scheme is caching. A cache is a small amount of very fast memory. If the required data are located in the cache, then the latency to access that data is much smaller than the latency to read it out of main memory. The computer system will load the cache with data which it believes the processor will need in the future. This is data which shows temporal or spatial locality. Data which expresses temporal locality are data which accessed once will probably be accessed again. To exploit temporal locality, data accessed once will be kept in the cache for as long as possible. Data which expresses spatial locality are data close to data already accessed that will probably be accessed as well. To exploit spatial locality, data are moved into the cache in blocks larger than what is initially required by the processor [9].

The final scheme is prefetching. Prefetching brings data from the main memory into the cache before it is required by the processor for computation. Transferring data into the cache before it is required hides the latency of the initial data access. There are a several popular and effective software and hardware prefetching techniques [26].

However, the number of cycles a processor is stalled due to memory bandwidth limitations increases as more aggressive latency tolerating schemes are employed [2]. Because the number of cycles a processor is stalled increases as more aggressive latency tolerating schemes are employed, the total memory system performance, both latency and bandwidth, are increasingly important to overall

system performance.

It is also possible to operate on data for which caching and other latency tolerance schemes are not effective. Data sets which have a low degree of locality or which are too large to fit in the cache defeat caching. Out-of-order execution does not greatly improve the throughput of dependent instructions, so it is possible for a long latency memory instruction to stall all other dependent instructions as well. Also, multimedia applications (e.g. video, picture, and sound encoding, decoding, and compression) have hard to predict data accesses and express only small amounts of locality. Therefore, multimedia applications are a class of applications for which latency hiding techniques do not work well [3].

2.2 Intelligent Memory Systems

An intelligent memory system is one architectural feature which shows promise in overcoming the performance bottleneck associated with memory accesses. Any intelligent memory system builds computational ability into the memory system. Intelligent memory systems take one of two directions: they migrate computational power into the DRAM system, or they migrate DRAM into the processor [2]. The former direction introduces more powerful primitives than simple reads and writes to be issued to the memory system. Migrating processing power into the DRAM allows for those operations which include a large number of memory accesses to be offloaded completely into the memory system, drastically lowering the number and frequency of memory accesses required by the processor. The second approach, migrating DRAM memory onto the processor, the processor experiences lower memory latency, increased memory bandwidth, and lower power consumption [24].

The following sections summarize the research efforts of some important memory system architectures. The first several sections describe Active Pages [14, 19, 20, 21, 22] and DIVA [8, 11], two architectures which migrate computational power into the DRAM system. The next section describes IRAM [6, 24, 25] and VIRAM [10], two architectures that take the second approach of migrating DRAM onto the same die as the processor. The final section covers other important intelligent memory system architectures including an image processing specific IRAM architecture [13, 12], a prefetching specific intelligent memory controller [26], an intelligent memory system which places the computational power into the memory controller [4], and several Processor In Memory (PIM) based multiprocessor computer systems [7, 28].

2.2.1 Active Pages

Introduced in 1998 at University of California Davis' department of Computer Science, Active Pages presents a new computational model. The Active Pages computational model allows the memory system to perform vector operations, such as add, multiply, find, insert, and delete, all within the memory system. The Active Pages architecture forces the program to partition applications between a processor and the intelligent memory system, leading to memory-centric and processor-centric operations.

While the computational model introduced by Active Pages could be implemented in a number of different ways, the Active Pages focuses on the integration of reconfigurable logic with DRAM. The physical implementation of Active Pages is the Reconfigurable Architecture RAM (RADram), shown in Figure 2.3. In the RADram implementation of Active Pages, the DRAM is broken into sub-arrays,

each of which has a dedicated reconfigurable functional unit. Each functional unit operates only on data in a single sub-array. The combination of a sub-array and a functional unit is an Active Page, capable of storing data and performing manipulations of that data. The entire memory system is partitioned into a large number of Active Pages.

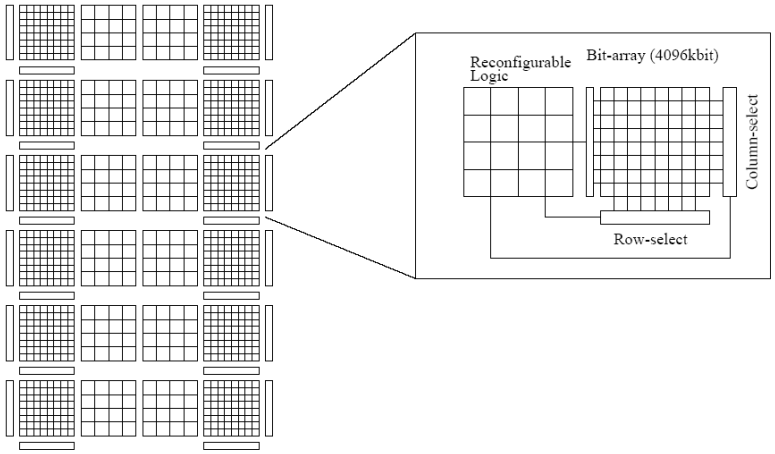

Figure 2.3: The RADram implementation of Active Pages [19].

In addition to the speedup associated with performing memory-centric operations solely in memory, the Active Pages architecture also introduces support for an enormous amount of parallelism. First, the processor is free to perform operations which have no dependence on the memory-centric operations. Second, each Active Page works independently of every other Active Page. Because each Active Page is an independent processing element, the Active Pages architecture can exploit massive amounts of parallelism, and simple operations can be performed on arbitrarily large data in a very small amount of time [19]

Active Pages used SimpleScalar v2.0 [1] as the base simulation environment. SimpleScalar was extended by replacing the conventional memory hierarchy with

a RADram enabled memory system. In simulations, Active Pages with reconfigurable logic is reported to show a 1000x speedup compared to a conventional cache-based system [19].

While the RADram implementation of Active Pages has been shown to be very promising, there is one major problem. Reconfigurable logic is expected to occupy fifty percent of the available chip area, making efficient memory density impossible. To alleviate the memory density problem, a different implementation of Active Pages was proposed. In the new implementation of Active Pages, the reconfigurable logic is replaced by a Very Long Instruction Word, (VLIW) processor. Active Pages using a VLIW processor is reported to require only thirty-one percent of the chip area to be occupied with computational logic. In simulations, the VLIW Active Pages demonstrates a speedup comparable to the reconfigurable logic implementation of Active Pages. Furthermore, VLIW Active Pages shows that instruction-level parallelism, and not hardware specialization, is what drives the performance gains in the Active Pages intelligent memory system [21]. This lesson can be applied to many intelligent memory system architectures.

A widely accepted assumption is that most computer systems are multiprogrammed. One can expect that in such a system some applications would utilize Active Pages memory, and others would not. Therefore, in such an environment, the computational resources available in the Active Pages memory system are not fully utilized. By sharing the computational logic between several logically distinct pages in memory, the area required by computational logic in the DRAM chip can be reduced from thirty-one percent to twelve percent while retaining virtually identical performance on simulated multi-threaded workloads. Sharing the computational logic between Active Pages greatly increases the chip area which

is devoted to DRAM. By devoting more chip area to DRAM, the memory density of the chip is increased, making the Active Pages architecture more commercially viable than was previously possible [22].

To further facilitate the adoption of an Active Pages architecture, an operating system, ActiveOS, has been introduced which supports and takes advantage of an Active Pages memory system. ActiveOS is aware of the intelligent memory system and schedules paging and inter-chip communication to achieve high performance for those applications who use Active Pages. It is reported that individual applications still experience up to a 1000x speed up, and total multiprogrammed workloads experience between a twenty and sixty percent speedup [20].

The cache coherence problem [1] arises in any intelligent memory system which merges extra computational ability into the memory system, because the memory system is another processing element which can alter memory's state. The Active Pages project has explored two approaches to enforce cache coherence in intelligent memory systems. The first approach is software driven. The processor's cache is explicitly flushed whenever the intelligent memory system may change some location in memory, preventing the retention of stale data.

The second cache coherence approach uses a hardware enforced cache coherence protocol, which is similar to the protocol used in conventional Symmetric Multiprocessor Systems (SMP). In the hardware enforced cache coherence pro-

[1]The cache coherence problem is when incorrect data can be read out of the cache. It arises in any system which has more than one processing element and employs caching. Incorrect data could be read out of the cache by one processing element when a different processing element changes that data in the memory and that change is not reflected in the cache of the first processing element [5].

tocol, the notion of owning pieces of memory is introduced. When a processing element owns a piece of memory, it is guaranteed not to be cached by any other processing element, allowing for safe modification of the data residing within that piece of memory. In the hardware enforced cache coherence protocol, each Active Page is a processing element with status in the cache coherence protocol equal to that of the host processor.

With small data sizes, explicit flushing and hardware coherence yield similar performance in simulations, but hardware coherence requires less bandwidth. Therefore, hardware coherence is the better method of cache coherence, because as the number of threads in a multiprogrammed environment increases, the bandwidth needs of a cache coherence system increase [14]. Active Pages has been found to be an extremely promising model for an intelligent memory system.

2.2.2 DIVA

Similar to the Active Pages architecture, the Data IntensiVe Architecture (DIVA) integrates processing elements into the memory chips. The DIVA architecture differs from Active Pages in the way the processing elements are integrated into the memory system. First, DIVA incorporates one complex processing element per memory chip while Active Pages integrates a number of simpler processing elements per chip. Also, DIVA allows communication between the processing elements in the memory system while Active Pages allows no such data sharing within the memory system by forcing all communication to pass through the host processor.

The DIVA architecture seeks to increase the memory bandwidth available to the processor by performing selected computations within the memory system.

DIVA seems unique in its explicit support of irregular applications, including sparse-matrix multiply and pointer chasing. The author's simulations have shown that such types of irregular applications show between 1.6 and 30 times speedup when run on a DIVA architecture [8].

In the DIVA architecture, the memory system consists of a number of Processor In Memory (PIM) modules. Each PIM module consists of a single processing element and an array of DRAM. Approximately forty percent of the die area is devoted to computation while sixty percent of the die area is DRAM memory. On each PIM chip, the processing functional unit is designed to support wide vector operations. The processor is able to process up to 256 bits in a single cycle to allow similar operations to be performed on every element in a single vector. Vector computation also maximizes the processor-memory bandwidth in a single PIM chip. The remainder of the computation logic on the chip is devoted to controlling communication [8].

The PIM array is tied together with a dedicated PIM-to-PIM network. The close coupling between PIM array elements allow for high bandwidth and low latency data movement between PIM chips without outside arbitration. As relatively few PIM chips are tied together on a single network, the PIM-to-PIM interconnect remains simple while retaining high performance [11].

The PIM array is controlled by one or more host processors. The host processor utilizes the PIM array as its "dumb" memory system and also initiates any computation which is to occur in the memory system. The approach of integrating processing elements with a conventional memory system while retaining a host processor makes the DIVA architecture closely related to the Active Pages project. One difference is that the Active Pages architecture integrates a

larger number of more limited processing elements in the memory system, while there are fewer, more complex processing elements in DIVA. One more key difference is Active Pages severely limits sharing between the processing elements in the memory system, whereas DIVA creates a dedicated network to support the communication between the PIM chips. By limiting sharing, Active Pages is able to simplify the processing structure of the memory system while achieving greater amounts of parallelism. However, DIVA's dedicated PIM network greatly increases the efficiency of largely inter-dependent irregular applications. The DIVA architecture is most significant in its explicit support of irregular applications, which don't traditionally perform well on intelligent memory systems [8].

2.2.3 IRAM and VIRAM

In contrast to the Active Pages architecture and the DIVA architecture, Intelligent RAM (IRAM) takes the second approach in intelligent memory design by migrating DRAM into the processor. The IRAM architecture, introduced at Berkeley, results in a single-chip computer consisting of a processor and several banks of DRAM. The migration of DRAM onto the processor die significantly increases the memory bandwidth and lowers the memory latency experienced by the processor. For instance, simulations indicate that the the VIRAM architecture is able to compute the transitive closure of a directed graph in a dense representation more than twice as fast as an Intel P4 1.5GHz workstation [6]. Furthermore, the power dissipation of the computer system as a whole could be decreased, as there is no need to drive an external memory bus [24].

IRAM has one major limitation when compared to other intelligent memory

system architectures. As IRAM is manufactured with a fixed amount of memory, there is no opportunity to tune the performance and price of the system by changing the amount of memory. Despite this limitation, the single chip design and energy efficiency make IRAM an interesting candidate for embedded devices and personal mobile multimedia devices [25].

VIRAM is an IRAM architecture based around a vector processor. A vector processor is able to perform identical operations on all elements of a vector, or array, in parallel. As vector computations are extremely valuable to many multimedia applications, the VIRAM system is mostly targeted for streaming or real-time multimedia applications [10]. A block level schematic of the VIRAM single-chip computer is shown in Figure 2.4.

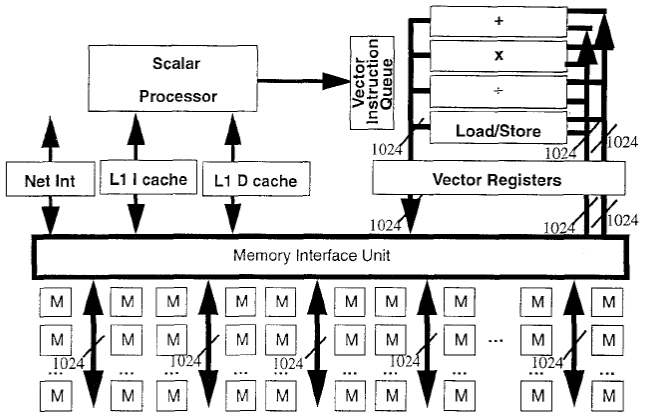

Figure 2.4: The VIRAM architecture [23].

In comparison with commercial cache-based machines, the VIRAM architecture is significantly faster for applications whose performance bottleneck is caused by memory system performance. The advantage is because the VIRAM architecture enjoys a significantly higher bandwidth to memory than most cache-based machines. The peak memory bandwidth of a VIRAM architecture is 6.4 GB/s,

which is 5–10 times higher than most cache-based machines [6].

Both IRAM and VIRAM have been proposed as possible intelligent memory system architectures for the support of multimedia and embedded applications. In addition, both have also been shown to perform extremely well in comparison to commercial computer systems for applications truly limited by memory system performance. IRAM and VIRAM are two very promising models for an intelligent memory system.

2.2.4 Other Important Architectures

There are several other proposed architectures which take advantage of intelligent memory systems which are more application specific than Active Pages, DIVA, IRAM, and VIRAM. One architecture is the Intelligent RAM architecture introduced at the University of Illinois (U. of I.) at Urbana-Champaign. While the architecture proposed at U. of I. is similar to Berkeley's IRAM project, the U. of I. Intelligent RAM architecture specifically supports complex image processing applications. This Intelligent RAM architecture supports rasterization, image analysis, and pattern recognition applications [13, 12].

Another intelligent memory architecture proposed by U. of I. at Urbana-Champaign in conjunction with the Michigan State University supports prefetching. In this architecture, a User-Level Memory Thread (ULMT), separate from the application running on the main processor, runs on a general-purpose processor in main memory. In addition to being prefetching specific, the ULMT intelligent memory system differs from Active Pages, DIVA, IRAM, and VIRAM in that the ULMT architecture does not require the integration of processing logic and DRAM onto the same die. The ULMT performs correlation prefetching in

support of the application running on the main processor. Correlation prefetching uses past sequences of memory accesses to predict future memory accesses and prefetch those addresses into the processor's cache [27].

The processor performing correlation prefetching resides within the memory system on the memory controller or "north bridge" chip [2] By integrating the prefetching engine within the North Bridge chip, a large amount of flexibility in the prefetching algorithm with minimal changes to both the commodity DRAM chips and the processor is possible. With a general-purpose processor residing in the memory controller, the sole change to the main processor is the top level of cache must be able to accept prefetches coming in from the ULMT. The DRAM chips remain completely unchanged. Results of several simulations indicate that the ULMT architecture can result in up to a fifty-three percent speedup on single threaded applications [26].

The ULMT architecture is important because of how it is different from other intelligent memory systems. The ULMT architecture precludes the need to integrate DRAM and processing logic onto a single chip and thus greatly reduces the cost of such a system. Additionally, the gains reported by using the ULMT architecture do not depend on the programmer having in-depth knowledge of the intelligent memory system [26].

Similar to the ULMT architecture, the Imagine architecture, currently developed at Stanford, creates an intelligent memory system without merging DRAM and processing logic onto a single chip. Imagine is a stream architecture designed

[2]In many consumer computer system motherboards, including those designed by Intel, the North Bridge chip, sometimes referred to as the chip-set, coordinates all data in and out of the processor. The North Bridge is connected directly to the processor by way of the Front Side Bus (FSB), and includes a memory controller and possibly graphics processing unit.

to support applications exhibiting high data parallelism and producer-consumer locality. Imagine includes several arithmetic clusters in the memory system, in a separate chip from the memory chips. Such separation allows for read and write transactions initiated by the host processor to be effectively overlapped with memory system computation [4]. A block level schematic of the Imagine architecture is shown in Figure 2.5.

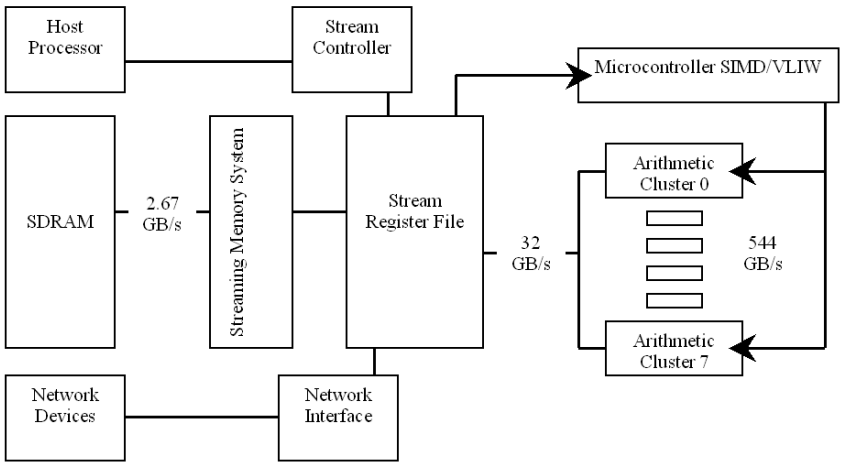

Figure 2.5: The Imagine architecture [4].

When compared to Berkeley's VIRAM architecture, Imagine performs very well. Studies indicate that VIRAM outperforms Imagine in those applications that have a low ratio of operations per memory access. However, Imagine is reported to significantly outperform VIRAM for those applications that have a much higher ratio of operations to memory accesses [4].

At least two purely PIM multiprocessor architectures also been proposed [7, 28]. Both architectures integrate a large number of limited processors into a single machine, creating support for an extremely large degree of parallelism. Each PIM in the parallel machine has very fast and efficient access to a limited amount of

memory. Massively parallel PIM architectures promise significant performance increases on parallel applications when compared to conventional multiprocessors.

2.2.5 General Intelligent Memory System Limitations

While every intelligent memory system architecture presented in this chapter promises a speedup for some class of applications, none of them have been generally accepted as a consumer computer architecture. There seems to be a two nearly universal reasons for this lack of acceptance. First, although merging logic and DRAM onto a single chip looks promising, merged logic technology has proven to be expensive. Second, all the intelligent memory systems summarized in Chapter 2 seek to move away from traditional commodity DRAM systems to some new DRAM technology or interface. However, the DRAM manufacturer community is extremely large and well funded. For instance, Kingston technology, just one DRAM manufacturer, has sales of over $1.8 billion in 2003 [29]. A more successful approach may be to leverage the multi-billion dollar DRAM fabrication industry by creating an intelligent memory system that uses commodity DRAM's.

In Chapter 3, yet another intelligent memory system architecture called the Memory Arithmetic Unit and Interface (MAUI) architecture will be presented. This new intelligent memory system architecture takes inspiration from the Active Pages, DIVA, and ULMT architectures by implementing a general purpose intelligent memory system architecture which uses a host processor to explicitly control computation within the memory system without integrating DRAM and processing logic onto a single chip. The MAUI architecture allows for the use of any commodity DRAM system and interface and makes only minimal changes to

the processor. When compared to conventional computer systems, major modification in the MAUI architecture is reserved for the memory controller or North Bridge chip.

Chapter 3: The Memory Arithmetic Unit and Interface (MAUI)

The memory system is increasingly becoming a performance bottleneck for total system performance. Intelligent memory systems increase the performance of memory bound computation buy adding computational power to the memory system. While there have been intelligent memory system architectures introduced that promise significant performance increases when compared to traditional cache-based computer systems, none of these architectures has gained popular support and acceptance for consumer computer systems. This chapter presents a new intelligent memory system named the Memory Arithmetic Unit and Interface (MAUI) architecture. The MAUI architecture combines architectural features of the Active Pages, DIVA, and ULMT architectures into a new intelligent memory system.

Like the Active Pages and DIVA architectures, the MAUI memory operations are explicitly invoked by the host processor, meaning that the processor's Instruction Set Architecture (ISA) is augmented to include MAUI instructions. The MAUI architecture performs vector operations on arbitrary size vectors. These computations include addition and multiplication of two vectors, scaling of a single vector, and data movement. Other computations that may prove valuable, but are not explored as design alternatives for the MAUI in the thesis, are other memory bound computations such as pointer chasing, searching, and sorting. By providing the host processor with explicit control of specialized memory operations, the MAUI architecture resembles both the Active Pages and DIVA architectures.

Whereas the Active Pages and DIVA architectures place computational power within the DRAM chips, the MAUI architecture integrates computational power onto the same chip as the memory controller. By placing the computational power of the intelligent memory system here, the MAUI architecture resembles the ULMT architecture. The placement of computational power within the memory controller decreases latency and increase bandwidth to memory when compared to the host processor. By avoiding the integration of processing logic and DRAM onto a single chip, the MAUI architecture is made less expensive than the Active Pages, DIVA, and IRAM architectures by using current processing technologies and conventional consumer DRAM chips. Explicit computational support of the application running on the host processor without integrating DRAM and processing logic onto a single chip combines the advantages of several intelligent memory systems to create the MAUI architecture.

The remainder of the chapter describes the MAUI architecture in detail. Section 3.1 describes the MAUI software interface. Section 3.2 describes the MAUI hardware, including the logical partitioning of the of the MAUI architecture's computational power. After the description of the MAUI architecture, Section 3.3 explores several performance drawbacks and discusses solutions to those drawbacks. To test the performance of the MAUI architecture, a MAUI augmented memory system has been added to the SimpleScalar v4.0 simulation environment. Section 3.4 discusses the details of the augmented version of SimpleScalar v4.0 used to test the performance of the MAUI architecture. Chapter 4 presents the simulated performance results of the MAUI architecture and Chapter 5 concludes the thesis.

3.1 MAUI Software Interface

Conventional memory systems support only two commands from the processor, read and write [1]. The MAUI architecture introduces a number of new memory system commands, or MAUI commands. The MAUI augmented memory system supports Single Instruction, Multiple Data (SIMD) type vector operations. SIMD operations perform the same operation on every element in a vector. An example illustrating a vector addition is shown in Figure 3.1. In Figure 3.1, notice that each addition operation is independent, allowing them all to be executed in parallel.

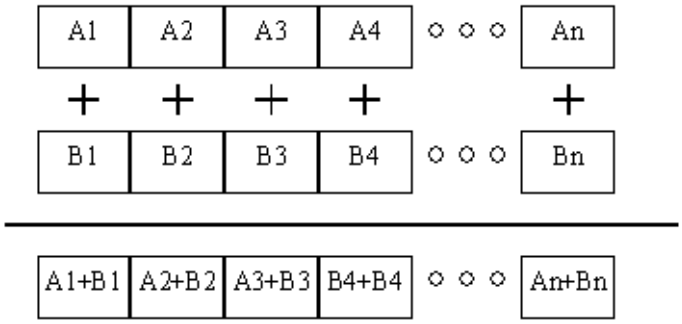

Figure 3.1: An example illustrating a SIMD vector addition operation.

The MAUI commands can be broken into two groups: setup commands and execution commands. Setup commands specify the size, source addresses, and destination addresses for the subsequent execution commands. Execution commands start the memory system computation. The MAUI architecture supports several integer computations, including addition and multiplication of two vectors

[1]Even prefetching requests represent a special category of read commands

and the scaling of a single vector. The MAUI commands are listed and explained in Table 3.1.

		Type	
Name	**Description**	**Setup**	**Execution**
maui-LD-a(x)	Loads the maui src. A address with *x*	X	
maui-LD-a&b(x,y)	Loads the maui src. A with address *x*, and src. B with *y*	X	
maui-LD-b(x)	Loads the maui src. B with address *x*	X	
maui-LD-c(x)	Loads the maui dest. C with x	X	
maui-LD-size(x)	Loads the maui block size with x	X	
maui-ADD-scale(x)	Adds x to every element in the src. vector A and stores the result in dest. vector C		X
maui-ADD( )	Vector addition, dest. C = src. A + B		X
maui-MUL( )	Vector multiplication, dest C = src. A * B		X
maui-MUL-scale(x)	Multiplies x to every element in the src. vector A, and stores in dest. vector C		X

Table 3.1: The MAUI instructions.

For instance, implementing a block copy using MAUI commands requires the use of four commands from Table 3.1. The first three commands are *maui-LD-size*, *maui-LD-a*, and *maui-LD-c*, which setup the MAUI with the correct source and destination addresses and block size which will be copied. The command

maui-ADD-scale is used to copy data by setting the scaling value to zero. The execution command to begin copying data is *maui-ADD-scalar*. Figure 3.2 shows the pseudo-C code for a block copy function named *bcopy* implemented with the four previously mentioned MAUI commands.

```
/* Function to copy a block of n bytes from src to dst */
void bcopy(const void *src, void *dst, int n) {
    maui-LD-size(n);
    maui-LD-a(src);
    maui-LD-c(dst);
    maui-ADD-scale(0);
}
```

Figure 3.2: A MAUI implementation of bcopy.

Although MAUI operations can take a significant amount of time to complete, and the latency of the MAUI operation is generally not known at issue time, the MAUI hardware allows the program to assume that the MAUI operation finishes instantly. The MAUI architecture ensures that any subsequent memory accesses, whether they are traditional memory system commands or other MAUI commands, will neither read stale data nor overwrite MAUI operands before the MAUI architecture has a chance to use them. The MAUI architecture does allow independent memory accesses to proceed and complete before the MAUI operation has completed. The logical ordering of memory accesses and MAUI operations is maintained automatically by the MAUI architecture.

3.2 MAUI Hardware

This section describes a hardware implementation to support the software interface described in Section 3.1. The MAUI architecture builds an intelligent memory system with only minor modifications to the processor. The MAUI architecture also leaves the DRAM system completely unchanged, so the MAUI enhanced memory system can use any consumer DRAM system. Major modification only occurs at the memory controller. By placing the MAUI architecture in the same chip as the memory controller, the MAUI computational engine enjoys a lower latency and higher bandwidth to memory than the processor. The MAUI architecture is split into two components, the Memory Arithmetic Unit (MAU) and the Memory Arithmetic Unit Interface (MAUI). The MAU performs all data computations while the MAUI controls the data flow, computes addresses, generates memory read and write requests, and enforces the logical ordering of memory accesses. The MAUI also includes a cache and registers to hold the source and result data for the computations.

The remainder of Section 3.2 is broken into three subsections. Subsection 3.2.1 describes the location of the MAU and MAUI. Subsection 3.2.2 then continues with a more detailed description of the MAU, and subsection 3.2.3 completes the section with a detailed description of the MAUI.

3.2.1 MAU and MAUI location

The MAU and MAUI are located on the same silicon die as the memory controller. Placing the MAU computational power inside the memory controller has two advantages. The first advantage is that integrating extra computation into the memory controller rides the technology trend of increased chip inte-

gration. Some consumer computer systems are already equipped with powerful graphics engines located within the memory system, such as NVIDIA's nForce, which integrates a powerful processor within the North Bridge Chip [18]. As process technology improves, the processing capability possible for a North Bridge Chip will increase. The MAUI architecture takes advantage of the possibility of additional processing power in the memory controller to improve total system performance. By riding the trend of integrating more processing power into the North Bridge chip, the MAUI architecture avoids the integration of processing logic and DRAM onto a single chip.

The second advantage is that the MAU and MAUI enjoy more efficient data transfer to and from the DRAM chips than the host processor. Data movement between the MAUI architecture and main memory does not stress the Front Side Bus (FSB), the connection between the processor and the rest of the system. By avoiding communication across the FSB, the the MAUI architecture has a lower latency and higher bandwidth connection to main memory than the does main processor. Because the MAUI has a higher bandwidth and lower latency to memory than the processor, it means that memory-bound computations can be performed more quickly within the MAUI architecture than they can be on the host processor.

3.2.2 The MAU

The Active Pages project demonstrated that the performance gain in using intelligent memory system architectures is mostly due to Instruction Level Parallelism (ILP) [21]. To exploit available ILP, the MAUI architecture performs vector computations on vectors as wide as a cache-line. With the SimpleScalar

architecture, the MAU supports two thirty-two byte vector operands. That means that the MAU performs eight integer arithmetic operations in parallel. Future possibilities for operations the MAU will support include searches, scatter-gather operations, pointer chasing, or other memory access bound operations which express significant ILP.

As the MAU is located on the same chip as the memory controller, it is limited to the same process technology, clock cycle, and power requirements as the memory controller. Fortunately, this limitation is mitigated by the fact that the MAU has a more efficient connection to main memory than the host processor and the SIMD nature of the vector operations it supports allow for significant exploitation of ILP.

3.2.3 The MAUI

The MAUI controls memory computations and acts as the intelligent memory system's interface to the rest of the computer system. It is the heart of the MAUI intelligent memory system computational model. The MAUI coordinates its caches, includes dedicated registers to hold the source and destination addresses, block size, and other run time information, performs address computation, and issues read and write requests to the DRAM system. The MAUI is also responsible for supplying the MAU with vector operands from memory. Lastly, the MAUI is responsible for ensuring the the logical ordering of traditional memory accesses and MAUI operations. While enforcing logical ordering the MAUI also allows non-MAUI memory operations to "leap-frog" long latency MAUI instructions and complete before the MAUI instructions are finished whenever possible. The block level schematic of the MAUI architecture is shown in

Figure 3.3. Notice that all of the data flow in the MAUI architecture passes through the MAUI.

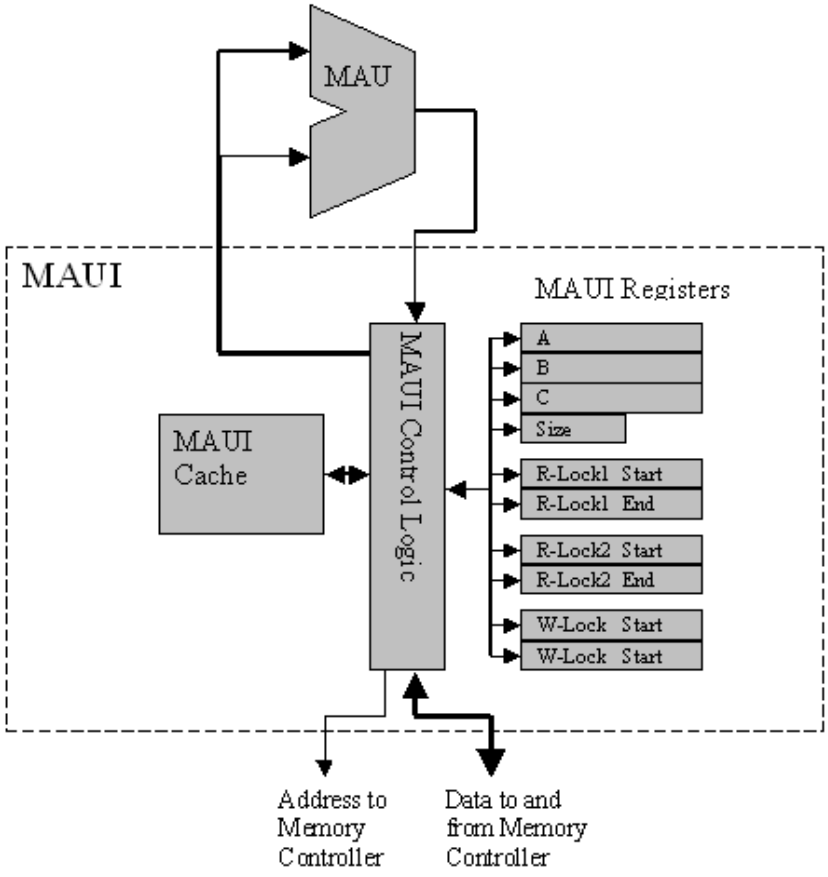

Figure 3.3: The block diagram of the MAUI architecture.

As shown in Table 3.1, MAUI commands are divided into setup and execution commands. The setup commands are used to load the source, destination, and size registers within the MAUI. The source registers shown in Figure 3.3 are registers A and B. These registers hold the beginning address of the source vectors. That means the source vectors occupy the memory ranges of A to $A+size-1$ and from B to $B+size-1$. The beginning address for the destination vector is

held in the register C, meaning that the destination vector occupies the memory range from C to $C+size-1$. The MAUI needs to be setup before any execution command is issued.

Once the MAUI is setup with valid source and destination vectors, the processor may issue an execution memory command. When the MAUI receives an execution memory command, it begins the execution of that command. Generally, the MAUI begins the the execution of the command by issuing read requests to main memory. When the data comes back from memory, it is stored in the MAUI cache until there are enough operands to perform some arithmetic in the MAUI cache. Once the required operands have been fetched from memory they are transferred to the MAU, which performs the actual arithmetic. Then, the result from the MAU's operation is sent back to memory with a write request to main memory. As an example of how the MAUI coordinates the data flow during the execution of a MAUI command, Figure 3.4 graphically details the execution of a *maui-ADD( )* command and how the data flows through the MAUI architecture.

To maximize the performance of the MAUI augmented memory system, non-MAUI memory operations are permitted to reorder with MAUI memory operations. However, the reordering cannot violate the logical ordering of memory operations and reorder dependent memory operations. To that end, one very important responsibility of the MAUI is to maintain the logical ordering of memory commands while allowing subsequent, independent memory operations to complete without waiting for the completion of the MAUI operation.

To maintain the logical ordering or traditional memory accesses and MAUI operations, the MAUI introduces the concept of locking memory. The MAUI

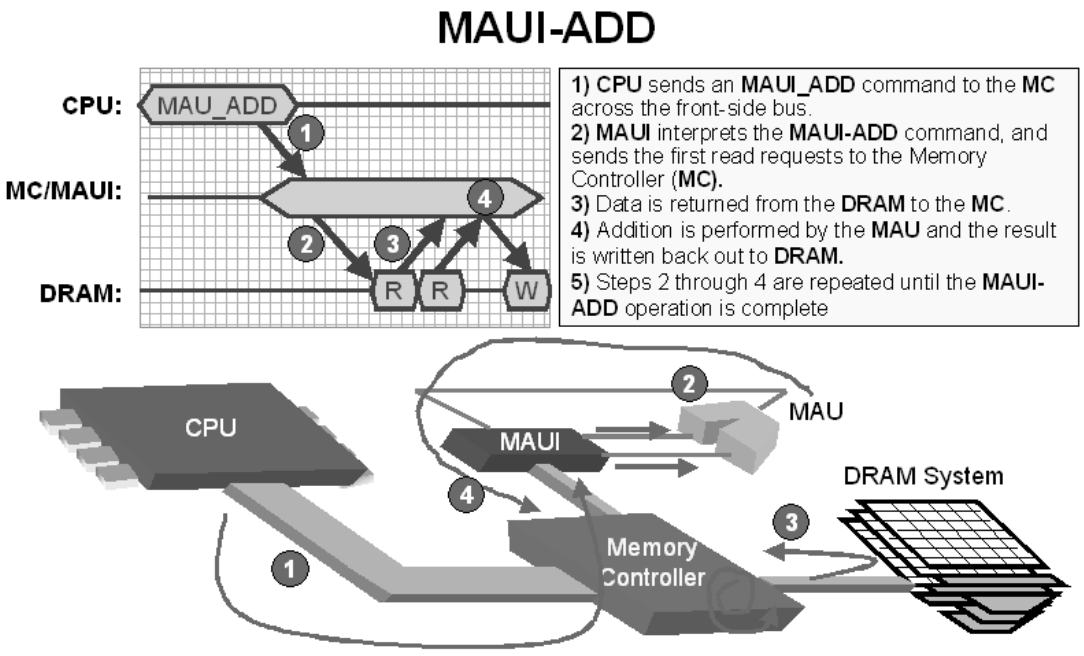

Figure 3.4: Illustration of the MAUI add operation. The MAUI has already been setup with the vector size and the source and destination addresses.

maintains two types of locks, *Read* and *Write* locks. The registers holding the values of the read and write locks are shown in Figure 3.3, and are called *R-Lock1 start* and *end*, *R-Lock2 start* and *end*, and *W-Lock start* and *end*. The addresses falling between the *R-Lock start* and *end* registers are *Read* locked, and those addresses falling between the *W-Lock start* and *end* registers are *Write* locked. A *Read* lock is placed on MAUI source addresses, or those memory locations that the MAUI needs to read. A *Write* lock is placed on MAUI destination addresses, or those memory locations that the MAUI needs to write to. A *Read* lock prevents later memory operations from modifying the data, but allows the data to be read by the host processor. A *Write* lock prevents later memory operations from modifying or reading the data. So, the *Read* lock prevents the

processor from modifying data that the MAUI hardware has not read yet, and the *Write* lock prevents the processor from reading stale data that the MAUI hardware has not over-written yet.

To enforce correctness, the MAUI stalls those memory commands which violate either the read or write locks. The MAUI rechecks stalled memory commands to see if they can be executed each time the MAUI completes any operation. When the MAUI is idle, memory commands are never artificially stalled. Because the MAUI must be able to stall memory commands that are not MAUI commands, the MAUI observes every command that enters the memory controller. Figure 3.5 illustrates how the MAUI read and write locks stall memory accesses.

Section 3.3 discusses host processor starvation and cache coherence as possible drawbacks to the MAUI architecture. These drawbacks can cause performance problems in using the MAUI architecture. Some solutions to the performance problems caused by these possible drawbacks are also discussed. Section 3.3 concludes with a discussion of security issues inherent with the MAUI architecture and Operating System (OS) support of the MAUI architecture as a solution to these issues.

3.3 Possible Drawbacks to the MAUI Architecture

One possible performance pitfall for the MAUI augmented memory system would be starvation of the processor going to memory. Starvation is defined here as the inability to either read or write to main memory. Because the MAUI is streaming data both into and out of the main memory, it would be possible for the MAUI's read and write requests to saturate the memory system and block

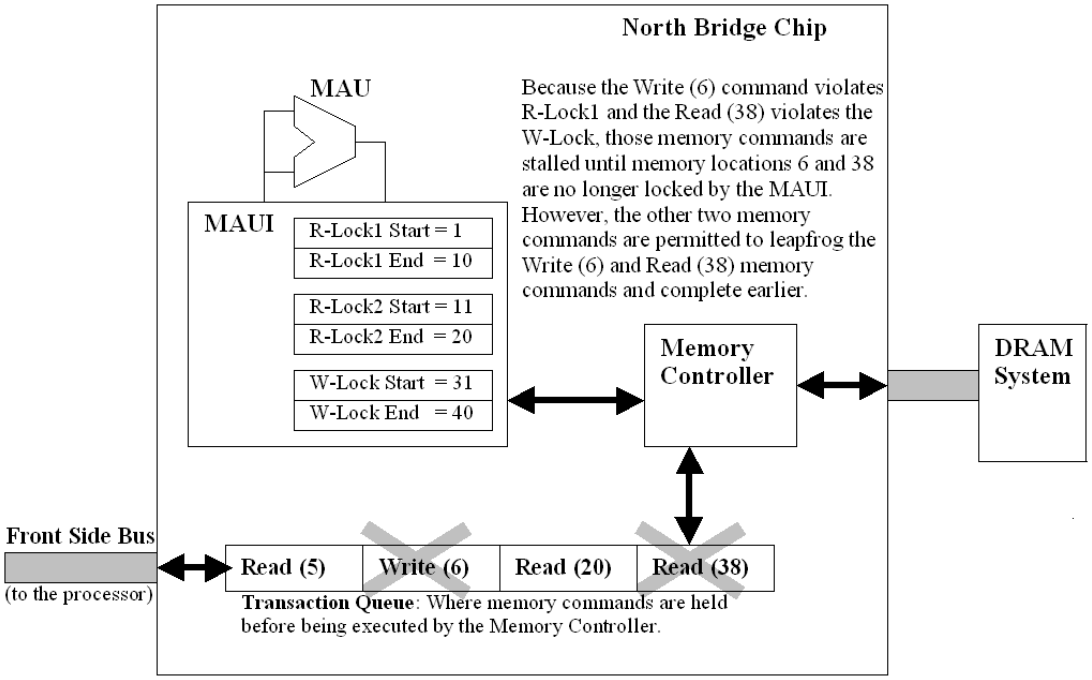

Figure 3.5: MAUI memory locks: illustrating how, although several memory commands are stalled because of the MAUI memory locks, other memory commands are permitted to complete.

any memory commands from the processor. The MAUI architecture employs two methods to help alleviate the processor starvation problem. The first method creates a priority scheme for memory requests. The priority scheme is that MAUI requests to memory have a lower priority than the host processor's memory commands. That way, as long as the processor's request did not violate any read or write locks, the processor's request to memory would bypass the MAUI's requests, helping to prevent starvation for the host processor.

However, the priority scheme does not completely eliminate the possibility of starvation. The MAUI could burst a large number of requests to memory at once,

saturating the memory system for an extended period of time. A burst of memory accesses would even stall higher priority processor memory commands which occur after the memory request burst. The second starvation prevention method solves the starvation problem by limiting the number of outstanding MAUI memory requests. An outstanding MAUI memory request is any read request from the MAUI for which it hasn't issued a corresponding write request. Therefore, the MAUI will only burst a small number of requests to memory, and won't block future, independent memory requests from the host processor. The MAUI restricts the number of outstanding memory accesses to four outstanding read requests: two from source A and two from source B. For those MAUI operations with only a single source, all possible outstanding memory accesses are allocated to source A.

Limiting the number of outstanding MAUI memory requests has the additional benefit of reducing the complexity of the MAUI architecture. Because the memory controller and DRAM system are free to reorder memory requests, the MAUI is required to keep track of all outstanding MAUI memory requests. Also, the MAUI will need to buffer up to one-half of the data for all outstanding memory requests. Looking at the *MAUI-ADD* operation as an example, half of the outstanding MAUI memory read requests are generated from source register A, and half of the outstanding MAUI memory requests are generated from source register B. From here, these data sets are referred to source A and source B. Because the memory requests could be reordered into any order, the data from all the read requests for source A could return from the DRAM system before any of the data from source B arrives. In this case, all of the data from the outstanding memory requests from source A needs to be buffered until the corresponding data

from B arrives from the memory system. If the MAUI were permitted to burst an unlimited number of requests to memory, the MAUI would need to buffer at least half of those requests. Unlimited buffering is impossible, of course, but unlimited outstanding MAUI memory requests, in the least, makes it very complex to keep track of the memory requests. Limiting the number of outstanding MAUI memory requests creates a less complex MAUI architecture.

By operating on data within the memory system, we run the risk of changing data that is stored in the processor's cache, introducing the cache coherence problem that is present in any system with more than one processing element. The MAUI architecture takes cues from the Active Pages project in solving the cache coherence problem. The Active Pages project showed that using software driven cache coherence results in similar performance to hardware driven cache coherence [14]. Because software driven cache coherence results in a more simple hardware implementation, hardware cache coherence is not explored for the MAUI architecture.

The software approach to cache coherence used for MAUI commands has two rules. First, the program must check addresses that fall within the source vector(s). The caches must write out any "dirty" data in the cache that falls within the source vector(s), but these addresses don't need to be invalidated. "Dirty" data are any data that the host processor has written to the cache but not to main memory. Writing "dirty" cache lines to memory ensures that main memory has the most recent copy of the data. However, the data are allowed to stay within the cache, allowing the processor to read data that falls within the source vector(s). The second rule applies to the destination addresses. The caches must be invalidated for all addresses that fall within the destination vector.

Invalidating the addresses that fall within the destination vector ensures that the processor would need to issue a request to main memory to read or modify data falling within the destination vector. Invalidating the correct cache lines allows the MAUI to prevent the host processor from either reading stale data or modifying data in the destination vector before the MAUI operation completes.

Another issue is introduced when examining the nature of virtual memory. Most, if not all, consumer computer systems implement some form of virtual memory. When a virtual address is translated to a physical address, it is clear that a contiguous segment of virtual memory does not always translate to a contiguous segment of physical memory [9]. The existence of virtual memory does not align well with the assumptions the MAUI makes. The MAUI only has access to the physical addresses, and expects the source and destination vectors to have consecutive addressing. Therefore, when a vector crosses a page boundary it's possible for the parts of the vectors that fall in different pages to be mapped to non-consecutive physical memory pages. If a user program were to issue a MAUI command for the entire vector, when the MAUI crosses the page boundary, it will be reading or modifying data that does not belong to the vectors that it is supposed to operate on. At best, this constitutes a security hole. To exploit the security hole, a malicious program would only have to issue a MAUI command to a portion of memory to which it shouldn't have access, then read or modify data. At worst, reading or modifying unexpected data can cause monumental system failure. MAUI instructions issued could modify data that the programmer did not intend, causing the entire system to crash.

To solve the previously mentioned problem, when the MAUI commands are issued, the program issuing the MAUI commands needs to know, at the time of

issue, that the commands are correct and will not step into parts of memory that aren't intended. In order to ensure correctness, the program would need to know the exact virtual address mapping on the system. However, it is unreasonable to expect the programmer to know the virtual address mapping and validate the correctness before runtime. However, the Operating System (OS) usually manages virtual memory, and could handle the correctness of MAUI commands. Therefore, the programmer would issue an OS system call asking to perform the MAUI vectors operation. The actual MAUI commands would be privileged, and would only run when the processor is running in kernel mode. The OS would then check to see if the source or destination vectors cross page boundaries. If the vectors do cross page boundaries, they would be split up into separate MAUI instructions, and issued to the MAUI hardware. This way, the OS would be able to enforce any security procedures it needed to, and it would also be able to check the mapping to ensure that no incorrect behavior would accidentally occur.

Section 3.4 introduces the simulation environment created to test the performance of the MAUI architecture. The simulation environment is based on a version of SimpleScalar v4.0. Specifically, the MAUI architecture augments the Micro-Architectural Simulation Environment (MASE) simulator within SimpleScalar.

3.4 Simulation Environment

The simulation environment used to simulate MAUI performance is based on the popular simulation environment, SimpleScalar [1]. SimpleScalar was chosen for simulation because it already possesses a detailed simulation of caches and out-of-order execution. Additionally, Dr. Bruce Jacob and David Wang of the

University of Maryland's Electrical and Computer Engineering Department have created a highly detailed, DRAM-based memory system enhancement to the Micro-Architectural Simulation Environment (MASE) portion of SimpleScalar v4.0.

The MASE simulator provides a highly detailed, out-of-order simulation environment with support for a non-deterministic memory system. As illustrated in Figure 3.6, MASE simulates 4 pipeline stages with a number of functional units and a flexible interface to memory. While a four-stage pipeline does not exactly mimic any commercially available microprocessor, the design is a general representation of a "modern" out-of-oder execution, super-scalar microprocessor. Most importantly however, MASE allows for the simulation of non-deterministic memory systems. A non-deterministic memory system is any memory system in which later memory accesses could effect the latency of previous memory accesses. Non-deterministic memory systems include any memory system that can reorder memory accesses to improve performance and intelligent memory systems [15].

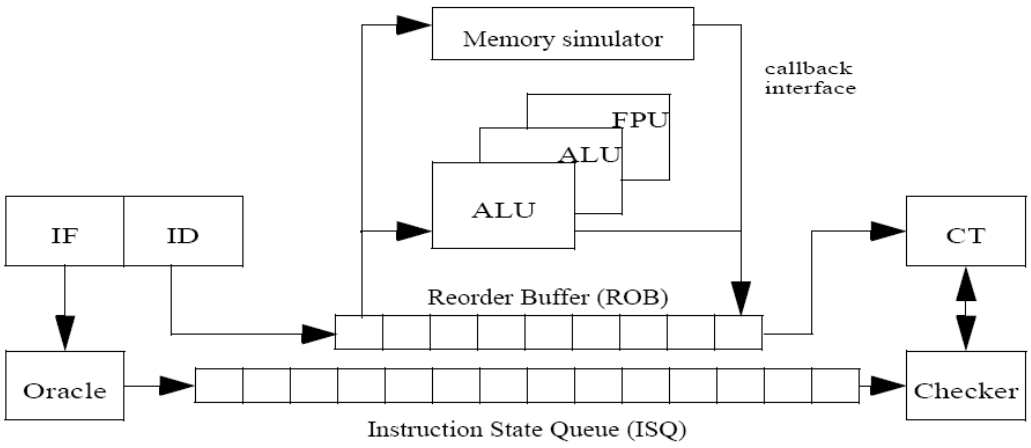

Figure 3.6: Block diagram of the MASE performance model [15].

MASE's ability to simulate non-deterministic memory systems is crucial to simulate the MAUI enhanced memory system. As MAUI operations can be extremely complex, the latency of these commands are not known at the time they are issued. Also, the highly detailed DRAM-based memory system simulator created at the University of Maryland for the MASE simulator allows for highly faithful simulations of numerous DRAM systems under various conditions.

The MAUI architecture calls for additional instructions to be added to the host processor's ISA. For SimpleScalar, the addition of new instructions is facilitated by using the "annote" field that is available for any instruction in the SimpleScalar Portable Instruction Set Architecture (PISA). As of this time, no MAUI enabled compiler has been developed. Therefore, benchmarks written for the MASE simulator that take advantage of the MAUI architecture need to be hand optimized and written partially in assembly language. Table 3.2 shows the instruction format for MAUI commands using in SimpleScalar's PISA compatible code and the "annote" field.

3.4.1 MAUI Enhancements to SimpleScalar

This section contains a description of the additions made to SimpleScalar's MASE simulator to simulate the MAUI hardware. The actual code that was created or modified for the purpose of simulating the MAUI hardware is included in Appendix A.

The simulated MAUI architecture is divided into two discrete parts, the processor-side and memory-side MAUI hardware. The processor-side part of the MAUI simulated architecture actually has no counterpart in the physical implementation. However, in the MASE simulator, the memory system actually

Name	Assembly Code	Description
maui-LD-a(x)	ADDU/15:0(13) $0, *x*, $0	Loads the maui src. A address stored in register *x*
maui-LD-a&b(x,y)	ADDU/15:0(2) $0, *x, y*	Loads the maui src. A with address in register *x*, and src. B with register *y*
maui-LD-b(x)	ADDU/15:0(3) $0, *x*, $0	Loads the maui src. B with address in register *x*
maui-LD-c(x)	ADDU/15:0(4) $0, *x*, $0	Loads the maui dest. C with address in regiuster x
maui-LD-size(x)	ADDU/15:0(5) $0, *x*, $0	Loads the maui block size with register x
maui-ADD-scale(x)	ADDU/15:0(12) $0, *x* $0	Adds register x to every element in the src. vector A and stores the result in dest. vector C
maui-ADD( )	ADDU/15:0(1) $0,$0,$0	Vector addition, dest. C = src. A + B
maui-MUL( )	ADDU/15:0(6) $0,$0,$0	Vector multiplication, dest C = src. A * B
maui-MUL-scale(x)	ADDU/15:0(7) $0, *x* $0	Multiplies register *x* to every element in the src. vector A, and stores the result in dest. vector C

Table 3.2: PISA assembly MAUI instructions.

holds no data and is only used for timing of memory system accesses. Therefore, the processor-side part of the simulated MAUI architecture performs the MAUI operations while the memory-system side of the simulated MAUI architecture determines the timing of the MAUI operations. Because the processor-side part of the simulated MAUI architecture performs the arithmetic for MAUI operations, it also maintains copies of the MAUI registers.

Within the processor's pipeline, when a MAUI instruction reaches the execute

pipeline-stage, the input dependencies have all been resolved. So, it is in the execute stage that the MAUI instruction is detected and saved to the reorder buffer. The input register values are also saved to the reorder buffer. In the commit pipeline-stage, the MAUI instructions are read out of the reorder buffer and sent to the memory system. For execution MAUI commands, the commit pipeline-stage is where the processor-side part of the simulated MAUI architecture performs the arithmetic for the MAUI instruction. For setup MAUI commands, the correct values are simply saved in the processor-side copies of the MAUI registers. The MAUI instructions are sent to memory in the commit stage to ensure that the memory system receives the MAUI instructions in program order.

The memory-side MAUI hardware is implemented almost identically as described in previous sections, except no arithmetic is actually performed. If the memory system receives a setup MAUI instruction, the appropriate value, which is passed along with the MAUI instruction, is loaded into the appropriate MAUI register. If the memory system receives an execution MAUI instruction, the MAUI operation is started and the appropriate memory operations are issued. On every memory system clock cycle, the memory-system side of the MAUI architecture simulation checks to see if the MAUI architecture has received any data from the DRAM system, and if any new memory transactions can be issued to the DRAM system. If data has arrived from the DRAM system, the MAUI checks to see if there are enough operands in the MAUI's cache to perform some arithmetic. If the MAUI determines that there are the required operands to perform arithmetic, the MAU is simulated to perform that arithmetic. The MAU is simulated to perform eight integer arithmetic operations in parallel, with addition taking a single clock cycle and multiplication taking three clock cycles. After the

MAU is finished simulating the latency of the arithmetic, the results are sent out to memory. If the MAUI is idle, these checks aren't made to try to optimize the performance of the simulation.

The next chapter presents the performance results of the simulated MAUI hardware. To test the performance of the MAUI-enhanced intelligent memory system, three benchmarks were created and simulated using the MAUI-enhanced version of SimpleScalar discussed in this chapter. The first two benchmarks were created to test what data sizes, memory system types, and processor speeds the MAUI-enhanced memory system performs well with. The final benchmark represents an accepted benchmark to test total memory system performance. The simulation results presented in Chapter 4 show that combining performance advantages of the MAUI-enhanced memory system and parallel execution of the MAUI hardware and the host processor can result in application speedup of up to 121%.

Chapter 4: Simulation Results and Conclusions

This chapter presents the simulation test results for the MAUI enhanced computer system and draws conclusions based on these results. Three benchmarks were written to test the performance of a MAUI enhanced architecture. The first two benchmarks, *MAUI-one* and *MAUI-two*, are "artificial," in that they do not necessarily reflect the behavior of real-world applications and were written specifically to test the MAUI architecture. The purpose of *MAUI-one* and *MAUI-two* was to determine under what circumstances the MAUI architecture can positively effect system performance. To that end, both benchmarks were simulated with a number of problem sizes, processor speeds, memory configurations, and cache configurations. Then, favorable processor, memory, and cache configurations were determined using the results of the *MAUI-one* and *MAUI-two* simulations.

The final benchmark, *Stream*, originally written by John D. McCalpin [16], tests the performance of the memory system by performing vector additions, multiplications, scaling, and multiplication-accumulation on extremely large vectors. The purpose of the *Stream* benchmark was to test how the MAUI enhanced intelligent memory system affects total memory system performance. All benchmarks were simulated using the MAUI enhanced version of SimpleScalar v4.0 discussed in Chapter 3.

Two versions of each benchmark were created. The first version, designated as the "unoptimized" version, is completely conventional and does not take advantage of the MAUI hardware at all. The second version, designated as the

"MAUI optimized" version, performs part of the program utilizing the MAUI hardware. The complete source code for both versions of *MAUI-one, MAUI-two*, and *Stream* are included in Appendix B.

The processor configuration used in every simulation is noted in Table 4.1. Every simulation used the same processor configuration, unless otherwise noted.

The remainder of Chapter 4 is broken in to four sections. Section 4.1, Section 4.2, and Section 4.3 describe the methodology used to simulate the MAUI architecture's performance using the *MAUI-one*, *MAUI-two*, and *Stream* benchmarks, respectively. Section 4.4 presents the results of the MAUI simulations, and draws conclusions based on those results.

4.1 Simulation Methodology for *MAUI-one*

The first benchmark, *MAUI-one*, performs a single vector addition operation, where individual members of two arrays are added together, and the result is stored in a third array. Pseudo-C code for both the unoptimized and MAUI optimized versions of the *MAUI-one* benchmark are shown in Figure 4.1. As Figure 4.1 shows, the unoptimized version of *MAUI-one* implements the vector addition in a simple *for* loop, while the MAUI optimized version of *MAUI-one* implements the vector addition with four MAUI commands. Not shown in Figure 4.1 is that in both the unoptimized and MAUI optimized version of the benchmark, the processor initializes the arrays.

The *MAUI-one* benchmark was simulated using MAUI enhanced version of SimpleScalar v4.0. Across simulations, processor speed was varied between 900 and 2900 MHz, problem size was varied between 1000 and 64,000 integers per vector, memory bus speed was varied between 100 and 800 MHz, and the mem-

General	
Issue Width	4 micro-ops/cycle
Instruction Fetch Queue Size	16
Load Store Queue Size	8
Reorder Buffer Size	16
Number of Reservation Stations	16
Branch Predictor Type/Size	Bimodal/2048 entries
Functional Units	
Number of Integer ALU's/Latency	4 / 1 cycle
Number of Multiply/Dividers	1
Multiply Latency	7 cycles
Divide Latency	12 cycles
Number of Memory Ports	2
Number of Floating Point(FP) Units	1
FP Add latency	4 Cycles
FP Multiply Latency	4 Cycles
FP Divide Latency	12 Cycles
Cache Configuration	
Level 1 Data Cache Size	16 KByte
Associativity	4 way
Block Size	32 Bytes
Latency	1 cycle
L1 Instruction Cache Size	16 KByte
Associativity	Direct Mapped
Block Size	32 Bytes
Latency	1 cycle
L2 unified Cache Size	256 KByte
Associativity	4 way
Block Size	32 Bytes
Latency	6 cycles

Table 4.1: Processor configuration.

```
/* MAUI-one, c = a + b */

int main() {
    int i, a[N], b[N], c[N];
    for(i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

```
/* MAUI-one -- using MAUI
   hardware, c = a + b */
int main() {
    int i, a[N], b[N], c[N];
    int size = sizeof(int) * N;
    maui-LD-size(size);
    maui-LD-ab(a,b);
    maui-LD-c(c);
    maui-ADD();
}
```

Figure 4.1: The pseudo-C representation of the unoptimized and MAUI optimized versions of the *MAUI-one* benchmark.

ory system type was varied between *SDRAM, DDR-SDRAM,* and *DRDRAM.* For each combination of processor speed, problem size, memory bus speed, and memory system type, two simulations were run: the first simulation used the MAUI optimized version of *MAUI-one* whereas the second used the unoptimized version of *MAUI-one.*

The percent speedup due to the MAUI optimization was calculated by:

$$\text{percent-speedup} = \left(\frac{\text{sim-cycle}_{non-MAUI}}{\text{sim-cycle}_{MAUI}} - 1 \right) * 100\%, \tag{4.1}$$

where $\text{sim-cycle}_{non-MAUI}$ is the number of processor cycles SimpleScalar reported that the unoptimized version of *MAUI-one* took to complete and sim-cycle_{MAUI} is the number of processor cycles SimpleScalar reported that the MAUI optimized version of *MAUI-one* took to complete. Equation 4.1 is only applied to two simulations with identical processor speeds, problem sizes, memory bus speeds,

and memory system types. In this way, only simulations with identical hardware configurations are compared directly using simulated execution time. Simulations with different processor speeds, problem sizes, memory bus speeds, or memory system types are compared using percent speedup due to MAUI optimization, as described above.

The *MAUI-one* benchmark tests how well the MAUI architecture can perform a single vector operation. Assuming that the MAUI optimized version of *MAUI-one* performs virtually all of the benchmark's execution using the MAUI hardware, the percent speedup due to MAUI optimization approximates the speedup due to MAUI optimization for a single vector operation. Therefore, the *MAUI-one* benchmark's simulations can provide cues to what vector operations should be off-loaded to the MAUI hardware for performance gains.

4.2 Simulation Methodology for *MAUI-two*

The second benchmark, *MAUI-two* performs two identical vector additions, except each vector addition reads and stores to different arrays. The pseudo-C code representation of both the MAUI optimized and unoptimized versions of *MAUI-two* are shown in Figure 4.2. As shown in Figure 4.2, the MAUI optimized version of *MAUI-two* performs the first vector addition using the MAUI hardware while the second vector addition is performed by the main processor. The unoptimized version, of course, does not utilize the MAUI hardware. The function *maui_add(c,a,b,size)* used in the MAUI optimized version of *MAUI-two* in Figure 4.2 is a function consisting of four instructions. The first three instructions setup the MAUI hardware by loading the MAUI's destination, sources, and the size registers with c, a and b, and size from the function call, respectively.

The final MAUI instruction is the MAUI-add instruction, which starts the MAUI execution. The function *maui_add(c,a,b,size)* implements exactly the same vector addition used in the *MAUI-one* benchmark.

```
/* MAUI-two, Two Array
   Additions */
int main() {
    int a[N], b[N], c[N], i;
    int d[N], e[N], f[N];

    for(i = 0; i < N; i++)
        c[i] = a[i] + b[i];
    for(i = 0; i < N; i++)
        f[i] = d[i] + e[i];
}
```

```
/* MAUI-two, using the
   MAUI hardware */
int main() {
    int a[N], b[N], c[N], i;
    int d[N], e[N], f[N], size;
    size = sizeof(int) * N;
    maui_add(c,a,b,size);

    for(i = 0; i < N; i++)
        f[i] = d[i] + e[i];
}
```

Figure 4.2: The pseudo-C code representation of the unoptimized and MAUI optimized versions of the *MAUI-two* benchmark.

MAUI-two simulations were conducted very similarly to *MAUI-one*. Using the MAUI enhanced version of SimpleScalar v4.0, *MAUI-two* was simulated with processor speed varying between 1000 and 3000 MHz, memory bus speed varying between 100 and 800 MHz, problem size varying between 1000 and 64,000 integers per array, and the memory system type varying across *SDRAM, DDR-SDRAM,* and *DRDRAM*. Identical to *MAUI-one*, for each combination of processor speed, memory bus speed, problem size, and memory system type, both the MAUI optimized and unoptimized versions of *MAUI-two* were simulated using SimpleScalar.

Again, calculating the percent speedup for *MAUI-two* due to the MAUI optimization is very similar to calculating the percent speed up for the *MAUI-one* simulations. Looking only at simulations with identical processor speed, memory bus speed, problem size, and memory type, the percent speedup due to the MAUI optimization was calculated by Equation 4.1, where sim-cycle$_{non-MAUI}$ is the number of processor cycles SimpleScalar reported that the unoptimized version of *MAUI-two* took to complete and sim-cycle$_{MAUI}$ is the number of processor cycles SimpleScalar reported that the MAUI optimized version of *MAUI-two* took to complete.

The *MAUI-two* benchmark tests how well the MAUI hardware and the processor are able to exploit parallelism when both are performing memory intensive tasks. Because the two datasets used in *MAUI-two* are completely independent, the MAUI optimized version of *MAUI-two* should be able to have the processor and the MAUI hardware execute in parallel.

4.3 Simulation Methodology for *Stream*

The third and final software benchmark used to test the MAUI enhanced architecture is *Stream*, originally written by John D. McCalpin [16]. *Stream* performs vector scaling, addition, multiply-addition, and copying operations on large sections of memory. The pseudo-C code representation of both the MAUI optimized and unoptimized version of *Stream* is given in Figure 4.3. In the MAUI optimized version of *Stream*, the MAUI operations are implemented by the functions *maui_add(c, a, b, size), maui_mul_scalar(c,a, scalar, size),* and *maui_copy(c, a, b, size).* Each MAUI function is implemented using four MAUI commands, and these four MAUI functions are used very similarly to the *maui_add* func-

tion described and used in the *MAUI-two* benchmark. As shown in Figure 4.3, and similar to both *MAUI-one* and *MAUI-two*, the MAUI optimized version of *Stream* performs several vector-type operations using the MAUI hardware, while the unoptimized version performs all the vector operations within the main processor.

```
/* Stream Benchmark */
int main() {
    int a[N], b[N], c[N];
    int j, scalar;

    for(j = 0; j < N; j++)
        c[j] = a[j];
    for(j = 0; j < N; j++)
        b[j] = scalar * c[j];
    for(j = 0; j < N; j++)
        c[j] = a[j] + b[j];
    for(j = 0; j < N; j++)
        a[j]=b[j]+scalar*c[j];
}
```

```
/* MAUI Stream Benchmark */
int main() {
    int a[N], b[N], c[N];
    int j, scalar;
    int size = sizeof(int) * N;
    maui_copy(c, a, size);

    maui_mul_scalar(b,c,scalar
                    size);
    maui_add(c,a,b,size);

    for(j = 0; j < N; j++)
        a[j]=b[j]+scalar*c[j];
}
```

Figure 4.3: The pseudo-C code representation of the unoptimized and MAUI optimized versions of the *Stream* benchmark.

Unlike the *MAUI-one* and *MAUI-two* benchmarks, *Stream* was simulated with a single combination of processor speed, memory bus speed, problem size, and memory system type. Processor speed was set to 2000 MHz, memory type was

chosen as *DRDRAM*, memory bus speed was chosen as as 800 MHz, and the problem size was chosen to be two million integers per array. These values for processor speed, memory type, memory bus speed, and problem size were chosen based results from the *MAUI-one* and *MAUI-two* simulations, common sense, and common practice for the original version of the *Stream* benchmark [16].

At two million integers per array, the total data memory used in the benchmark about 23MB, which is much larger than the 8KB L1 data-cache and 256KB L2 cache used for the simulation of *Stream*. In fact, at over 7MB per array, no single vector used in the *Stream* benchmark can fit in the caches. This keeps with the common practice of choosing the array sizes for *Stream* to be much larger than the cache size of the system that's being tested. By making *Stream's* dataset too large to fit in the cache, *Stream* provides an indication of total memory system performance. As the MAUI architecture aims to improve memory system performance, the simulated running time of the *Stream* benchmark is a good indication at how well the MAUI architecture achieves the goal of improving memory system performance.

4.4 Simulation Results

This section summarizes the simulation results, showing only those simulation results that best illustrate the effects MAUI optimization has on memory system performance. Results of all simulations can be found in Appendix C. Section 4.4 is broken into three subsections. Subsection 4.4.1 summarizes the results from simulations of the *MAUI-one* benchmark, Subsection 4.4.2 summarizes the results from the *MAUI-two* benchmark, and the final subsection, Subsection 4.4.3, summarizes the results from the *Stream* benchmark.

4.4.1 Simulation Results from the *MAUI-one* Benchmark

Simulations of *MAUI-one* showed that the speedup due to the MAUI architecture is dependent on memory system type, memory bus speed, processor speed, problem size, and cache configuration. Figure 4.4 illustrates the effect memory performance has on the speedup due to the MAUI architecture. Specifically, as the memory bandwidth increases, the speedup due to the MAUI architecture increases. Note that for the memory systems shown in Figure 4.4, memory bandwidth, from smallest to largest, is *SDRAM* 100 MHz, *SDRAM* 133 MHz, *DDR-SDRAM* 166 MHz, *DDR-SDRAM* 232 MHz, *DRDRAM* 400 MHz, and *DRDRAM* 800 MHz. The trend illustrated in Figure 4.4 is repeated for each combination processor speed and problem size; refer to Appendix C for all other simulation results.

Figure 4.4 illustrates how, as the memory system performance increases, the speedup due to optimization for the MAUI architecture increases. Intuitively, this performance trend makes sense. Recall that, because MAUI architecture is located within the memory system, its performance is limited by the memory system performance. That means that for a 100 MHz *SDRAM* system, the MAUI is only operating at 100 MHz. As the performance of the memory system increases, the speedup due to the MAUI architecture also increases. In examination of Figure 4.4, notice that the break-even point seems to be at *SDRAM* 133 MHz, where the MAUI optimized version of *MAUI-one* performs slower than the unoptimized version when run with memory systems slower than *SDRAM* 133 MHz and faster when run on memory systems faster than *SDRAM* 133 MHz.

The effect of processor speed on the speedup due to the MAUI architecture is shown in Figure 4.5. Figure 4.5 shows that, as the processor speed increases, the

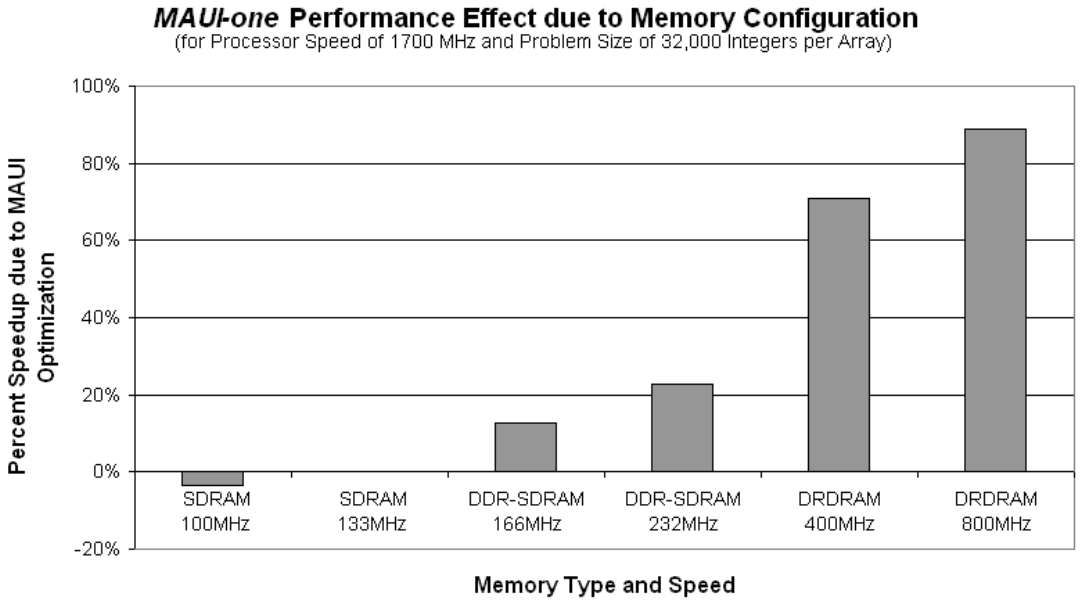

Figure 4.4: Graph illustrating the effect memory configuration has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark simulated with a processor speed of 1700 MHz and a problem size of 32,000 integers per array.

percent speedup due to optimization for the MAUI architecture decreases. Again, this intuitively makes sense. For faster processors, the arithmetic performed by the MAUI becomes comparatively more expensive.

The effect problem size has on the speedup due to the MAUI architecture is shown in Figure 4.6. Examining Figure 4.6 one can deduce that, as the problem size increases, the percent speedup due to optimization for the MAUI architecture also increases. There are two reasons for speedup increasing while the problem size increases. First, the cost of initializing the MAUI hardware is the same for across all problem sizes. That means that the startup costs of sending the setup MAUI instructions to the MAUI hardware is amortized over a larger amount of

computation for larger problem sizes.

The second reason the speedup of *MAUI-one* increases when problem size increases is that although the MAUI hardware's efficiency in accessing memory doesn't change when operating on large vectors, the processor's effective efficiency decreases significantly on large datasets. The processor's effective efficiency decreases when operating on large vectors because when a dataset can fit in the cache, the processor has very high bandwidth, low latency access to the data, but when the dataset grows so that it no longer fits in the cache, the processor must access main memory, and its efficiency in accessing memory significantly decreases. For smaller datasets, the data is loaded into the cache by the initialization, and the data remains there for the entire simulation. Figure 4.7 shows how as the cache configuration becomes better, the percent speedup due to the MAUI architecture decreases for the same problem size. Note that there are two variables that make a cache "better" in Figure 4.7. First, and most obviously, a larger cache has better performance. Secondly, the cache's associativity effects performance. For the cache configurations shown in Figure 4.7, the caches' performances, from best to worst, are the 512 KB 4-way cache, 256 KB 8-way cache, 256 KB 4-way cache, and then 256 KB 2-way cache. As the performance of the cache increases, the percent speedup due to MAUI optimization decreases. Because the MAUI hardware is aimed at improving the performance of those operations that access memory often, the fact that the MAUI hardware doesn't speedup those operations that already fit well in the cache is not a drawback to the MAUI architecture.

Additionally, the "knee" in Figure 4.6's performance graph can also be explained by examining the effect cache has on the speedup due to the MAUI

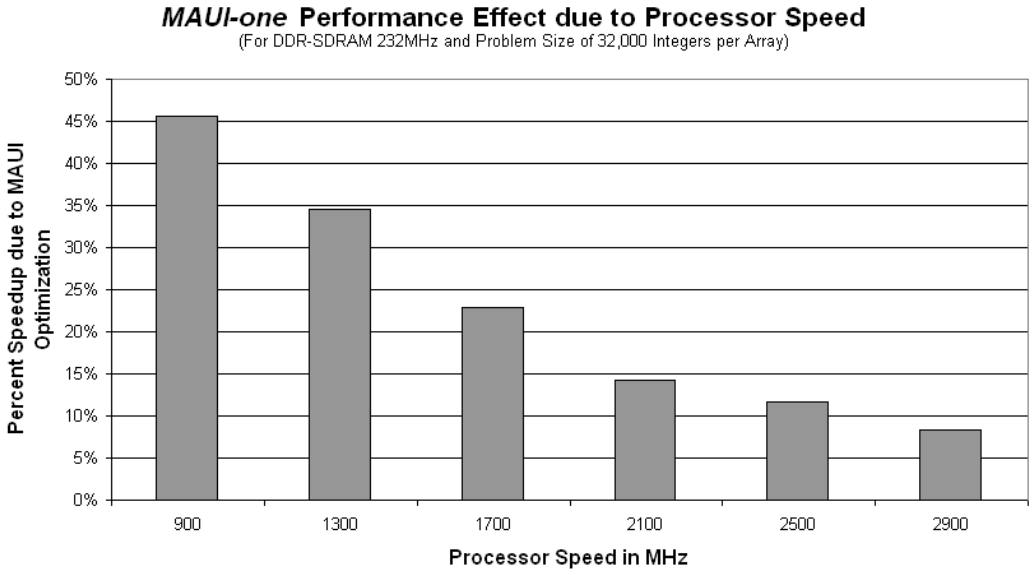

Figure 4.5: Graph illustrating the effect processor speed has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark.

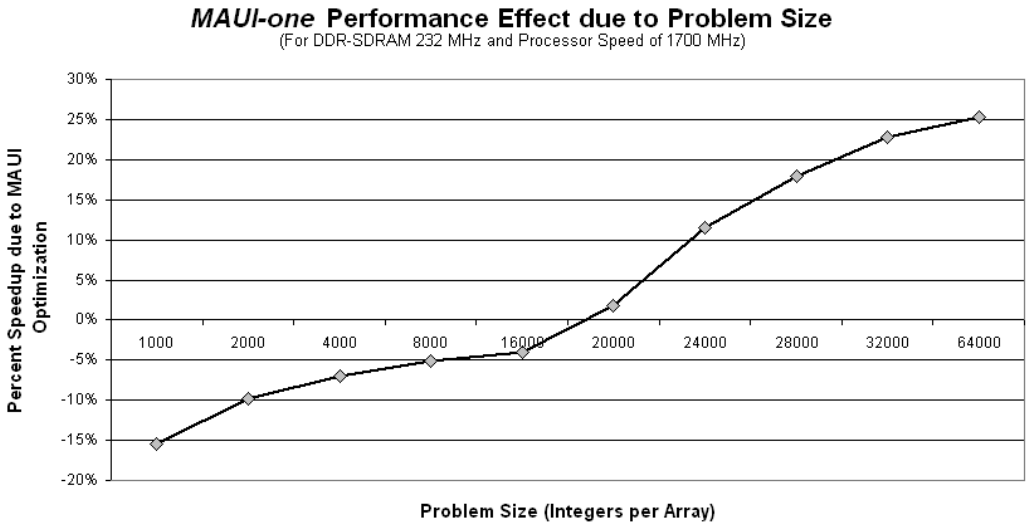

Figure 4.6: Graph illustrating the effect problem size has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark.

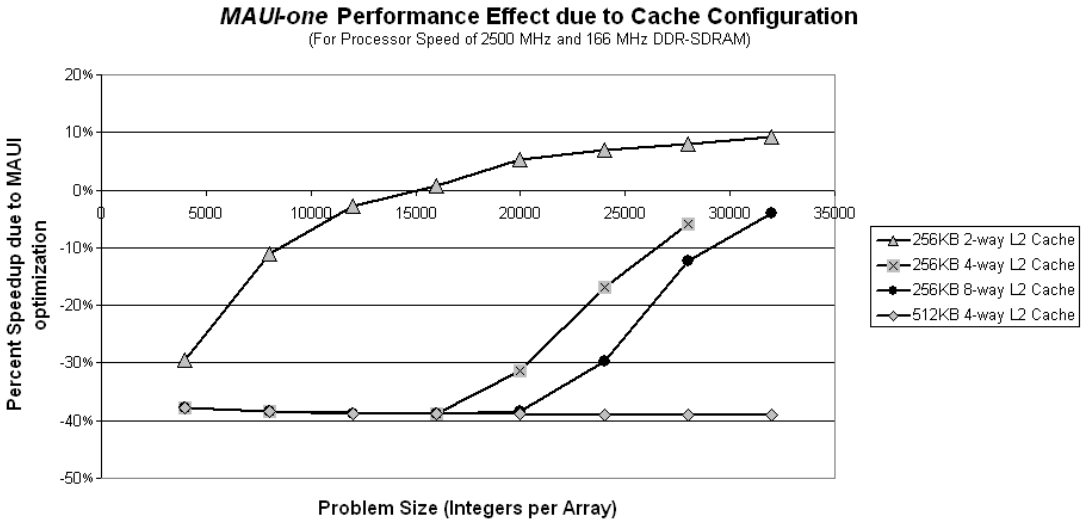

Figure 4.7: Graph illustrating the effect cache configuration has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark.

optimizations. The "knee" in Figure 4.6 is at about 20,000 integers per array, where the percent speedup due to MAUI optimization suddenly increases. For a 256KB four-way L2 cache, the unoptimized version of of *MAUI-one* no longer comfortably fits in the cache at about 20,000 integers per array. Therefore, the running time significantly increases at about 20,000 integers per array. Figure 4.7 shows how the performance "knee" moves as the cache configuration changes. As the cache configuration improves, the "knee" moves towards larger problem sizes, illustrating that the MAUI hardware provides the largest speedup when working on datasets that do not fit in the cache.

Across all simulations of *MAUI-one*, the largest speedup due to MAUI optimizations was 102.6%. The 102.6% speedup was realized when run on a system with a 900 MHz processor, and an 800 MHz DRDRAM memory system. At 900 MHz, this simulation used the slowest processor and the 800 MHz DRDRAM

system was the highest bandwidth memory system tested. Additionally, the problem size for the simulation of *Stream* was set to 100,000 integers per array, the largest problem size simulated. This all follows the conclusions drawn earlier, which predict that the speedup due to MAUI optimizations increases as the processor speed decreases, the memory system's speed increases, and the problem size increases.

Because the *MAUI-one* benchmark performs a single vector addition, *MAUI-one* demonstrates the speedup achievable for a single vector operation when performed by the MAUI architecture instead of using the processor. The MAUI architecture is able to perform these vector operations more efficiently than the processor because, although the MAUI hardware doesn't posses as much computational power as the processor, it experiences a higher bandwidth, lower latency connection to memory. Therefore, the results of *MAUI-one* shows that, for a large enough data set running on a system with a low performance processor and a high performance memory system, vector operations can run slightly more than twice as fast when run on MAUI hardware instead of using the processor.

Throughout Section 4.4.1, the results were illustrated using a single combination of either processor speed, memory system type, or problem size. However, the trends illustrated in Figures 4.4, 4.5, and 4.6 are repeated across all simulations. Refer to Appendix C to see all the other simulation results.

4.4.2 Simulation Results from the *MAUI-two* Benchmark

As with the *MAUI-one* benchmark, the speedup due to MAUI optimizations for the *MAUI-two* benchmark is dependent on memory system configuration, processor speed and problem size. Figure 4.8 shows the effect memory configuration

has on the speedup due to MAUI optimizations; as the memory systems' possible bandwidth increases, the speedup due to MAUI optimizations increases. The reason for this performance trend is identical to the reason why the memory system affects the speedup due to MAUI optimizations for the *MAUI-one* benchmark: because the MAUI architecture is located in the memory system, its performance is limited to be the same as the memory system. Again, note that for the memory systems shown in Figure 4.8, possible memory bandwidth, from smallest to largest, is *SDRAM* 100 MHz, *SDRAM* 133 MHz, *DDR-SDRAM* 133 MHz, *DDR-SDRAM* 166 MHz, *DDR-SDRAM* 266 MHz, *DDR-SDRAM* 333 MHz, and *DRDRAM* 800 MHz.

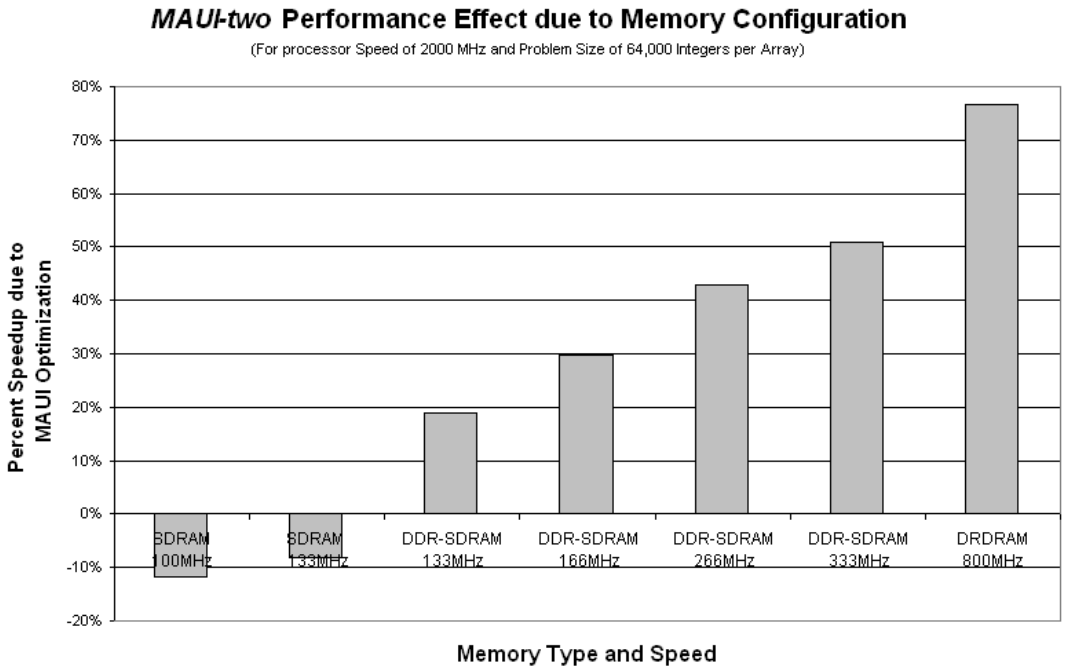

Figure 4.8: Graph illustrating the effect memory configuration has on the speedup due to the MAUI architecture for the *MAUI-two* benchmark.

Figure 4.9 shows the effect processor speed has on the speedup of *MAUI-*

two due to MAUI optimizations. The trend shown in 4.9 is the same as the trend shown in 4.5, illustrating the effect processor speed has on the speedup of the *MAUI-one* benchmark. As the processor speed increases, the speedup due to MAUI optimization decreases. The reason for this trend is that as the processor's performance increases, performing arithmetic with the MAUI hardware becomes relatively more expensive.

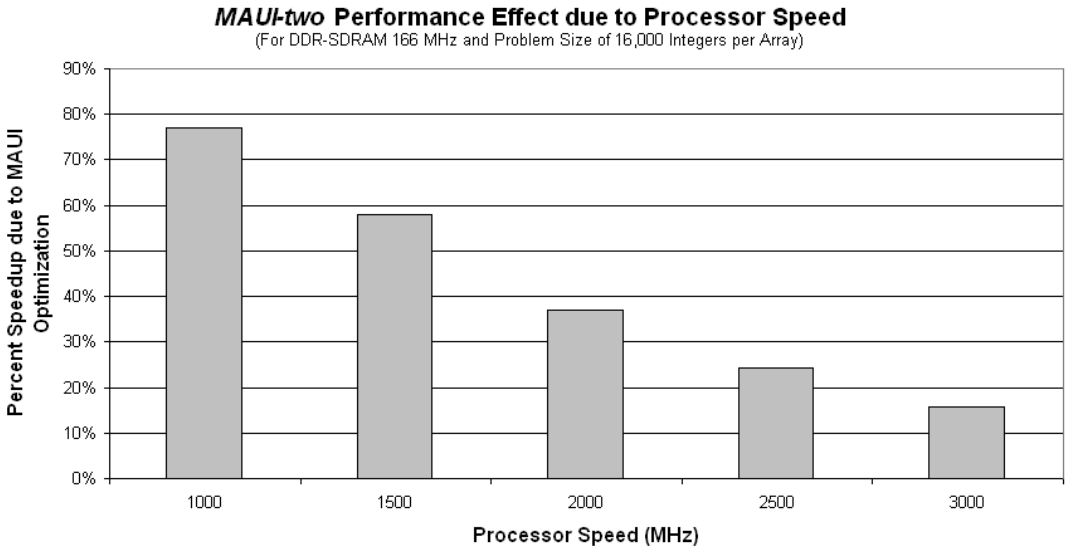

Figure 4.9: Graph illustrating the effect processor speed has on the speedup due to the MAUI architecture for the *MAUI-two* benchmark.

Simulations show that the speedup due to MAUI optimization for *MAUI-two* are also dependent on problem size. The trend for *MAUI-two* is very similar to that shown in Figure 4.6 for *MAUI-one*: as the problem size increases, the speedup due to MAUI optimization increases. The effect problem size has on the speedup of *MAUI-two* is shown in Figure 4.10.

There is one significant difference when comparing the effect problem size has

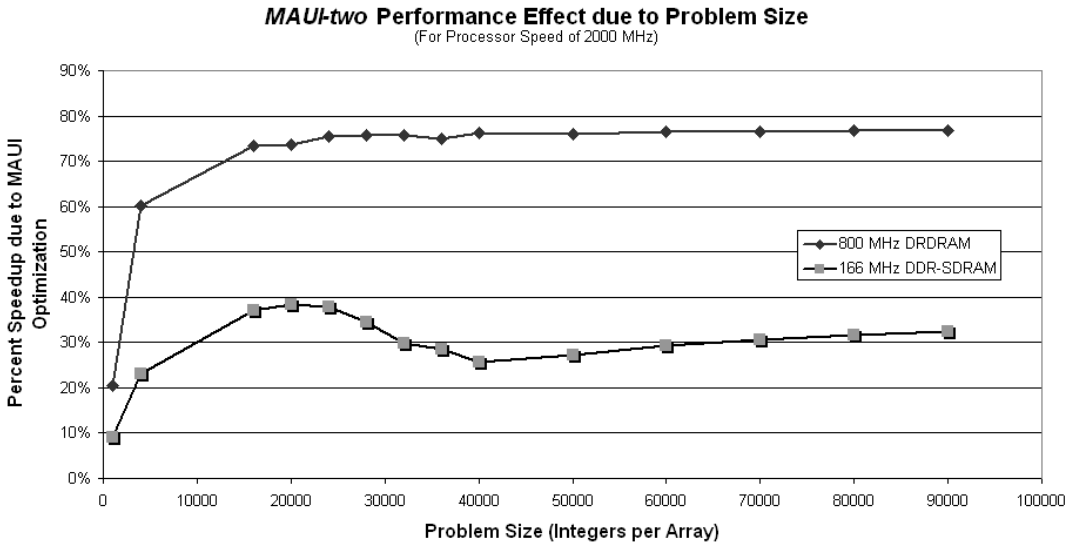

Figure 4.10: Graph illustrating the effect problem size has on the speedup due to the MAUI architecture for the *MAUI-two* benchmark.

on the speedup due to MAUI optimization for *MAUI-one* (Figure 4.6) to the effect problem size has on the speedup due to MAUI optimization for *MAUI-two* (Figure 4.10). Looking at Figure 4.10, notice that for 166 MHz *DDR-SDRAM* the speedup shows a noticeable decline when the problem size reaches about 20,000 integers per array.

Remember that the MAUI optimized version of the *MAUI-two* benchmark performs two vector additions, one in memory and the second in the processor, while the unoptimized version performs all the vector operations using the processor. For the MAUI optimized version of *MAUI-two*, when the problem size grows to 20,000 integers per array, the data set the processor is working on can no longer comfortably fit in the cache. Notice the point where the data set can no longer fit comfortably in the cache is the same problem size the the *MAUI-one* benchmark experiences the performance "knee". However, for *MAUI-two* this translates to

a decline in the speedup, instead of the percent speedup increases, as is seen in the *MAUI-one* benchmark. This is because once the processor's dataset can no longer fit in the cache, the processor and the MAUI hardware begin competing for access to memory. Because both the processor and the MAUI hardware are accessing memory in parallel, each now only has access to half the available memory bandwidth. There is still a significant speedup however, because of the significant amount of parallel execution. The percent speedup decline starting at about 20,000 integers per array is not as significant for the 800 MHz *DRDRAM* curve shown in Figure 4.10 because the bandwidth available for that memory system type is significantly greater than that of 166 MHz *DDR-SDRAM*.

The largest speedup due to MAUI optimization for the *MAUI-two* benchmark was found to be 80.1%. The 80.1% speedup was realized when run on a 2500 MHz processor with 400 MHz *DRDRAM* memory configuration and a problem size of 64,000 integers, the largest problem size simulated for the *MAUI-two* benchmark. Although the largest speedup for *MAUI-two* was not realized using the fastest memory , as predicted in Figure 4.8, the fastest memory system tested, 800 MHz DRDRAM, does show a 78.8% speedup on *MAUI-two* (which is not significantly slower than the 80.1% speedup experienced with 400 MHz *DRDRAM*). Additionally, although the largest speedup for the *MAUI-two* benchmark was not realized using the slowest processor, as would be predicted by the trends shown in Figure 4.9, the speedup *MAUI-two* experienced on a 1000 MHz processor (the slowest processor simulated for the *MAUI-two* benchmark) was 76.5%, which is also not significantly lower than the 80.1% speedup when run on a 2500 MHz processor.

The speedup of the *MAUI-two* benchmark due to MAUI optimization is due to exploiting the parallelism of two separate vector operations. Performing the

vector operations of *MAUI-two* in parallel falls short of a two processing element perfect parallel execution speedup by only about 20%. The 20% parallel execution overhead would be expected to shrink if the operations performed by the processor didn't compete with the MAUI hardware for memory access.

Throughout Section 4.4.2, the results were illustrated using a single combination of either processor speed, memory system type, or problem size. However, the trends illustrated in Figures 4.8, 4.9, and 4.10 are repeated across all simulations. Refer to Appendix C to see all the other simulation results.

4.4.3 Simulation Results from the *Stream* Benchmark

The results of simulations of *MAUI-one* and *MAUI-two* indicate that higher-performance memory and a lower performance processor and cache result in a higher performing MAUI architecture. Therefore, to simulate *Stream*, the memory system was chosen to be DRDRAM running at 800 MHz, the highest performing, real-world memory system supported by SimpleScalar v4.0. The processor's frequency was chosen to be 2 GHz, which being neither laboriously slow nor as fast as is currently available, appears to be a good choice to parallel real-world, mid-range consumer computer systems. Additionally, the cache was chosen to be 512KB, 4-way set associative. Again, the cache configuration seems to parallel real-world, mid-range consumer computer systems. The problem size for *Stream* was set to twenty-million integers per array. At twenty million integers per array, the problem size follows common practice for the original *Stream* benchmark [16].

Simulating the *Stream* benchmark showed a 121.5% speedup due to MAUI optimization. There are two reasons behind *Stream* benchmark's speedup. First, referring back to Figure 4.3, the first three vector operations are performed with

the MAUI hardware. As shown in simulations of *MAUI-one*, these vector operations can complete about twice as fast as the corresponding vector operations in the unoptimized version of *Stream*.

The second reason for the 121.5% speedup of *Stream* due to MAUI optimization is found examining how, in the MAUI optimized version of *Stream*, the final vector operation is performed using the processor. Although the final vector operation performed by the processor and the preceding MAUI operation both operate on the same data, the MAUI hardware allows significant execution overlap. The processor is permitted to start execution of the final vector operation before the preceding *maui_add* is finished. Therefore, while the processor is executing the beginning of the final vector operation, the MAUI hardware is executing the end of the preceding vector operation. This parallelism means that the final vector operation is mostly overlapped with the preceding vector operation. If it were completely overlapped, then the speedup due to MAUI optimization would be expected to be about 166%. At 121.5%, the simulated speedup is significantly less than 166% because the final vector addition takes longer than the preceding *maui_add* operation, meaning that its execution is not completely overlapped with the *maui_add* operation.

Figure 4.11 compares the speedup of *MAUI-one*, *MAUI-two*, and *Stream* for identical memory configurations and processor speeds. Simulations of *Stream* show how, by combining parallel execution between the processor and the MAUI hardware and by off-loading memory bound operations to the MAUI enhanced memory system, the speedup of memory bound benchmarks can exceed that predicted by the simplistic *MAUI-one* and *MAUI-two* benchmarks.

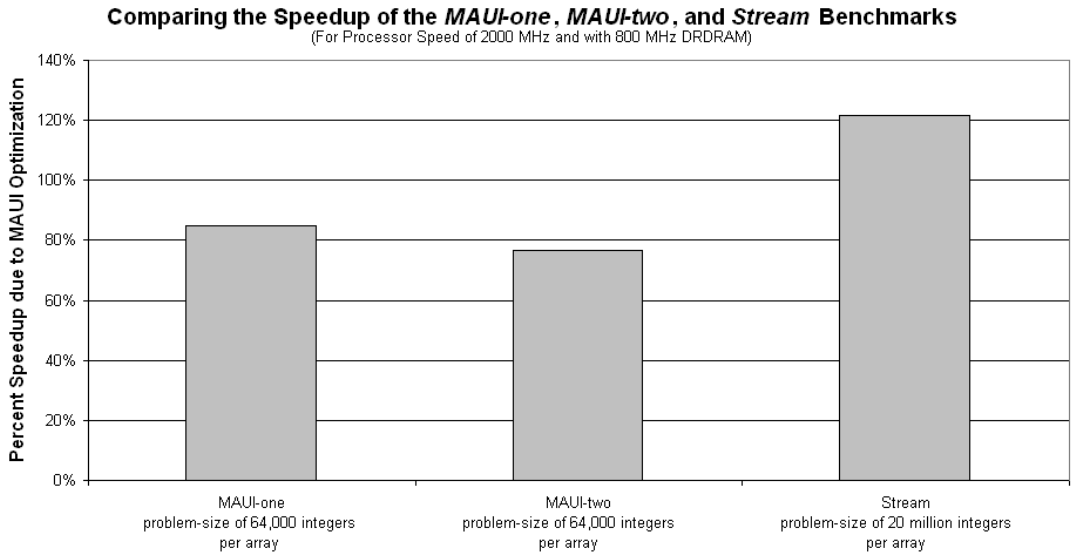

Figure 4.11: Graph comparing the speedup of *MAUI-one, MAUI-two* and *Stream* due to MAUI optimizations.

Chapter 5: Conclusions and Recommendations for Further Research

Computer system performance is greatly increasing as time progresses. In general, Moore's law predicts that the performance of a computer system will double every eighteen months. However, the memory system's performance has not increased as quickly as the processor's performance. The performance gap between the memory system and the rest of the computer system has become a performance bottleneck to total computer system performance. The memory-processor performance gap is increasing as time progresses, only making the performance bottleneck worse [9].

Intelligent memory systems represent one architectural feature that shows promise in overcoming the performance bottleneck associated with memory accesses. Any intelligent memory system builds computational ability into the memory system. Intelligent memory systems fall into one of two categories: either they migrate computational power into the DRAM system, or they migrate DRAM into the main processor [2].

Several intelligent memory systems have already been proposed, and their performance characteristics explored. The Active Pages [19] and DIVA [8] architectures represent two intelligent memory system architectures that take the former approach of migrating computational power into the DRAM system. The IRAM architecture [25] represents an intelligent memory system architecture that takes the latter approach of migrating DRAM into the processor. These architectures have shown, through simulation, to perform up to 1000 times faster on some benchmarks.

Despite impressive simulation studies, none of these proposed intelligent memory system architectures have gained popular support. One reason may be that the integration of logic and DRAM onto a single silicon die has proven to be difficult and expensive. There has already been one intelligent memory system proposed that does not require the integration of logic and DRAM onto a single silicon die. The User-Level Memory Thread (ULMT) architecture builds additional computational power into the memory controller. However, the ULMT architecture is not explicitly controlled by the application running in the processor, and is used specifically to aid in prefetching. Despite this inflexibility, the ULMT architecture has shown, in simulations, to provide up to a 58% speedup for some applications.

This thesis presents a new intelligent memory system architecture named the Memory Arithmetic Unit and Interface (MAUI) architecture. The MAUI architecture combines traits from the Active Pages, DIVA, and ULMT architectures to create a new computational model. Like the Active Pages and DIVA architectures, the MAUI architecture migrates computational power into the memory system, and the MAUI hardware is explicitly controlled by the application running in the host processor. Like the ULMT architecture, but unlike the Active Pages and DIVA architectures, the MAUI architecture does not require logic and DRAM to be integrated onto a single silicon die. The MAUI architecture integrates additional computational power onto the same chip as the memory controller. The MAUI architecture is split into to separate parts: the Memory Arithmetic Unit (MAU) and the Memory Arithmetic Unit Interface (MAUI). The MAU performs the actual arithmetic performed by the MAUI architecture, while the MAUI coordinates the data flow through the MAUI architecture.

Because the MAU is located on the same chip as the memory controller, it has a higher bandwidth, lower latency connection to memory. Because of this more efficient connection to memory, memory-bound operations can be performed more quickly by the MAUI hardware than by the processor. Additionally, because the MAUI hardware is a separate processing element from the processor, further application speedup is possible by exploiting parallelism.

5.1 Summary of Results

For the purpose of testing the performance of the MAUI architecture, the SimpleScalar v4.0 simulator was modified to include a MAUI enhanced memory system. Then, three benchmarks to test the MAUI enhanced memory system were created. The first two benchmarks, *MAUI-one* and *MAUI-two*, are "artificial," in that they do not represent real-world applications and were designed only to determine what instances the MAUI hardware performs well. Simulations of *MAUI-one* and *MAUI-two* have shown that the performance of the MAUI hardware increases as the memory system's performance increases, the problem size increases, and the processor speed decreases. Simulations of *MAUI-one* have shown that the MAUI hardware can perform a single vector operation up to 103% faster than the processor, and simulations of *MAUI-two* have shown that by using the MAUI hardware and the host processor in parallel, applications can run about 80% faster than by using the processor alone.

The final benchmark, *Stream*, is a well accepted benchmark used to test total memory system performance. Originally written by John D. McCalpin [16], *Stream* performs four vector operations on three extremely large arrays. Performing three of *Stream*'s vector operations using the MAUI hardware resulted in a

121% speedup compared to the unoptimized version in simulations. Because the MAUI optimization for *Stream* exploited both the fact that the MAUI performs vector operations faster than the processor, as well as some parallelism, *Stream* performed better than what was predicted from the *MAUI-one* and *MAUI-two* simulations.

The application speedups found when using the MAUI hardware arise because the MAUI can more efficiently stream through memory. This is because the MAUI pipelines to memory more effectively, making better utilization of the memory's available bandwidth. Referring back to Table 4.1, you can see that the processor's load/store queue has only eight entries and that there are only sixteen reservation stations. That means that, during the any of the loops in all three benchmarks, at most eight loads are are waiting for responses from the memory system. Because the cache line is thirty-two bytes, and each integer is four bytes, each cache line comprises of eight integers. That means that when the processor is waiting for loads to return from the memory system, it is probably waiting on half of a cache line from two separate sources. That means the pipeline to memory the processor uses is only two cache-lines deep. The pipeline the MAUI hardware sets up is three cache lines deep. The deeper pipeline allows the MAUI hardware to more efficiently pipeline to memory.

Further application speedups could be expected in a multiprogrammed computer system. For instance, assume that a computer system is running three separate applications that, without using the MAUI-hardware, take the same amount of time to complete. If two of the applications are memory-bound and the third is not, total running time could be reduced to 1/3 of the original running time. A 300% increase in total system performance can be realized because,

as indicated by the simulations of *MAUI-one*, *MAUI-two*, and *Stream*, the first two memory-bound applications could run about 100% faster on the MAUI hardware. Then, the final application could be executed in parallel with the first two memory-bound applications. An illustration showing how the MAUI architecture could increase total system performance by 300% on a multiprogrammed computer system is shown in Figure 5.1.

5.2 Suggestions for Further Research

The thesis concludes by suggesting three major directions for further research concerning the MAUI architecture. The first direction for further research is to expand the MAUI hardware. The second research direction is to determine for which applications the MAUI hardware would be most useful. The final direction for further research is to expand the software support for the MAUI hardware.

Currently, the MAU and the MAUI represent an extremely limited vector processor. One advantage of this limitation is that, because the data accesses of the operations the MAUI hardware are simple, cache coherence is simple. Also, vector operations represent a well understood computational model, already implemented in architectures such as the MMX and Altivec SIMD architectures. However, because of the MAUI architecture's limitations,the types of operations that the MAUI can perform are severely limited. Also, because the MAUI architecture supports only a single vector-type operation at a time and requires that all instructions come from the processor, all data dependent control flow is controlled by the processor. One direction of future work would be to make the MAUI architecture a more general purpose vector processor.

By making the MAUI hardware a more general purpose vector processor,

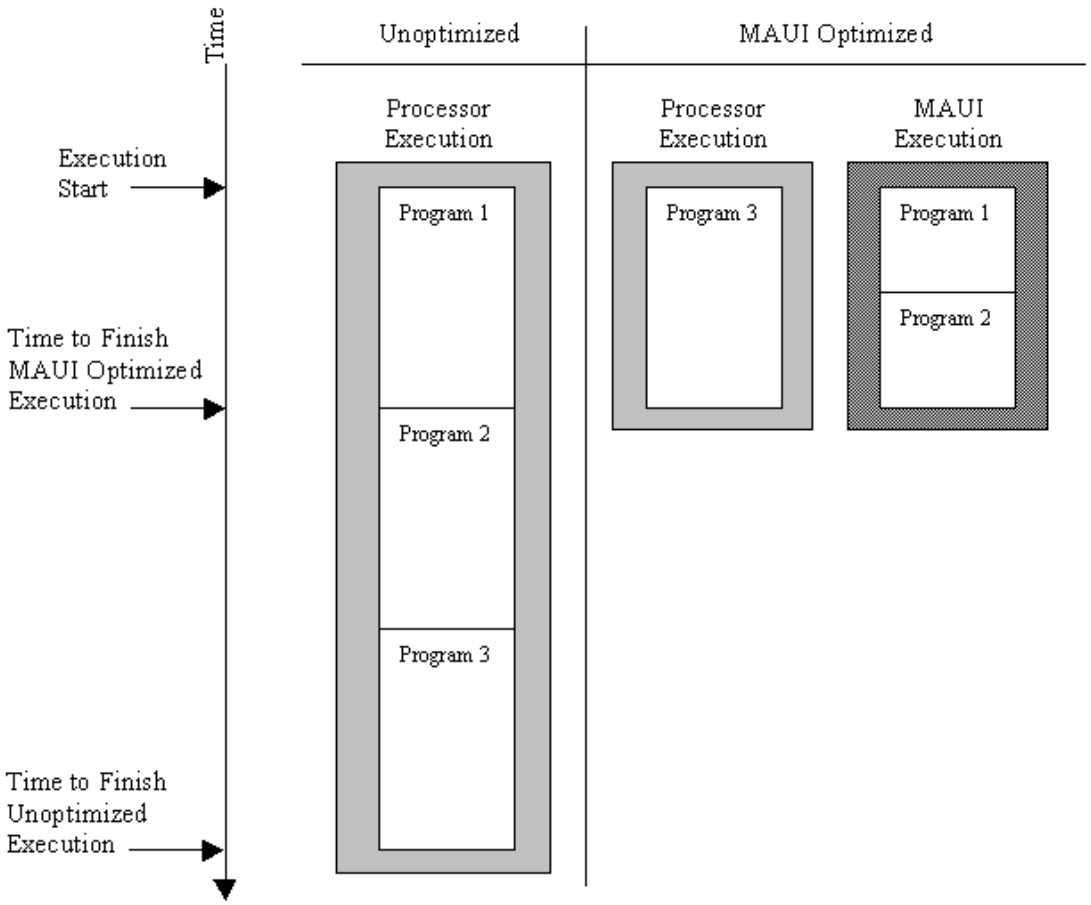

Figure 5.1: Illustration of how the MAUI architecture could increase total system performance by up to 300% on a multiprogrammed computer system. In this example, Program 1 and 2 are both memory-bound programs that take the same amount of time to complete as Program 3 when executed by the processor, but each take half that time when executed by the MAUI architecture.

many more types of memory-bound operations could be off-loaded to the MAUI hardware. For instance, pointer chasing represents one memory bound computation that the MAUI currently cannot perform. If the MAUI hardware were able to fetch and execute instructions from memory, pointer chasing may represent one type of operation the MAUI hardware could perform more efficiently than the processor. However, by creating a general purpose vector processor in the memory system, cache coherence between the main processor and the MAUI hardware becomes much more complex. The additional flexibility provided by using a general purpose vector processor in the MAUI architecture is another area that could stand further research.

The second direction of further research would be into what applications the MAUI hardware would be most useful in. For instance, operating systems are now designed assuming that copying large sections of memory is a time intensive task, and so those operations are avoided. However, the MAUI hardware not only speeds up these block copies, but also provides a separate computational engine to perform them, freeing the processor to perform other tasks. The availability of a MAUI enhanced memory system may significantly change the way that operating systems are designed and implemented.

The final suggested direction for further research is to expand the software support for the MAUI hardware. Expanding the software support for the MAUI hardware could encompass developing a MAUI aware compiler and a MAUI aware multi-tasking Operating System. Expanding the software support for the MAUI architecture is a logical extension of research in determining the kinds of applications for which the MAUI architecture would be most useful. Using this knowledge, the compiler and Operating System could then efficiently allocate

processes among the MAUI hardware and the processor.

Appendix A: The MAUI Architecture Simulator

Appendix A contains the code written to simulate the MAUI architecture within SimpleScalar v4.0's MASE simulator.

A.1 MAUI.H

```
/*
 * maui.h - simulates the MAUI addition to a memory controller/DRAM
 *          system. This is the header for the processor side of the
 *          simulated MAUI hardware.
 *
 * this file is being contributed to the SimpleScalar tool suite
 * written by Todd M. Austin as part of the Multiscalar Research
 * Project.
 *
 * Justin Teller
 * Dept of Electrical & Computer Engineering
 * University of Maryland, College Park
 * All Rights Reserved
 *
 * This software, should it be distributed, is distributed with
 * *ABSOLUTELY NO SUPPORT* and *NO WARRANTY*.  Permission is given to
 * modify this code as long as this notice is not removed.
 */
```

```
/*
 * April 2004
 */

#ifndef MAUI_H
#define MAUI_H

#define MAUI_UNDEF       -1
#define MAUI_ADDI        0
#define MAUI_ADD         1
#define MAUI_ST_RA       2
#define MAUI_ST_RA_RB    3
#define MAUI_ST_RB       4
#define MAUI_ST_RC       5
#define MAUI_ST_RC_I     6
#define MAUI_ST_SIZE     7
#define MAUI_ST_SIZE_I   8
#define MAUI_MUL         9
#define MAUI_MULI        10
#define MAUI_FP_ADD      11
#define MAUI_FP_ADDI     12
#define MAUI_FP_MUL      13
#define MAUI_FP_MULI     14

#define MAUI_RID         100

typedef struct _q_node {
```

```
  unsigned long d1, d2;
  int type;
  unsigned int lat;
  struct _q_node *next;
} maui_q_node;

typedef struct _maui_q {
  maui_q_node *head, *tail;
  int size;
} maui_q;

typedef struct _maui_ROB_info {
  int type;
  long immed, rs, rt;
} maui_ROB;

typedef struct _maui_reg_file {
  /* ra and rb are the source registers */
  md_addr_t ra;
  md_addr_t rb;
  /* rc is the destination register */
  md_addr_t rc;
  long size;
} maui_regs;

void maui_init();
```

```
/* additional coommands */
int
maui_check_tlb(unsigned long a);

void
maui_proc_side_update();

void
maui_insert_wq(md_addr_t ra, md_addr_t rb, int biu_type,
               unsigned int lat);

#endif
```

A.2 MAUI.C

```
/*
 * maui.c - simulates the MAUI addition to a memory controller/DRAM
 * system. This is actually the processor side to the MAUI hardware.
 *
 * this file is being contributed to the SimpleScalar tool suite
 * written by Todd M. Austin as part of the Multiscalar Research
 * Project.
 *
 * Justin Teller
 * Dept of Electrical & Computer Engineering
 * University of Maryland, College Park
 * All Rights Reserved
```

```
 *
 * This software, should it be distributed, is distributed with
 * *ABSOLUTELY NO SUPPORT* and *NO WARRANTY*.  Permission is given to
 * modify this code as long as this notice is not removed.
 */

/*
 * April 2004
 */

#ifndef MAUI_C
#define MAUI_C

#include "mase.h"
#include "maui2.h"
#include "mem-maui.h"

maui_regs in_proc_maui;
maui_q proc_maui_q;

void
maui_init(){
  proc_maui_q.head = NULL;
  proc_maui_q.tail = NULL;
  proc_maui_q.size = 0;

  in_proc_maui.ra = in_proc_maui.rb = in_proc_maui.rc = 0;
```

```
  in_proc_maui.size = 0;

  mem_maui_init();
}

int
maui_check_tlb(unsigned long a){

  mem_status_t result;
  int lat;
  /* we need to find the *real* address by passing it through the
  TLB*/

  if(dtlb){
    result =
      cache_access(dtlb, Read, (md_addr_t)a,
   NULL, 1, sim_cycle,
   NULL, NULL, RID_DTLB_HACK, NULL);

    if(result.status == MEM_KNOWN) lat = result.lat;
    else lat = 10;

    /* right now, I'm not truly emulating a TLB miss.
     * Instead, I am just printing this and then I'll add
     * it to the total at the end
     */
    if(lat > 1)    printf("We have a TLB miss! lat = %d\n", lat);
```

```
    /*  else           printf("No TLB miss.\n");*/
  } else lat = 0;

  /*  return(lat); */
  return(0);
}

void
maui_insert_wq(md_addr_t ra, md_addr_t rb, int biu_type,
               unsigned int lat){
  maui_q_node *new;

  new = malloc(sizeof(maui_q));
  if(new == NULL)
     panic("AHHH!  Couldn't allocate memory for waiting q");

  new->d1 = ra;
  new->d2 = rb;
  new->type = biu_type;
  /* Lat is +1 because the update will be called after this call in
     the same cycle*/
  new->lat = lat + 1;
  new->next = NULL;

  proc_maui_q.size++;
  /*  printf("Size = %d\n", proc_maui_q.size);*/
```

```
  if(proc_maui_q.head == NULL){  /*this happens if this is the first
                                   on the queue*/
    proc_maui_q.head = new;
    proc_maui_q.tail = new;
  } else {
    /* this happens every other time */
    proc_maui_q.tail->next = new;
    proc_maui_q.tail = new;
  }
}
#endif
```

A.3 MAUI2.H

```
/*
 * maui2.h- simulates the MAUI addition to a memory controller/DRAM
 *          system. This is the header for the processor side of the
 *          simulated MAUI hardware.
 *
 * this file is being contributed to the SimpleScalar tool suite
 * written by Todd M. Austin as part of the Multiscalar Research
 * Project.
 *
 * Justin Teller
 * Dept of Electrical & Computer Engineering
 * University of Maryland, College Park
 * All Rights Reserved
```

```
 *
 * This software, should it be distributed, is distributed with
 * *ABSOLUTELY NO SUPPORT* and *NO WARRANTY*.  Permission is given to
 * modify this code as long as this notice is not removed.
 */

/*
 * April 2004
 */


#ifndef MAUI2_H
#define MAUI2_H

#include "maui.h"

void
add_maui_biu(unsigned long ra, unsigned long rb, int type);

int
try_biu_ins(int type, unsigned long d1, unsigned long d2);

void
maui_drain_wq();

void
maui_ex(int type, long rs, long rt, long imm);
```

```
#endif
```

A.4 MAUI2.C

```
/*
 * maui2.c- simulates the MAUI addition to a memory controller/DRAM
 *          system. This is actually for the processor side of the
 *          simulated MAUI hardware.  As opposed to maui.c, this file
 *          has access to the memory system, and so implements that
 *          part of the processor side of the MAUI architecture.
 *
 * this file is being contributed to the SimpleScalar tool suite
 * written by Todd M. Austin as part of the Multiscalar Research
 * Project.
 *
 * Justin Teller
 * Dept of Electrical & Computer Engineering
 * University of Maryland, College Park
 * All Rights Reserved
 *
 * This software, should it be distributed, is distributed with
 * *ABSOLUTELY NO SUPPORT* and *NO WARRANTY*.  Permission is given to
 * modify this code as long as this notice is not removed.
 */

/*
```

```
 * April 2004
 */


#ifndef MAUI2_C
#define MAUI2_C

#include "mem-system.h"
#include "maui.h"
#include "mase.h"
#include "maui2.h"

extern biu_t  biu;
extern tick_t sim_cycle;
extern maui_regs in_proc_maui;
extern maui_q proc_maui_q;
extern struct mem_t *mem;



void
execute_addi(unsigned long ra) {
  int i;
  int src, dst;
  for(i=0; i < in_proc_maui.size/sizeof(int); i++){
    mem_access(mem, Read, (in_proc_maui.ra + (i*sizeof(int))),
       &src, sizeof(int));
    dst = src + ra;
```

```
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(int))),
       &dst, sizeof(int));
  }

}

void
execute_add() {
  int i;
  int src1, src2, dst;

  for(i=0; i<(in_proc_maui.size/sizeof(int)); i++){
    /*    fprintf(stderr, "MAUI:\tResult should be: %d\n",
    (in_proc_maui.ra + in_proc_maui.rb));
    */
    mem_access(mem, Read,  (in_proc_maui.ra + (i*sizeof(int))),
       &src1, sizeof(int));
    mem_access(mem, Read,  (in_proc_maui.rb + (i*sizeof(int))),
       &src2, sizeof(int));
    dst = src1 + src2;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(int))),
       &dst,  sizeof(int));
  }
}

void
execute_mul(){
```

```
  int i;
  int src1, src2, dst;

  for(i=0; i<(in_proc_maui.size/sizeof(int)); i++){

    mem_access(mem, Read,  (in_proc_maui.ra + (i*sizeof(int))),
       &src1, sizeof(int));
    mem_access(mem, Read,  (in_proc_maui.rb + (i*sizeof(int))),
       &src2, sizeof(int));
    dst = src1 * src2;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(int))),
       &dst,  sizeof(int));
  }
}

void
execute_muli(unsigned long ra) {
  int i;
  int src, dst;
  for(i=0; i < in_proc_maui.size/sizeof(int); i++){
    mem_access(mem, Read, (in_proc_maui.ra + (i*sizeof(int))),
       &src, sizeof(int));
    dst = src * ra;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(int))),
       &dst, sizeof(int));
  }
}
```

```
void
execute_fadd() {
  int i;
  double src1, src2, dst;

  for(i=0; i < in_proc_maui.size/sizeof(int); i++){
    mem_access(mem, Read, (in_proc_maui.ra + (i*sizeof(double))),
       &src1, sizeof(double));
    mem_access(mem, Read, (in_proc_maui.rb + (i*sizeof(double))),
       &src2, sizeof(double));
    dst = src1 + src2;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(double))),
       &dst, sizeof(double));
  }
}

void
execute_faddi(double ra) {
  int i;
  double src1, dst;

  printf("Executing an faddi with immediate = %g\n", ra);

  for(i=0; i < in_proc_maui.size/sizeof(int); i++){
    mem_access(mem, Read, (in_proc_maui.ra + (i*sizeof(double))),
       &src1, sizeof(double));
```

```
    dst = src1 + ra;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(double))),
       &dst, sizeof(double));
  }
}

void
execute_fmul() {
  int i;
  double src1, src2, dst;

  for(i=0; i < in_proc_maui.size/sizeof(int); i++){
    mem_access(mem, Read, (in_proc_maui.ra + (i*sizeof(double))),
       &src1, sizeof(double));
    mem_access(mem, Read, (in_proc_maui.rb + (i*sizeof(double))),
       &src2, sizeof(double));
    dst = src1 * src2;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(double))),
       &dst, sizeof(double));
  }
}

void
execute_fmuli(double ra) {
  int i;
  double src1, dst;
```

```
  for(i=0; i < in_proc_maui.size/sizeof(int); i++){
    mem_access(mem, Read, (in_proc_maui.ra + (i*sizeof(double))),
       &src1, sizeof(double));

    dst = src1 * ra;
    mem_access(mem, Write, (in_proc_maui.rc + (i*sizeof(double))),
       &dst, sizeof(double));
  }
}


void
maui_ex(int type, long rs, long rt, long imm){

  /*printf("Here with type %d\n", type);*/

  switch(type){
  case MAUI_UNDEF:
    return;
    break;
  case MAUI_ADDI:
    execute_addi(rs);
    break;
  case MAUI_ADD:
    execute_add();
    break;
```

```
case MAUI_ST_RA:
  in_proc_maui.ra = rs;
  break;
case MAUI_ST_RA_RB:
  in_proc_maui.ra = rs;
  in_proc_maui.rb = rt;
  break;
case MAUI_ST_RB:
  in_proc_maui.rb = rs;
  break;
case MAUI_ST_RC:
  in_proc_maui.rc = rs;
  break;
case MAUI_ST_RC_I:
  in_proc_maui.rc = imm;
  break;
case MAUI_ST_SIZE:
  in_proc_maui.size = rs;
  break;
case MAUI_ST_SIZE_I:
  in_proc_maui.size = imm;
  break;
case  MAUI_MUL:
  execute_mul();
  break;
case MAUI_MULI:
  execute_muli(rs);
```

```
    break;
  case MAUI_FP_ADD:
    execute_fadd();
    break;
  case MAUI_FP_ADDI:
    execute_faddi(rs);
    break;
  case MAUI_FP_MUL:
    execute_fmul();
    break;
  case MAUI_FP_MULI:
    execute_fmuli(rs);
    break;
  default:
    panic("Incorrect MAUI code in execute!!");
    break;
  }
}

void
add_maui_biu(unsigned long ra, unsigned long rb, int type) {
  int sid;
  int biu_type;
  unsigned int lat1, lat2, lat3;
  /*printf("Maui instr at %lld\n", sim_cycle);*/

  lat1 = lat2 = lat3 = 0;
```

```
/* Possible fix:
 * Taken out the tlb checks because it was messing up the
 * order in which commands are sent to memory.
 * TODO:  Make the TLB checks happen and not mess up order.
 */



/* FIXED
 * The cache flush latencies don't matter, because they occur
 * before (on the same clock period) as the command is put on the
 * BIU.
 */
switch(type) {
case MAUI_ADDI:
  biu_type = MAUI_ADDI_BIU;

  break;
case MAUI_ADD:
  biu_type = MAUI_ADD_BIU;

  break;
case MAUI_ST_RA:
  biu_type = MAUI_LD_A_BIU;
  break;
case MAUI_ST_RA_RB:
  biu_type = MAUI_LD_AB_BIU;
  break;
```

```
case MAUI_ST_RB:
  biu_type = MAUI_LD_B_BIU;
  break;
case MAUI_ST_RC:
case MAUI_ST_RC_I:
  biu_type = MAUI_LD_C_BIU;
  break;
case MAUI_ST_SIZE:
case MAUI_ST_SIZE_I:
  biu_type = MAUI_LD_SIZE_BIU;
  break;

case MAUI_MUL:
  biu_type = MAUI_MUL_BIU;
  break;

case MAUI_MULI:
  biu_type = MAUI_MULI_BIU;
  break;

case MAUI_FP_ADD:
  biu_type = MAUI_FADD_BIU;
  break;

case MAUI_FP_ADDI:
  biu_type = MAUI_FADDI_BIU;
  break;
```

```
  case MAUI_FP_MUL:
    biu_type = MAUI_FMUL_BIU;
    break;
  case MAUI_FP_MULI:
    biu_type = MAUI_FADDI_BIU;
    break;
  default:
    break;

  }

  sid = find_free_biu_slot(INVALID);
  if(sid == INVALID){
    /* can't get a free slot.  Put this on the MAUI queue */
    /*printf("Putting something on the MAUI waiting queue.\n");*/
    maui_insert_wq(ra, rb, biu_type, 0);
  } else {
    fill_biu_slot(sid, sim_cycle, MAUI_RID, ra,
  biu_type, 0, NULL);
    /* and for this one, we have to also put in rb */
    biu.slot[sid].field2 = rb;
  }
}

int
try_biu_ins(int type, unsigned long d1, unsigned long d2){
  int sid;
```

```
  sid = find_free_biu_slot(0);
  if(sid != INVALID){

    fill_biu_slot(sid, sim_cycle, MAUI_RID, d1,
  type, 0, NULL);
    /* and for this one, we have to also put in rb */
    biu.slot[sid].field2 = d2;
  }

  return(sid);
}

void
maui_print_biu(){
  int i;

  printf("\nBIU STATUS\nStatus\tRid\tType\n");

  for(i=0;i<MAX_BUS_QUEUE_DEPTH;i++){
    printf("%d\t%d\t%d\t%d\n\n", biu.slot[i].status,
   biu.slot[i].rid, biu.slot[i].access_type,
            biu.slot[i].address);
  }

}
```

```
void
maui_drain_wq(){
  int sid;
  maui_q_node *ptr;

  if(proc_maui_q.size == 0)
    return;

  while((ptr = proc_maui_q.head) != NULL){
    sid = find_free_biu_slot(INVALID);
    if(sid == INVALID){

      /*printf("Trying to drain WQ, had to stop.\n");*/
      /*maui_print_biu();*/

      break;
    } else {
      fill_biu_slot(sid, sim_cycle, MAUI_RID, ptr->d1,
    ptr->type, 0, NULL);
      /* and for this one, we have to also put in rb */
      biu.slot[sid].field2 = ptr->d2;
      /*printf("Put one maui instr onto the BIU (wq)\n");*/
    }
    proc_maui_q.head = ptr->next;
    free(ptr);
    proc_maui_q.size--;
```

```
  }

  if(proc_maui_q.head == NULL){
    proc_maui_q.tail = NULL;
    proc_maui_q.size = 0;
  }
}

#endif
```

A.5 MEM_MAUI.H

```
/*
 * mem-maui.h - simulates the MAUI addition to a memory
 * controller/DRAM system.  This is the header for the memory system
 * side of the MAUI architecture.  The memory system side is where
 * the timing of the MAUI instructions are determined.
 * this file is being contributed to the SimpleScalar tool suite
 * written by Todd M. Austin as part of the Multiscalar Research
 * Project.
 *
 * Justin Teller
 * Dept of Electrical & Computer Engineering
 * University of Maryland, College Park
 * All Rights Reserved
 *
 * This software, should it be distributed, is distributed with
 * *ABSOLUTELY NO SUPPORT* and *NO WARRANTY*.  Permission is given to
```

```
 * modify this code as long as this notice is not removed.
 */

/*
 * April 2004
 */

#ifndef MEM_MAUI_H
#define MEM_MAUI_H

/* The next few attributes are places that can change to
   behavior of the MAUI */
/* This is the number of outstanding reads that are allowed per
   register */
#define  MAUI_READ_MAX        1
#define  MAUI_ISSUE_PER_CYCLE 3

/* This is the integer add latency, or the number of mem system cycles
   it takes the MAUI to perform an add */
#define  MAUI_ADD_LAT    1
#define  MAUI_MUL_LAT    2  /* integer multiplication latency */
#define  MAUI_FADD_LAT   2  /* floating point addition latency */
#define  MAUI_FMUL_LAT   4  /* floating point mult. latency */

/* this is the size of the cache that the MAUI uses,
   in wordline bytes */
#define  MAUI_CACHE_SIZE      10  /* ex: 100 = 3200 bytes */
```

```
/* These are definitions of the maui commands */
#define    MEM_MAUI_ADD   1
#define    MEM_MAUI_ADDI  2
#define    MEM_MAUI_MUL   3
#define    MEM_MAUI_MULI  4
#define    MEM_MAUI_FADD  5
#define    MEM_MAUI_FADDI 6
#define    MEM_MAUI_FMUL  7
#define    MEM_MAUI_FMULI 8



#define    M_WORDLINE_SIZE 32
#define    M_INTEGER_SIZE  4
#define    M_DOUBLE_SIZE   8
#define    M_FLOAT_SIZE    4

#define    MAUI_RA_SID     1000
#define    MAUI_RB_SID     1001
#define    MAUI_RC_SID     1002

/* These are the statistic types */
#define    M_LOCK          1
#define    M_CACHE         2
```

```
typedef struct _wqnode_t {
  int type;
  unsigned long d1, d2;
  struct _wqnode_t *next;
} wqnode_t;


typedef struct _maui_arch_wq {
  wqnode_t *head, *tail;
  int size;
} maui_arch_wq_t;



typedef struct _maui_arch {
  /* these are registers that hold the addresses */
  unsigned long ra, rb, rc, size;
  /* this is the number of finished writes */
  unsigned long finished;
  /* this is the queue of waiting MAUI instructions */
  maui_arch_wq_t waiting;
  /*
    cmd holds the type of command we're doing right now
     (ADD, ADDI, etc)
   */
  int cmd;
  /*
```

```
    this is a boolean telling us of the maui is occupied
    with a command
  */
  int busy;
} maui_arch_t;

/* this is cache that the maui has.  In here, we only need to keep
   track of addresses */
typedef struct _maui_cache {
  unsigned int addresses[MAUI_CACHE_SIZE];
  int next;
} maui_cache_t;

void
maui_biu_address(int next_slot_id);

void
mem_maui_init();

void
sink_maui_instr(int biu_sid);

void
update_mem_maui();

void
maui_mem_trans(int sid, int trans_type, unsigned int address);
```

```
void
add_to_mem_wq(int biu_sid);

int
maui_add_transaction(tick_t now, int transaction_type, int slot_id);

int
maui_lock_chk(int next_slot_id);

int
maui_in_cache_slot(int sid);

int
maui_in_cache(unsigned int address);

void
maui_add_cache(unsigned int address);

int
maui_finished();

void
maui_stat(int type);

void
print_maui_stat();
```

```
#endif
```

A.6 MEM_MAUI.C

```
/*
 * mem-maui.c - simulates the MAUI addition to a memory
 * controller/DRAM system.  This is the implementation of the memory
 * system side of the MAUI architecture.  This is where the timing of
 * MAUI operations are determined.
 * this file is being contributed to the SimpleScalar tool suite
 * written by Todd M. Austin as part of the Multiscalar Research
 * Project.
 *
 * Justin Teller
 * Dept of Electrical & Computer Engineering
 * University of Maryland, College Park
 * All Rights Reserved
 *
 * This software, should it be distributed, is distributed with
 * *ABSOLUTELY NO SUPPORT* and *NO WARRANTY*.  Permission is given to
 * modify this code as long as this notice is not removed.
 */

/*
 * April 2004
 */
```

```
#include "mem-system.h"
#include "mem-maui.h"

extern biu_t biu;
extern dram_system_t dram_system;

maui_arch_t mem_maui;

maui_cache_t maui_cache;



/* these variables are used to keep track of the state of the
   oustanding MAUI instrs */
static int sent_ra = 0;
static int sent_rb = 0;
static int recieved_ra    = 0;
static int recieved_rb    = 0;

static int sent_rc = 0;
static int recieved_rc = 0;
static int lat_remain = -1;
static long long int last_time;

void
mem_maui_init(){
  int i;
```

```
  mem_maui.ra =  mem_maui.rb = mem_maui.rc = mem_maui.size = 0;
  mem_maui.waiting.size = 0;
  mem_maui.waiting.head = mem_maui.waiting.tail = NULL;

  mem_maui.busy = 0;

  for(i=0;i<MAUI_CACHE_SIZE;i++){
    maui_cache.addresses[i] = 0;
  }

  maui_cache.next = 0;
  sent_ra = sent_rb = sent_rc = 0;
  recieved_ra = recieved_rb = recieved_rc = 0;

}

void
sink_maui_instr(int biu_sid){
  unsigned int d1, d2;

  d2 = biu.slot[biu_sid].field2;
  d1 = biu.slot[biu_sid].address;

  /* we want to make sure that we sink the instr only if
     the maui isn't already busy and if there aren't any
     instrs already waiting.  */
  if((mem_maui.busy == 0) && (mem_maui.waiting.size == 0)){
```

```
/*    printf("We're in here, and we should be.\n"); */
switch(biu.slot[biu_sid].access_type){
case MAUI_LD_A_BIU:
  mem_maui.ra = d1;
  printf("Ra = %d\n", d1);
  break;
case MAUI_LD_B_BIU:
  mem_maui.rb = d1;
  break;
case MAUI_LD_AB_BIU:
  mem_maui.ra = d1;
  mem_maui.rb = d2;
  break;
case MAUI_LD_C_BIU:
  mem_maui.rc = d1;
  /*printf("Rc = %d\n", d1);*/
  break;
case MAUI_LD_SIZE_BIU:
  mem_maui.size = d1;
  break;
case MAUI_ADD_BIU:
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_ADD;
  sent_ra = sent_rb = sent_rc = 0;
  printf("We're starting a MAUI ADD instr!, %f\n",
 dram_system.current_dram_time *
 dram_system.config.cpu2mem_clock_ratio);
```

```
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_ADDI_BIU:
  mem_maui.busy = 1;
  /* no need to actually pass this data */
  /* because all of this is just for timing */
  /* mem_maui.imm_v = d1; */
  mem_maui.cmd = MEM_MAUI_ADDI;
  sent_ra = sent_rb = sent_rc = 0;
  printf("Starting a MAUI ADDI instr!, %f\n",
 dram_system.current_dram_time *
 dram_system.config.cpu2mem_clock_ratio);
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_MUL_BIU:
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_MUL;
  sent_ra = sent_rb = sent_rc = 0;
  printf("Starting a MUL instr\n");
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_MULI_BIU:
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_MULI;
  sent_ra = sent_rb = sent_rc = 0;
  printf("Starting a MULI instr!, %f\n",
```

```
 dram_system.current_dram_time *
 dram_system.config.cpu2mem_clock_ratio);
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_FADD_BIU:
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_FADD;
  sent_ra = sent_rb = sent_rc = 0;
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_FADDI_BIU:
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_FADDI;
  sent_ra = sent_rb = sent_rc = 0;
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_FMUL_BIU :
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_FMUL;
  sent_ra = sent_rb = sent_rc = 0;
  if(mem_maui.size == 0) mem_maui.busy = 0;
  break;
case MAUI_FMULI_BIU:
  mem_maui.busy = 1;
  mem_maui.cmd = MEM_MAUI_FMULI;
  sent_ra = sent_rb = sent_rc = 0;
  if(mem_maui.size == 0) mem_maui.busy = 0;
```

```
      break;
    default:
      panic("Uh oh.  Somewhow an invalid MAUI command made it to
            \"sinking\"");
      break;
    }
  }
  else add_to_mem_wq(biu_sid);
  release_biu_slot(biu_sid);
}

void
print_status_mem_maui(){
  printf("Status of the MAUI in memory:\n");
  printf("RA\t\tRB\t\tRC\t\tSize\n");
  printf("%ld\t%ld\t%ld\t%ld\n", mem_maui.ra, mem_maui.rb,
                                 mem_maui.rc, mem_maui.size);
  printf("-----------------------------------------------\n");
  printf("\tSentRA\tSentRb\tSentRc\n");
  printf("\t%d\t%d\t%d\n", sent_ra, sent_rb, sent_rc);
  printf("\trecievedRA\trecievedRB\n");
  printf("\t%d\t\t%d\n\n\n", recieved_ra, recieved_rb);
}

/* this function sinks maui instrs one at a time
   until we hit one that makes the maui busy   */
void
```

```
maui_sink_waiting(){
  wqnode_t *tmp;
  while((mem_maui.busy == 0) && (mem_maui.waiting.size > 0)){

    if(mem_maui.waiting.head == NULL)
      panic("The in mem maui waiting queue is messed up");

    switch(mem_maui.waiting.head->type){
    case MAUI_LD_A_BIU:
      mem_maui.ra = mem_maui.waiting.head->d1;
      break;
    case MAUI_LD_B_BIU:
      mem_maui.rb = mem_maui.waiting.head->d1;
      break;
    case MAUI_LD_AB_BIU:
      mem_maui.ra = mem_maui.waiting.head->d1;
      mem_maui.rb = mem_maui.waiting.head->d2;
      break;
    case MAUI_LD_C_BIU:
      mem_maui.rc = mem_maui.waiting.head->d1;
      break;
    case MAUI_LD_SIZE_BIU:
      mem_maui.size = mem_maui.waiting.head->d1;
      break;
    case MAUI_ADD_BIU:
      mem_maui.busy = 1;
      mem_maui.cmd = MEM_MAUI_ADD;
```

```
      sent_ra = sent_rb = sent_rc = 0;
      printf("We're starting a MAUI ADD instr! (wq), %f\n",
dram_system.current_dram_time *
dram_system.config.cpu2mem_clock_ratio);
      if(mem_maui.size == 0) mem_maui.busy = 0;
      break;
    case MAUI_ADDI_BIU:
      mem_maui.busy = 1;
      /* no need to actually pass this data */
      /* because all of this is just for timing */
      /* mem_maui.imm_v = d1; */
      mem_maui.cmd = MEM_MAUI_ADDI;
      sent_ra = sent_rc = 0;
      printf("We're starting a MAUI ADDI instr! (wq), %f\n",
dram_system.current_dram_time *
dram_system.config.cpu2mem_clock_ratio);
      if(mem_maui.size == 0) mem_maui.busy = 0;
      break;
    case MAUI_MUL_BIU:
      mem_maui.busy = 1;
      mem_maui.cmd = MEM_MAUI_MUL;
      sent_ra = sent_rb = sent_rc = 0;
      printf("We're starting a MAUI MUL instr! (wq), %f\n",
dram_system.current_dram_time *
dram_system.config.cpu2mem_clock_ratio);
      if(mem_maui.size == 0) mem_maui.busy = 0;
      break;
```

```
   case MAUI_MULI_BIU:
     mem_maui.busy = 1;
     mem_maui.cmd = MEM_MAUI_MULI;
     sent_ra = sent_rb = sent_rc = 0;
     printf("We're starting a MAUI MULI instr! (wq), %f\n",
dram_system.current_dram_time *
dram_system.config.cpu2mem_clock_ratio);
     if(mem_maui.size == 0) mem_maui.busy = 0;
     break;
   case MAUI_FADD_BIU:
     mem_maui.busy = 1;
     mem_maui.cmd = MEM_MAUI_FADD;
     sent_ra = sent_rb = sent_rc = 0;
     if(mem_maui.size == 0) mem_maui.busy = 0;
     break;
   case MAUI_FADDI_BIU:
     mem_maui.busy = 1;
     mem_maui.cmd = MEM_MAUI_FADDI;
     sent_ra = sent_rb = sent_rc = 0;
     if(mem_maui.size == 0) mem_maui.busy = 0;
     break;
   case MAUI_FMUL_BIU :
     mem_maui.busy = 1;
     mem_maui.cmd = MEM_MAUI_FMUL;
     sent_ra = sent_rb = sent_rc = 0;
     if(mem_maui.size == 0) mem_maui.busy = 0;
     break;
```

```
    case MAUI_FMULI_BIU:
      mem_maui.busy = 1;
      mem_maui.cmd = MEM_MAUI_FMULI;
      sent_ra = sent_rb = sent_rc = 0;
      if(mem_maui.size == 0) mem_maui.busy = 0;
      break;

    default:
      panic("Uh oh.  Somewhow an invalid MAUI command made it to
             the wq");
      break;
    }

    tmp = mem_maui.waiting.head->next;
    free(mem_maui.waiting.head);
    mem_maui.waiting.head = tmp;
    mem_maui.waiting.size--;
  }
}

/****************************************
 * This function is called once every DRAM
 * cycle.  It puts reads and writes on the transaction
 * queue, and takes the responses off.
 ****************************************/
void
update_mem_maui(){
```

```
  int t_id;
  int num_trans;
  int old;

  /* If the maui isn't currently busy, there may be a maui instr
     waiting on the queue */
  if(mem_maui.busy == 0) {
    if(mem_maui.waiting.size > 0)
      maui_sink_waiting();
    return;
  }

  num_trans = 0;
  while(num_trans < MAUI_ISSUE_PER_CYCLE){
    /* printf("Here\n");*/
    /*    if(dram_system.current_dram_time > 500000)
  print_status_mem_maui();*/
    /* if it's not busy, there's nothing to do */
    if(mem_maui.busy == 0)   return;

    old = num_trans;
    switch(mem_maui.cmd){
    case  MEM_MAUI_ADD:

      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
```

```
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }

      if((sent_rc < recieved_rb) && (sent_rc < recieved_ra)){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_ADD_LAT - 1; /* minus 1 because it takes
    one cycle to get here
    after recieving something
                                              from ra or rb */
  /* check to make sure lat remain doesn't go negative */
  if(lat_remain < 0) lat_remain = 0;
}

if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
                                       MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }
```

```
  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI ADD instr!, %f\n",
   dram_system.current_dram_time*
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }

      /* Don't let ra get ahead of rb */
      if(sent_ra <= sent_rb){
if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
   (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
  /* since we're in here, we should issue a read for ra */
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_READ_COMMAND,
```

```
      MAUI_RA_SID);

  if(t_id == INVALID){
    /* do nothing, because transaction queue is full
    printf("Full Transaction Queue\n"); */
    return;
  }

  sent_ra++;
  num_trans++;
  /*   printf("Sent out a read for ra. %d\n", sent_ra);*/
}
      }
      if(((sent_rb - recieved_rb) <= MAUI_READ_MAX) &&
  (sent_rb * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for rb */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RB_SID);

if(t_id == INVALID){
  /* do nothing, because transaction queue is full
  printf("Full Transaction Queue\n"); */
  return;
}
sent_rb++;
num_trans++;
/*printf("Sent out a read for rb. %d\n", sent_rb);*/
```

```
      }

      break;
      /* Next, do the ADDI function */

    case MEM_MAUI_ADDI:

      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }

      /* this is very similar to the add, except we only fetch from
          ra */
      if(sent_rc < recieved_ra){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_ADD_LAT - 1; /* minus 1 because it takes
                                    one
    cycle to get here after
    recieving something from
    ra or rb */
  /* check to make sure lat remain doesn't go negative */
  if(lat_remain < 0) lat_remain = 0;
```

```
}
if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }

  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;
  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI ADDI instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
```

```
}
      }

      if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
 (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for ra */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RA_SID);
if(t_id == INVALID){
  /* do nothing, because transaction queue is full
     printf("Full Transaction Queue\n"); */
  return;
}

sent_ra++;
num_trans++;
/*   printf("Sent out a read for ra. %d\n", sent_ra);*/

      }
      break;

   case  MEM_MAUI_FMULI:
      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */

recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
```

```
      }

      if(sent_rc < recieved_ra){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_FMUL_LAT - 1; /* minus 1 because it takes
                                                  one
     cycle to get here after
     recieving something from
     ra or rb */
}
if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }

  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
```

```
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI FMULI instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }

      if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
 (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for ra */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RA_SID);

if(t_id == INVALID){
  /* do nothing, because transaction queue is full
     printf("Full Transaction Queue\n"); */
  return;
}
```

```
sent_ra++;
num_trans++;
/*   printf("Sent out a read for ra. %d\n", sent_ra);*/
      }

      break;

    case  MEM_MAUI_MULI:
      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }

      /* this is very similar to the add, except we only fetch
         from ra */
      if(sent_rc < recieved_ra){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_MUL_LAT - 1; /*minus 1 because it takes a
    cycle to get here after
    recieving something from
    ra or rb */
  /* check to make sure lat remain doesn't go negative */
```

```
  if(lat_remain < 0) lat_remain = 0;
}

if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }

  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI MULI instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
```

```
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }

      if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
 (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for ra */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RA_SID);


if(t_id == INVALID){
  /* do nothing, because transaction queue is full
     printf("Full Transaction Queue\n"); */
  return;
}

sent_ra++;
num_trans++;
/*   printf("Sent out a read for ra. %d\n", sent_ra);*/

      }

      break;
```

```
   case  MEM_MAUI_FADDI:
      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }

      if(sent_rc < recieved_ra){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_FADD_LAT - 1; /*minus 1 because it takes
    a cycle to get here after
    recieving something from
    ra or rb */
}
if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }
```

```
  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI FADDI instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }



      if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
 (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for ra */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RA_SID);
```

```
if(t_id == INVALID){
  /* do nothing, because transaction queue is full
     printf("Full Transaction Queue\n"); */
  return;
}

sent_ra++;
num_trans++;
/*   printf("Sent out a read for ra. %d\n", sent_ra);*/
      }
      break;

   case  MEM_MAUI_FMUL:

      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }

      if((sent_rc < recieved_rb) && (sent_rc < recieved_ra)){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
```

```
  lat_remain = MAUI_FMUL_LAT - 1; /* minus 1 because it takes
                                     a cycle to get here after
    recieving something from
    ra or rb */
}
          if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }

  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI FMUL instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
```

```
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }

      /* Don't let ra get ahead of rb */
      if(sent_ra <= sent_rb){
if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
   (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
  /* since we're in here, we should issue a read for ra */
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_READ_COMMAND,
      MAUI_RA_SID);


  if(t_id == INVALID){
    /* do nothing, because transaction queue is full
    printf("Full Transaction Queue\n"); */
    return;
  }
  sent_ra++;
  num_trans++;
  /*   printf("Sent out a read for ra. %d\n", sent_ra);*/
```

```
}
      }
      if(((sent_rb - recieved_rb) <= MAUI_READ_MAX) &&
  (sent_rb * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for rb */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RB_SID);

if(t_id == INVALID){
  /* do nothing, because transaction queue is full
  printf("Full Transaction Queue\n"); */
  return;
}

sent_rb++;
num_trans++;
/*printf("Sent out a read for rb. %d\n", sent_rb);*/
      }

      break;
    case  MEM_MAUI_MUL:

      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }
```

```
      if((sent_rc < recieved_rb) && (sent_rc < recieved_ra)){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_MUL_LAT - 1; /* minus 1 because it takes
    a cycle to get here after
    recieving something from
    ra or rb */
}

if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
  }

  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
```

```
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI MUL instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }

      /* Don't let ra get ahead of rb */
      if(sent_ra <= sent_rb){
if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
   (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
  /* since we're in here, we should issue a read for ra */
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_READ_COMMAND,
      MAUI_RA_SID);

  if(t_id == INVALID){
    /* do nothing, because transaction queue is full
    printf("Full Transaction Queue\n"); */
```

```
    return;
  }

  sent_ra++;
  num_trans++;
  /*   printf("Sent out a read for ra. %d\n", sent_ra);*/

}
      }
      if(((sent_rb - recieved_rb) <= MAUI_READ_MAX) &&
  (sent_rb * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for rb */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RB_SID);

if(t_id == INVALID){
  /* do nothing, because transaction queue is full
  printf("Full Transaction Queue\n"); */
  return;
}

sent_rb++;
num_trans++;
/*printf("Sent out a read for rb. %d\n", sent_rb);*/
      }

      break;
```

```
    case  MEM_MAUI_FADD:

      if((sent_ra == 0) && (sent_rb == 0)){
/* here we're just starting */
recieved_ra = recieved_rb = recieved_rc = 0;
mem_maui.finished = 0;
      }

      if((sent_rc < recieved_rb) && (sent_rc < recieved_ra)){
/* since we're in here, we should issue a write to rc */
/* but only after the add latency is finished */
if(lat_remain == -1){
  /* Here we know that we're just starting the add instr */
  lat_remain = MAUI_FADD_LAT - 1; /* minus 1 because it takes
    a cycle to get here after
    recieving something from
    ra or rb */
}

if(lat_remain == 0){
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_WRITE_COMMAND,
      MAUI_RC_SID);
  if(t_id == INVALID){
    /* do nothing, because transaction queue is full */
    /*printf("Full Transaction Queue\n"); */
    return;
```

```
  }

  maui_add_cache(mem_maui.rc + (sent_rc * M_WORDLINE_SIZE));
  sent_rc++;
  num_trans++;
  mem_maui.finished = mem_maui.finished + (M_WORDLINE_SIZE);
  /*printf("Sent a total of %d writes.\n", sent_rc);*/
  lat_remain = -1;

  if(mem_maui.finished >= mem_maui.size){
    printf("We finished the MAUI FADD instr!, %f\n",
   dram_system.current_dram_time *
   dram_system.config.cpu2mem_clock_ratio);
    mem_maui.busy = 0;
    return;
  }
} else if(last_time != dram_system.current_dram_time){
  last_time = dram_system.current_dram_time;
  lat_remain--;
}
      }

      /* Don't let ra get ahead of rb */
      if(sent_ra <= sent_rb){
if(((sent_ra - recieved_ra) <= MAUI_READ_MAX) &&
   (sent_ra * (M_WORDLINE_SIZE) <  mem_maui.size)){
  /* since we're in here, we should issue a read for ra */
```

```
  t_id = maui_add_transaction(dram_system.current_dram_time,
      MEMORY_READ_COMMAND,
      MAUI_RA_SID);


  if(t_id == INVALID){
    /* do nothing, because transaction queue is full
    printf("Full Transaction Queue\n"); */
    return;
  }
  sent_ra++;
  num_trans++;
  /*   printf("Sent out a read for ra. %d\n", sent_ra);*/

}
      }
      if(((sent_rb - recieved_rb) <= MAUI_READ_MAX) &&
  (sent_rb * (M_WORDLINE_SIZE) <  mem_maui.size)){
/* since we're in here, we should issue a read for rb */
t_id = maui_add_transaction(dram_system.current_dram_time,
    MEMORY_READ_COMMAND, MAUI_RB_SID);

if(t_id == INVALID){
  /* do nothing, because transaction queue is full
  printf("Full Transaction Queue\n"); */
  return;
}
```

```
sent_rb++;
num_trans++;
/*printf("Sent out a read for rb. %d\n", sent_rb);*/
      }

      break;
    default:
      panic("Tried to do the timing for an unimplemented MAUI instr");
    }

    if(num_trans == old){
      /* if we get here, it means that there is nothing that
         the maui can do */
   /* printf("we're exiting, because there's nothing we can do\n");*/
      return;
    }

  }
}

int
maui_finished()
{
#if 0
  if(dram_system.current_dram_time > 500000){
      printf("we're in here though ... \n");
```

```
    }
#endif
  if((sent_rc + 10 < recieved_rc) || (mem_maui.busy > 0))
    return(0);
  else
    return(1);
}


void
maui_mem_trans(int sid, int trans_type, unsigned int address){


  if(trans_type !=  MEMORY_WRITE_COMMAND){
    if(sid == MAUI_RA_SID){
      recieved_ra++;
      maui_add_cache(address);
      /*printf("Got a read from ra back! %d\n", recieved_ra);*/
    } else if(sid == MAUI_RB_SID){
      recieved_rb++;
      maui_add_cache(address);
      /*printf("Got a read from rb back! %d\n", recieved_rb);*/
    }
  } else if(sid == MAUI_RC_SID){
    /*printf("Got a write back from rc!%d\n", recieved_rc + 1);*/
    recieved_rc++;
  }
}
```

```
int
maui_add_transaction(tick_t now, int transaction_type, int slot_id){
  transaction_t   *this_t;
  unsigned int    address;
  addresses_t     this_a;
  int           transaction_id;



  switch(slot_id){
  case MAUI_RA_SID:
    address     = mem_maui.ra + (sent_ra * M_WORDLINE_SIZE);
    if(maui_in_cache(address)){
      /* Since we're in here, we don't have to look
         all the way out in the memory */
      recieved_ra++;
      /* just need to return anything except invalid */
      return(1);
    }
    break;
  case MAUI_RB_SID:
    address = mem_maui.rb + (sent_rb * M_WORDLINE_SIZE);
    if(maui_in_cache(address)){
      /* Since we're in here, we don't have to look all the way out
         to the memory */
      recieved_rb++;
      /*  Just need to return anything except invalid */
```

```
    return(1);
  }
  break;
case MAUI_RC_SID:
  address = mem_maui.rc + (sent_rc * M_WORDLINE_SIZE);
  break;
default:
  panic("MAUI tried to add a transaction from an invalid SID");
  return(INVALID);
  break;
}

if(dram_system.transaction_queue.transaction_count >=
         MAX_TRANSACTION_QUEUE_DEPTH){
  return INVALID;
}
transaction_id = dram_system.transaction_queue.transaction_count;

this_t = &(dram_system.transaction_queue.entry[transaction_id]);
/* in order queue. Grab next entry */
this_t->status                          = VALID;
this_t->arrival_time          = now;
this_t->completion_time       = 0;
this_t->transaction_type      = transaction_type;
this_t->slot_id                         = slot_id;
this_t->critical_word_ready   = FALSE;
this_t->critical_word_ready_time= 0;
```

```
  this_t->physical_address    = address;
  this_a.physical_address     = address;
  convert_address(dram_system.config.mapping_policy, &(this_a));
  this_t->chan_id                      = this_a.chan_id;

  this_t->next_c = transaction2commands(now,
                    dram_system.transaction_queue.transaction_count,
                    transaction_type,
                    &(this_a));
  dram_system.transaction_queue.transaction_count++;
  return(transaction_id);
}

void
add_to_mem_wq(int biu_sid){
  wqnode_t *new;

  new = malloc(sizeof(wqnode_t));
  if(new == NULL)
        panic("AHHH! Couldn't allocate memory for mem_maui queue!");

  new->d1 = biu.slot[biu_sid].address;
  new->d2 = biu.slot[biu_sid].field2;
  new->type = biu.slot[biu_sid].access_type;
  new->next = NULL;

  mem_maui.waiting.size++;
```

```
  /* this first one takes care of the case
     where this will be the first on the queue */
  if(mem_maui.waiting.head == NULL){
    mem_maui.waiting.head = new;
    mem_maui.waiting.tail = new;
  } else {
    mem_maui.waiting.tail->next = new;
    mem_maui.waiting.tail = new;
  }
}



/* This function checks those instructions which are on the
   waiting queue to
   see if they conflict with the new address */

int
mlock_chk_waiting(int next_slot_id){
  unsigned long loc_ra, loc_rb, loc_rc, loc_size;

  int loc_busy;
  wqnode_t *tmp;

  /* Start out with current values */
  loc_ra = mem_maui.ra;
  loc_rb = mem_maui.rb;
```

```
loc_rc = mem_maui.rc;
loc_size = mem_maui.size;
loc_busy = 0;

tmp = mem_maui.waiting.head;
/* Here we traverse the list of waiting MAUI instrs, updating the
   local ra, rb, etc
   and when we get an "action" instruction like ADD, ADDI, etc, then
   check the lock.
   if there is a violation, return 0.  If we get to the end of the
   function, that
   means that the next_slot_id biu slot does not conflict with
   what's waiting in the
   waiting queue, so we return a 1   */
while(tmp != NULL){
  switch(tmp->type){
  case MAUI_LD_A_BIU:
    loc_ra = tmp->d1;
    break;
  case MAUI_LD_B_BIU:
    loc_rb = tmp->d1;
    break;
  case MAUI_LD_AB_BIU:
    loc_ra = tmp->d1;
    loc_rb = tmp->d2;
    break;
  case MAUI_LD_C_BIU:
```

```
      loc_rc = tmp->d1;
      break;
    case MAUI_LD_SIZE_BIU:
      loc_size = tmp->d1;
      break;

      /* The following four cases have the same thing happen*/
      /* Notice that this is very similar to the main lock check
         function */
    case MAUI_ADD_BIU:
    case MAUI_MUL_BIU:
    case MAUI_FADD_BIU:
    case MAUI_FMUL_BIU:
      if((biu.slot[next_slot_id].address >= loc_rb) &&
 (biu.slot[next_slot_id].address <= loc_rb + loc_size) &&
 (biu.slot[next_slot_id].access_type == MEMORY_WRITE_COMMAND)){
/* since we're in here, it means that the address falls
   within RB which hasn't
   been read yet, and the command is a write.  This violates
   the "read" lock
   placed on RB for any of the commands which look at both
   ra and rb
 */
 return(0);
      }

      /* these cases all fall through to the next block of code
```

```
 as do the next four cases  */
    case MAUI_ADDI_BIU:
    case MAUI_MULI_BIU:
    case MAUI_FADDI_BIU:
    case MAUI_FMULI_BIU:
      if((biu.slot[next_slot_id].address >= loc_ra) &&
 (biu.slot[next_slot_id].address <= loc_ra + loc_size) &&
 (biu.slot[next_slot_id].access_type ==
                        MEMORY_WRITE_COMMAND)){
/* since we're in here, it means that the address falls in
   RA which hasn't been
   read yet, and the command is a write.  This violates the
   read lock placed
   on RA for the any command. */
return(0);
      }
      if((biu.slot[next_slot_id].address >= loc_rc) &&
 (biu.slot[next_slot_id].address <= loc_rc + loc_size)){
/* since we're in here, it means that the address falls in RC
   which hasn't been
   written to yet.  ANY command (read or write) violates this
   write lock placed
   on RC for the any command.  */

return(0);
      }
      break;
```

```
    default:
      panic("In waiting queue lock chk, major problems");
      break;
    }
    tmp = tmp->next;
  }
  return(1);
}

int
maui_lock_chk(int next_slot_id){

  if(mem_maui.busy == 0 && mem_maui.waiting.size == 0){
    return(1);
  } else if (mem_maui.busy == 1){

    switch(mem_maui.cmd) {

    case MEM_MAUI_ADD:
    case MEM_MAUI_MUL:
    case MEM_MAUI_FMUL:
    case MEM_MAUI_FADD:
      if((biu.slot[next_slot_id].address >= mem_maui.rb + (
          sent_rb * M_WORDLINE_SIZE)) &&
 (biu.slot[next_slot_id].address <= mem_maui.rb +
  mem_maui.size) &&
 (biu.slot[next_slot_id].access_type ==
```

```
  MEMORY_WRITE_COMMAND)){
/* since we're in here, it means that the address falls
   within RB which hasn't
   been read yet, and the command is a write.  This violates
   the "read" lock
   placed on RB for any of the commands which look at both
   ra and rb
 */
/*printf("locked on rb\n");*/
return(0);
      }

      /* The above look at A and C too, so it just falls through
         into this case */
    default:
      if((biu.slot[next_slot_id].address >= mem_maui.ra +
         (sent_ra * M_WORDLINE_SIZE)) &&
 (biu.slot[next_slot_id].address <= mem_maui.ra +
          mem_maui.size) &&
 (biu.slot[next_slot_id].access_type ==
  MEMORY_WRITE_COMMAND)){
/* since we're in here, it means that the address falls in RA
   which hasn't been
   read yet, and the command is a write.  This violates the
   read lock placed
   on RA for the any command. */
return(0);
```

```
      }
      if((biu.slot[next_slot_id].address >= mem_maui.rc + sent_rc *
          M_WORDLINE_SIZE) &&
  (biu.slot[next_slot_id].address <= mem_maui.rc +
  mem_maui.size)){
  /* since we're in here, it means that the address falls in
     RC which hasn't been
     written to yet.  ANY command (read or write) violates
     this write lock placed on RC for the any command.  */
/*printf("locked on rc\n");*/
return(0);
      }
      break;
    }
  }
  /* now we have to check the waiting queue to see if "future" this
     address conflicts with
     any of those */

  return(mlock_chk_waiting(next_slot_id));

}

int
maui_in_cache_slot(int sid){
  return(maui_in_cache(biu.slot[sid].address));
}
```

```
int
maui_in_cache(unsigned int address){
  int i;

  /*  We're not using the cache, so we always return 0, for
      no match, however I do assume that the cache exists
      elsewhere in the code, so I don't want to rip it out
      entirely, so this is the workaround */
  return(0);

  for(i=0;i<MAUI_CACHE_SIZE;i++){
    if((maui_cache.addresses[i] <= address) &&
       ((maui_cache.addresses[i] + M_WORDLINE_SIZE) >= address)){
      /* then we have it in the cache! */
      /*printf("FOUND SOMETHING IN MAUI CACHE!!\n"); */
      maui_stat(M_CACHE);
      return(1);
    }
  }
  /* if we're here, there was no match */
  return(0);
}

void
maui_add_cache(unsigned int address){
```

```
  /* if it's already in there, there's nothing to do! */
  if(maui_in_cache(address) == 1)
    return;

  maui_cache.addresses[maui_cache.next] = address;
  maui_cache.next = (maui_cache.next + 1) % MAUI_CACHE_SIZE;
}

static long maui_lock_num = 0;
static long maui_cache_hit = 0;

void
maui_stat(int type){
  switch(type){
  case M_LOCK:
    maui_lock_num++;
    /*if(maui_lock_num > 500000 ){
      printf("We're in trouble\n");
      }*/

    break;
  case M_CACHE:
    maui_cache_hit++;
    break;
  default:
    printf("Warning, undefined maui stat!\n");
    break;
```

```
  }
}

void
print_maui_stat(){
  fprintf(stderr,"\n");
  fprintf(stderr,"Maui cache hits:\t%ld\n", maui_cache_hit);
  fprintf(stderr,"Maui lock hits:\t\t%ld\n",  maui_lock_num);
}

void
maui_biu_address(int next_slot_id){

  printf("Locked on address: %d\n", biu.slot[next_slot_id].address);
}
```

A.7 Excerpt from MASE-EXEC.C pertaining to the MAUI architecture

```
/* executes the instruction */
void
execute_inst(struct ROB_entry *re)
{
  md_inst_t inst = re->IR;
  enum md_opcode op = re->op;
  union val_union mem_value;
  union val_union reg_value;
```

```
  int annotate;



#ifdef TARGET_ARM
  union val_union psr_value;
  int setPSR = 0;


  psr_value.w = PSR;
#endif


  /* assume there is no branch */
  /* execution of syscalls are delayed until commit */
#define DECLARE_FAULT(FAULT) { re->fault = (FAULT); break; }
#define SYSCALL(INST) ;
  /* execute the instruction of non oracle mode */
  /* read_idep_list & set_odep_list */
  switch (op)
  {
#define DEFINST(OP,MSK,NAME,OPFORM,RES,FLAGS,O1,O2,O3,I1,I2,I3,I4)\
  case OP:  \
    SYMCAT(OP,_IMPL);  \
    break;
#define DEFUOP(OP,NAME,OPFORM,RES,CLASS,O1,O2,O3,I1,I2,I3,I4)   \
  case OP: \
    SYMCAT(OP,_IMPL); \
    break;
#define DEFLINK(OP,MSK,NAME,MASK,SHIFT)   \
```

```
  case OP:  \
     panic("attempted to execute a linking opcode");
#define CONNECT(OP)
#include "machine.def"
    default:
      fatal("Invalid operation");
  }

  /*  Here is where the MAUI instructions are detected and then
      placed on the ROB
      Added 2004, Teller
  */

  annotate = (int) (inst.a >> 16);

  if((annotate > 0) && ((op == ADDU) || (op == ADDIU))) {
    /* This is a MAUI command! */
    /* in this stage, we're not loading everything into the BIU
     * just yet.
     * We need to make sure that we're sending these things to the
     * memory
     * system in program order for this to work out right.  So,
     * here we're
     * just modifying the ROB entry for this instruction to make
     * sure the
     * commit stage knows that this instruction is a MAUI instruction,
     * and is also
```

```
 * able to figure out what the register values were when it was
 * "executed."
 */

/* load the operands into the reorder buffer entry */
re->maui_info.immed = IMM;
re->maui_info.rs = GPR(RS);
re->maui_info.rt = GPR(RT);

switch(annotate){
case 1:
  if(op != ADDU) panic("Improper use of MAUI add");
  /* the regular MAUI ADD instr takes no arguments */
  re->maui_info.type = MAUI_ADD;

  break;

case 2:
  if(op != ADDU) panic("Improper use of MAUI load");
  /* MAUI ST RA, RB */
  re->maui_info.type = MAUI_ST_RA_RB;

  break;

case 3: /* We're loading RB */
  if(op != ADDU) panic("Improper use of MAUI load");
  /* RB */
```

```
      re->maui_info.type = MAUI_ST_RB;
      break;

    case 4: /* We're loading RC with some value */
      if(op != ADDU) panic("Improper use of MAUI load");
      /* Loading it from the register value */
      re->maui_info.type = MAUI_ST_RC;

      break;
    case 5: /* We're loading the size register with some value */
      if(op == ADDIU){
/* Immediate */
re->maui_info.type = MAUI_ST_SIZE_I;

      }else if(op == ADDU){
/* Loading it from RS */
re->maui_info.type = MAUI_ST_SIZE;

      }
      break;
    case 6:
      if(op != ADDU)  panic("Inproper use of maui mul");
      re->maui_info.type = MAUI_MUL;

      break;
    case 7:
      if(op != ADDU)  panic("Inproper use of maui muli");
```

```
      re->maui_info.type = MAUI_MULI;


      break;
    case 8:
      if(op != ADDU)  panic("Inproper use of maui fadd");
      re->maui_info.type = MAUI_FP_ADD;
      break;
    case 9:
      if(op != ADDU)  panic("Inproper use of maui faddi");
      re->maui_info.type = MAUI_FP_ADDI;
      break;
    case 10:
      if(op != ADDU)  panic("Inproper use of maui fmul");
      re->maui_info.type = MAUI_FP_MUL;
      break;
    case 11:
      if(op != ADDU)  panic("Inproper use of maui fmuli");
      re->maui_info.type = MAUI_FP_MULI;
      break;


    case 12:
      if(op != ADDU)
panic("Improper use of maui addi");
      re->maui_info.type = MAUI_ADDI;
      break;
    case 13:
      if(op != ADDU) panic("Improper use of maui load");
```

```
      re->maui_info.type = MAUI_ST_RA;
      break;

    case 14:
      break;
    default:
      re->maui_info.type = MAUI_UNDEF;
      panic("Invalid annotate field for MAUI instruction.");
      break;
    }
    maui_ex(re->maui_info.type, re->maui_info.rs,
    re->maui_info.rt, re->maui_info.immed);


  } else   re->maui_info.type = MAUI_UNDEF;  /* added so that
             non-maui instrs aren't effected */

  /* End of added code for MAUI instructions
   *
   * The annotated instructions do go through the pipeline
   * and are executed as regular ADDU and ADDUI instructions.
   * However, their target register is (and should be)
   * $0, so it doesn't actually change the state of the
   * system $.
   */

#ifdef TARGET_ARM
```

```
  /* set PSR reg into dep_list */
  if (setPSR != 0)
    set_odep_list(re, DPSR, psr_value, vt_word);
#endif
}
```

A.8 Excerpt from MASE-COMMIT.C pertaining to the MAUI architecture

```
void
mase_commit(void)
{
  int i;                          /* loop traversal variable */
  int lat;                        /* latency of store */
  int events;                     /* summary of events */
  int is_write;                   /* set if instr is store */
  int lsq_chk_error;         /* set if checker occurs for LSQ entry */
  int n_committed = 0;            /* number of instrs committed */
  int dlite_made_check = FALSE;
  md_addr_t mem_addr = 0;         /* address of memory operation */
  int done = FALSE;
  md_inst_t inst;

  struct ROB_entry *re, *lsq;

  inst = MD_NOP_INST;
  re = lsq = NULL;
```

```
/*******************************************************
 * MAUI function call to make sure the MAUI waiting     *
 * queue is drained before we try to put any other      *
 * instructions onto the BIU.                           *
 *                                                      *
 * Added JT  4-2004
 *******************************************************/

maui_drain_wq();



/*  end MAUI code */
/* all values must be retired to the architected reg file in
   program order */
while (!done && n_committed < commit_width)
{
  /* default values */
  events = 0;
  done = TRUE;
  is_write = FALSE;
  lsq_chk_error = 0;

  /* retire the oldest: either the head of the ROB or LSQ */
  if (ROB_num > 0 && (LSQ_num == 0 || LSQ[LSQ_head].seq >
      ROB[ROB_head].seq))
  {
```

```
      /* retire ROB entry */
      re = &(ROB[ROB_head]);

      /* is ROB entry complete? */
      if (!re->completed) break;

      /*****************************************************
       * This the code to deal with MAUI instructions and  *
       * put the instruction into the BIU                  *
       *                                                   *
       * Added by Justin Teller, March 2004                *
       *****************************************************/

      /* check to make sure it is a MAUI instr ...*/

      if(re->maui_info.type != MAUI_UNDEF){

switch(re->maui_info.type) {
  /*the check TLB happens in the add_maui_biu func */
case MAUI_ADDI:
  add_maui_biu(re->maui_info.rs,0,MAUI_ADDI);
  break;
case MAUI_ADD:
  add_maui_biu(0,0,MAUI_ADD);
  break;
case MAUI_ST_RA:
  /*   maui_check_tlb(re->maui_info.immed);*/
```

```
  add_maui_biu(re->maui_info.rs,0,MAUI_ST_RA);
  break;
case MAUI_ST_RA_RB:
  /*   maui_check_tlb(re->maui_info.rs);
  maui_check_tlb(re->maui_info.rt); */
  add_maui_biu(re->maui_info.rs, re->maui_info.rt,
                      MAUI_ST_RA_RB);
  break;
case MAUI_ST_RB:
  /*   maui_check_tlb(re->maui_info.immed);*/
  add_maui_biu(re->maui_info.rs, 0, MAUI_ST_RB);
  break;
case MAUI_ST_RC:
  /*   maui_check_tlb(re->maui_info.rs);*/
  add_maui_biu(re->maui_info.rs, 0, MAUI_ST_RC);
  break;
case MAUI_ST_RC_I:
  /*   maui_check_tlb(re->maui_info.immed);*/
  add_maui_biu(re->maui_info.rs, 0, MAUI_ST_RC);
  break;
case MAUI_ST_SIZE:
  add_maui_biu(re->maui_info.rs, 0, MAUI_ST_SIZE);
  break;
case MAUI_ST_SIZE_I:
  add_maui_biu(re->maui_info.immed, 0, MAUI_ST_SIZE);
  break;
```

```
case MAUI_MUL:
  add_maui_biu(0,0,MAUI_MUL);
  break;
case MAUI_MULI:
  add_maui_biu(0,0,MAUI_MULI);
  break;
case MAUI_FP_ADD:
  add_maui_biu(0,0,MAUI_FP_ADD);
  break;
case MAUI_FP_ADDI:
  add_maui_biu(re->maui_info.rs,0,MAUI_FP_ADDI);
  break;
case MAUI_FP_MUL:
  add_maui_biu(0,0,MAUI_FP_MUL);
  break;
case MAUI_FP_MULI:
  add_maui_biu(re->maui_info.rs,0,MAUI_FP_MULI);

default:
  panic("Tried to execute an unimplemented MAUI operation!!");

}
      }

      /*************************************************
       * End of added code, Teller                     *
       *************************************************/
```

```
    /* skip ISQ entry of NOP */
    if (ISQ[ISQ_head].op == MD_NOP_OP)
    {
```

. . . with the remainder of the *mase_commit* function following.

A.9 Excerpt from MEM-INTERFACE.C pertaining to the MAUI architecture

```
...

void dram_update_system(tick_t now) {
  int sid,rid, access_type;
  unsigned int    address;
  int latency;             /* latency is in terms of CPU cycles */
  tick_t start_time;

  /* here we need to update the MAUI */
  /* Added by Justin Teller March 24, 2004 */
  if((bus_queue_status_check(now) == BUSY) || (maui_finished() == 0)){
    update_dram_system(now);
  }
  /* End of added/modified code */

   ...
```

A.10 Excerpt from MEM-SYSTEM.C pertaining to the MAUI architecture

```
...

#define MEMORY_ACCESS_TYPES_COUNT 6
                /* 5 different ways of accessing memory */
#define MEMORY_UNKNOWN_COMMAND 0
#define MEMORY_IFETCH_COMMAND 1
#define MEMORY_WRITE_COMMAND 2
#define MEMORY_READ_COMMAND 3
#define MEMORY_DTLB_FROM_COMMIT_IGNORE 4
#define MEMORY_PREFETCH 5
#define AUTO_REFRESH_TRANSACTION 6
#define AUTO_PRECHARGE_TRANSACTION 7
/* more ways with the maui! -Justin Teller */
#define MAUI_LD_A_BIU                   8
#define MAUI_LD_B_BIU                   9
#define MAUI_LD_AB_BIU                  10
#define MAUI_LD_C_BIU                   11
#define MAUI_LD_SIZE_BIU                12
#define MAUI_ADD_BIU                    13
#define MAUI_ADDI_BIU                   14
#define MAUI_MUL_BIU                    15
#define MAUI_MULI_BIU                   21
#define MAUI_FADD_BIU                   17
#define MAUI_FADDI_BIU                  18
#define MAUI_FMUL_BIU                   19
```

```
#define MAUI_FMULI_BIU                  20
/* end maui code JT */
   ...
```

A.11 Excerpts from MEM-DRAM.C pertaining to the MAUI architecture

Within the *update_base_dram* function ...

```
  dram_system.dram_controller[chan_id].rank[rank_id].
       bank[bank_id].last_command = PRECHARGE;
}
      }
    }
  }

  /*********************************************
   * MAUI Handling function                    *
   * Added March 2004, JT                      *
   *********************************************/
  update_mem_maui();
  /*********************************************
   * End added code for MAUI                   *
   *********************************************/

  while(dram_system.current_dram_time <= dram_stop_time){
     /* continue to simulate until time limit */
   ...
```

```
 /* If this is a refresh or auto precharge transaction,
 just ignore it. Else... */
 /* Begin added code for MAUI simulation */
  if(dram_system.transaction_queue.entry[0].slot_id >= 1000){
             /* this means that it's a MAUI instr */
    /* This updates the MAUI and lets it know that another
               transaction is finished */
maui_mem_trans(dram_system.transaction_queue.entry[0].slot_id,
   dram_system.transaction_queue.entry[0].transaction_type,
   dram_system.transaction_queue.entry[0].physical_address);
     } else {
  /* End of added code -- J. Teller March, 2004 */

   ...

  /* If command bus is idle, see if there is another request
   * in BIU that needs to be serviced.
   * We start by finding the request we want to service.
   * Specifically, we want the slot_id of the request
   * and either move it from VALID to SCHEDULED  or from
   * SCHEDULED to COMPLETED
   */

  /* the next few lines implement the MAUI pulling stuff
     off the BIU */
  /* Added JT, March 2004 */
  if(get_access_type(next_slot_id) >= MAUI_LD_A_BIU){
   /* This next line is taken care of by the sink func call */
```

```
    /* set_biu_slot_status(next_slot_id, COMPLETED); */
    sink_maui_instr(next_slot_id);

  } else {
    /* now, check to make sure that the maui hasn't locked
       that piece of memory */
    if(maui_lock_chk(next_slot_id) == 1){
      /* since it returned a 1, we're OK! */
      transaction_id =
      add_transaction(dram_system.current_dram_time,
                      get_access_type(next_slot_id),
      next_slot_id);
if(transaction_id != INVALID){
  set_biu_slot_status(next_slot_id, SCHEDULED);
  if(transaction_debug()){
    fprintf(stderr,"Starting New Transaction\n");
    print_transaction(dram_system.current_dram_time,
                      transaction_id);
  }
} else if (transaction_debug()){
  fprintf(stderr,"Transaction queue full.
       Cannot start new transaction.\n");
}
      }
    } else {
      /*printf("LOCKED!!\n");  */
      /*maui_biu_address(next_slot_id);*/
```

```
      maui_stat(M_LOCK);
    }
    /* if the MAUI has locked a piece of memory, do nothing */
  }
  /* end of added/modified code, JT */
}

   ...
```

Appendix B: Benchmarks

Appendix B contains both the MAUI optimized and unoptimized code for the *MAUI-one*, *MAUI-two*, and *Stream* benchmarks used to test the performance of the MAUI architecture.

B.1 The *MAUI-one* benchmark

B.1.1 Unoptimized version of *MAUI-one*

```
#include<stdio.h>
#define ARR_SIZE 100000

int main(int argc, char *argv[]){
  int a[ARR_SIZE], b[ARR_SIZE], c[ARR_SIZE];
  int i, size;

  if(argc == 1)
    size = 100;
  else
    size = atoi(argv[1]);

  /* First initialize */
  for(i=0;i<size;i++){
    a[i] = b[i] = i;
  }
```

```
  /* Now the compute */
  for(i=0;i<size;i++){
    c[i] = a[i] + b[i];
  }
}
```

B.1.2 MAUI optimized version of *MAUI-one*

```
#include<stdio.h>
#define ARR_SIZE 100000

int main(int argc, char *argv[]){
  int a[ARR_SIZE], b[ARR_SIZE], c[ARR_SIZE];
  int i, size, insize;

  if(argc == 1)
    insize = 100;
  else
    insize = atoi(argv[1]);

  size = insize*sizeof(int);
  /* First initialize */
  for(i=0;i<insize;i++){
    a[i] = b[i] = i;
  }

  __asm__ ("ADDU/15:0(2) \t$0,%0,%1 \t#MAUI load ra, rb\n\t"
```

```
            "ADDU/15:0(4) \t$0,%2,$0 \t#MAUI load rc\n\t"
            "ADDU/15:0(5) \t$0,%3,$0 \t#MAUI load size\n\t"
            "ADDU/15:0(1) \t$0,$0,$0 \t#MAUI add\n\t"

            : : "r"(a), "r"(b), "r"(c), "r"(size));
  /* putting in a read on rc to make sure that the simulation doesn't $
     end until the maui instruction is finished */
  i = c[insize - 1];

}
```

B.2 The *MAUI-two* benchmark

B.2.1 Unoptimized version of *MAUI-two*

```
#include<stdio.h>
#define ARR_SIZE 100000

int main(int argc, char *argv[]){
  int a[ARR_SIZE], b[ARR_SIZE], c[ARR_SIZE];
  int d[ARR_SIZE], e[ARR_SIZE], f[ARR_SIZE];
  int i, size;

  if(argc == 1)
    size = 100;
  else
    size = atoi(argv[1]);
```

```
  /* Now the compute */
  for(i=0;i<size;i++){
    c[i] = a[i] + b[i];

    f[i] = d[i] + e[i];
  }
}
```

B.2.2 MAUI optimized version of *MAUI-two*

```
#include<stdio.h>
#define ARR_SIZE 100000

int main(int argc, char *argv[]){
  int a[ARR_SIZE], b[ARR_SIZE], c[ARR_SIZE];
  int d[ARR_SIZE], e[ARR_SIZE], f[ARR_SIZE];

  int i, size, insize;

  if(argc == 1)
    insize = 100;
  else
    insize = atoi(argv[1]);

  size = insize*sizeof(int);
  __asm__ ("ADDU/15:0(2) \t$0,%0,%1 \t#MAUI load ra, rb\n\t"
           "ADDU/15:0(4) \t$0,%2,$0 \t#MAUI load rc\n\t"
           "ADDU/15:0(5) \t$0,%3,$0 \t#MAUI load size\n\t"
```

```
          "ADDU/15:0(1) \t$0,$0,$0 \t#MAUI add\n\t"

          : : "r"(d), "r"(e), "r"(f), "r"(size));

  for(i=0;i<insize;i++){
    c[i] = a[i] + b[i];
  }
  /* putting in a read on rc to make sure that the simulation doesn't$
     end until the maui instruction is finished */

  i = f[insize - 1];
}
```

B.3 The *Stream* benchmark

B.3.1 The unoptimized version of *Stream*

```
#include<stdio.h>
#define N        2000000
#define OFFSET   0
#define NTIMES   10

static int a[N + OFFSET];
static int b[N + OFFSET];
static int c[N + OFFSET];

int main(){
  int j,k;
```

```
int scalar;


scalar = 3;
printf("This uses %d Bytes of memory\n",
       (3*N * sizeof(int)));


for(j=0;j<N;j++){
  a[j] = 1;
  b[j] = 2;
  c[j] = 0;
}
for(k=0;k < NTIMES; k++){
  printf("Starting the %d time\n",k);
  for(j=0; j<N; j++){
    c[j] = a[j];
  }


  for(j=0; j<N; j++){
    b[j] = scalar * c[j];
  }


  for(j=0; j<N; j++){
    c[j] = a[j] + b[j];
  }


  for(j=0; j<N; j++){
    a[j] = b[j] + scalar*c[j];
```

```
    }
  }
}
```

B.3.2 The MAUI optimized version of *Stream*

```
#include<stdio.h>

#define N        2000000
#define OFFSET   0
#define NTIMES   10

static int a[N + OFFSET];
static int b[N + OFFSET];
static int c[N + OFFSET];

void maui_add(void *dst, void *src1, void *src2, int size);
void maui_mul_scalar(void *dst, void *src, int scalar, int size);
void maui_copy(void *dst, void *src, int size);

int main(){
  int j,k;
  int scalar;
  int size;
  int test;

  scalar = 3;
  size = N * sizeof(int);
```

```
printf("This uses %d Bytes of memory\n",
       (3*N * sizeof(int)));



for(j=0;j<N;j++){
  a[j] = 1;
  b[j] = 2;
  c[j] = 0;
}



for(k=0;k < NTIMES; k++){
  printf("Starting the %d time\n", k);

  maui_copy(c, a, size);

  maui_mul_scalar(b, c, scalar, size);

  maui_add(c, a, b, size);

  for(j=0;j<N;j++)
    a[j]=b[j]+scalar*c[j];

}
```

```
}

void maui_add(void *dst, void *src1, void *src2, int size){
  __asm__("ADDU/15:0(2) \t$0,%0,%1 \t#MAUI load ra, rb\n\t"
          "ADDU/15:0(4) \t$0,%2,$0 \t#MAUI load rc \n\t"
          "ADDU/15:0(5) \t$0,%3,$0 \t#MAUI load size\n\t"
          "ADDU/15:0(1) \t$0,$0,$0  \t#MAUI add\n\t"

          : : "r"(src1), "r"(src2), "r"(dst), "r"(size));
}

void maui_mul_scalar(void *dst, void *src, int scalar, int size){
  __asm__("ADDU/15:0(13) \t$0,%0,$0 \t#MAUI load ra\n\t"
          "ADDU/15:0(4) \t$0,%1,$0 \t#MAUI load rc \n\t"
          "ADDU/15:0(5) \t$0,%2,$0 \t#MAUI load size\n\t"
          "ADDU/15:0(7) \t$0,%3,$0  \t#MAUI addi\n\t"

          : : "r"(src), "r"(dst), "r"(size), "r"(scalar));
}

void maui_copy(void *dst, void *src, int size){
  __asm__("ADDU/15:0(13) \t$0,%0,$0 \t#MAUI load ra\n\t"
          "ADDU/15:0(4) \t$0,%1,$0 \t#MAUI load rc \n\t"
          "ADDU/15:0(5) \t$0,%2,$0 \t#MAUI load size\n\t"
          "ADDU/15:0(12) \t$0,$0,$0  \t#MAUI addi\n\t"

          : : "r"(src), "r"(dst), "r"(size));
```

```
}
```

Appendix C: All Simulation Results

Appendix C contains all the simulation results generated in testing the performance characteristics of the MAUI architecture.

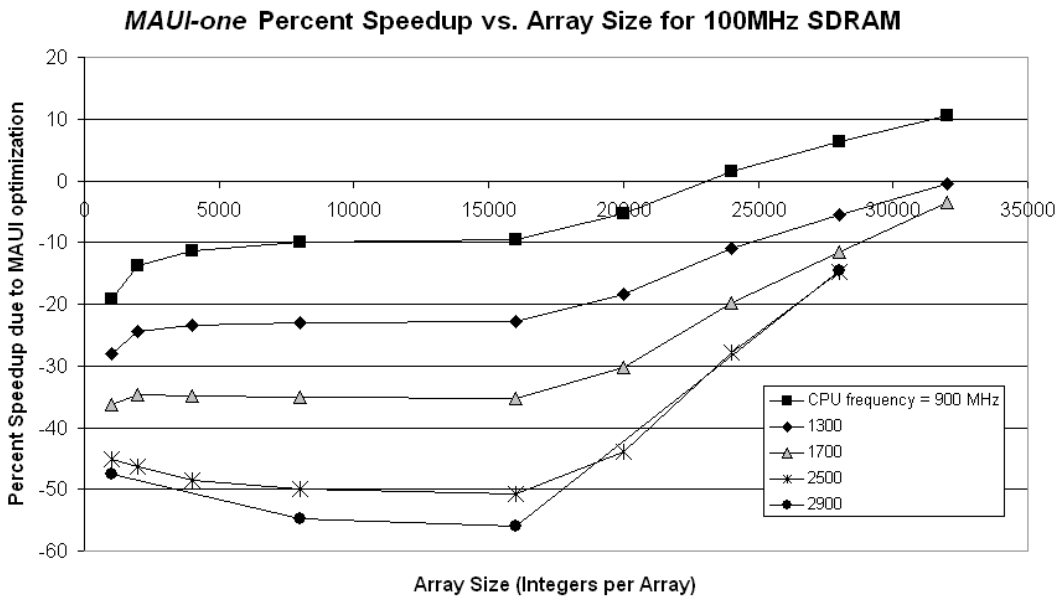

Figure C.1: Simulation results of *MAUI-one* when run with a 100 MHz *SDRAM* memory system.

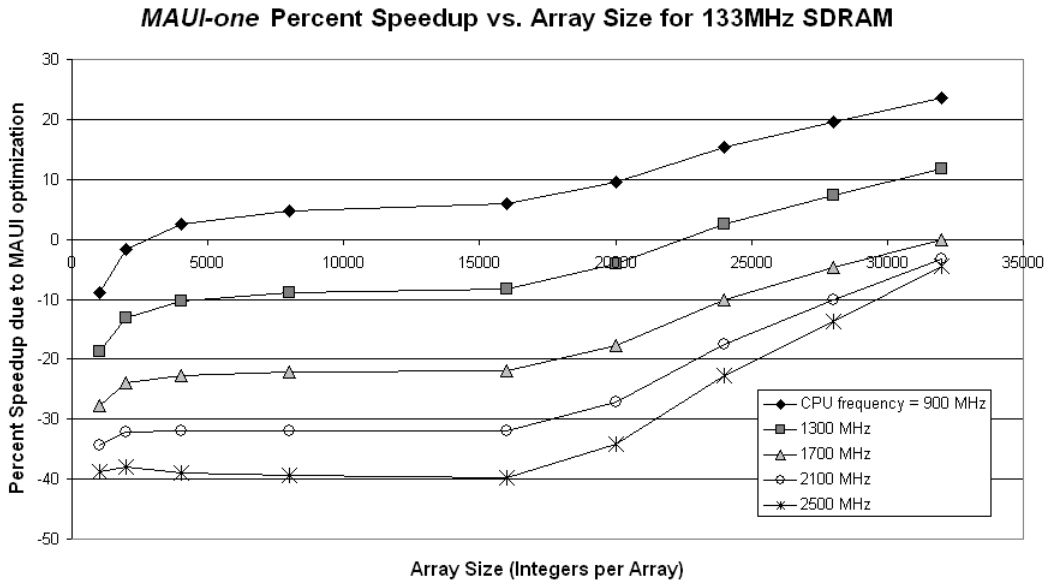

Figure C.2: Simulation results of *MAUI-one* when run with a 133 MHz *SDRAM* memory system.

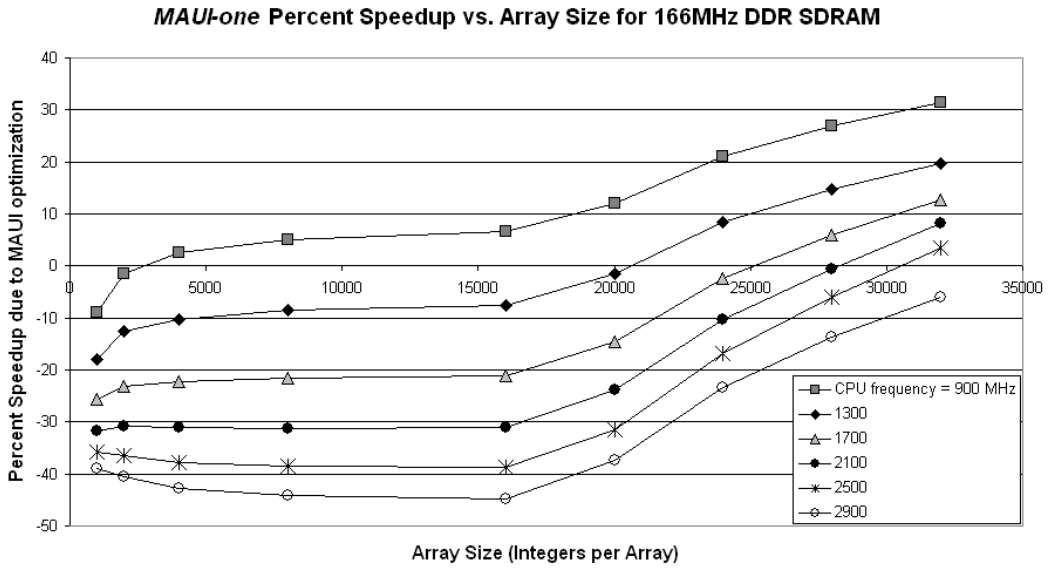

Figure C.3: Simulation results of *MAUI-one* when run with a 166 MHz *DDR-SDRAM* memory system.

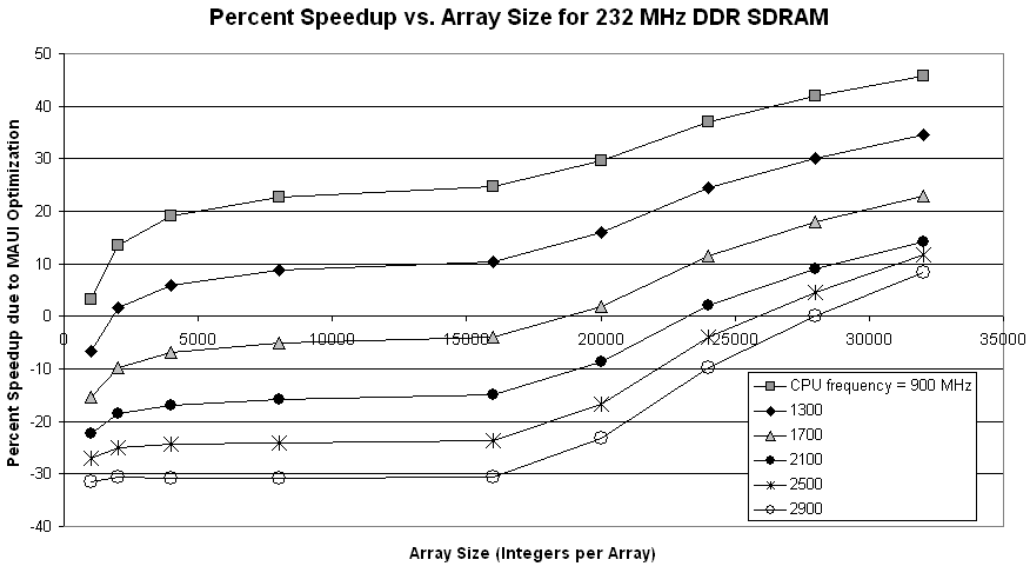

Figure C.4: Simulation results of *MAUI-one* when run with a 232 MHz *DDR-SDRAM* memory system.

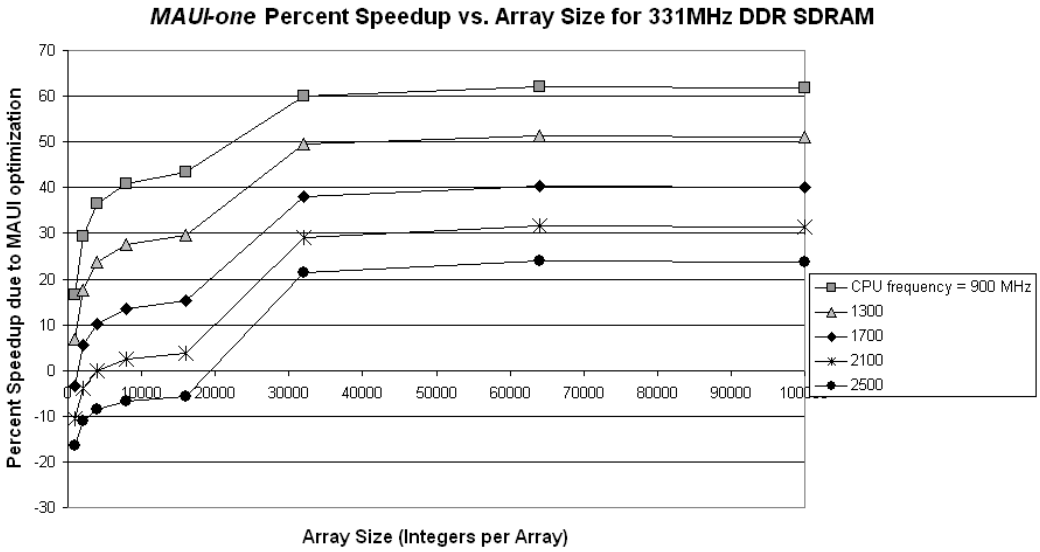

Figure C.5: Simulation results of *MAUI-one* when run with a 331 MHz *DDR-SDRAM* memory system.

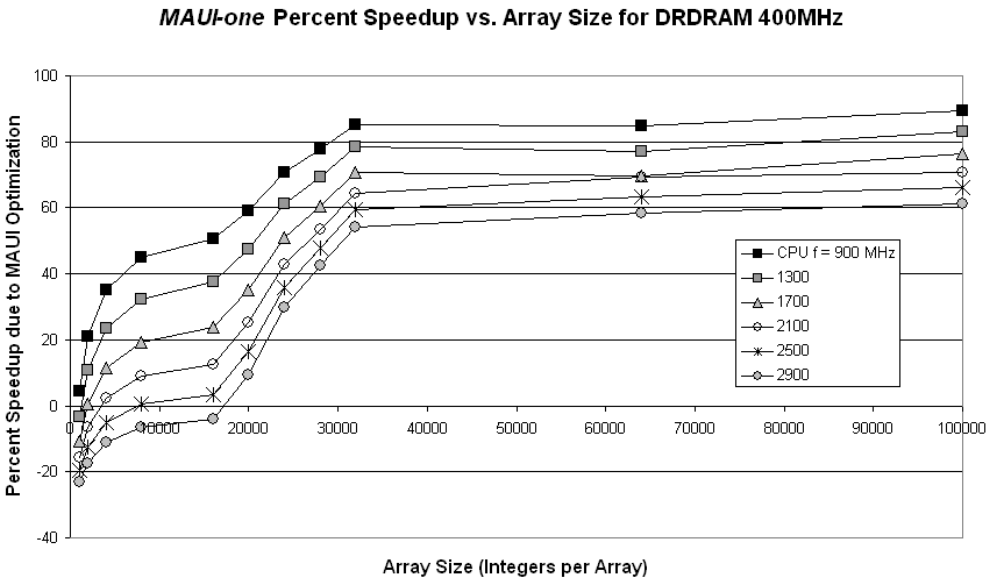

Figure C.6: Simulation results of *MAUI-one* when run with a 400 MHz *DRDRAM* memory system.

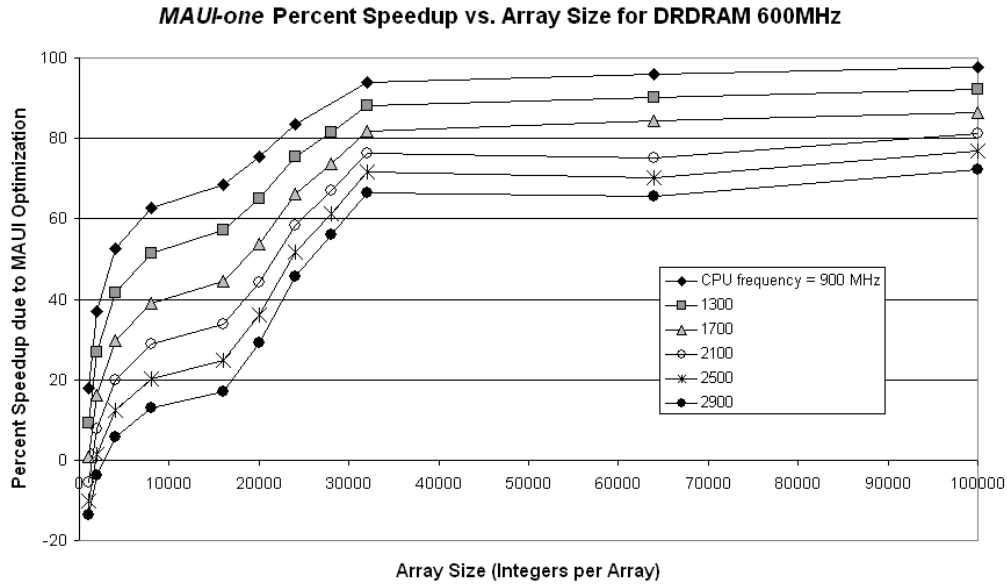

Figure C.7: Simulation results of *MAUI-one* when run with a 600 MHz *DRDRAM* memory system.

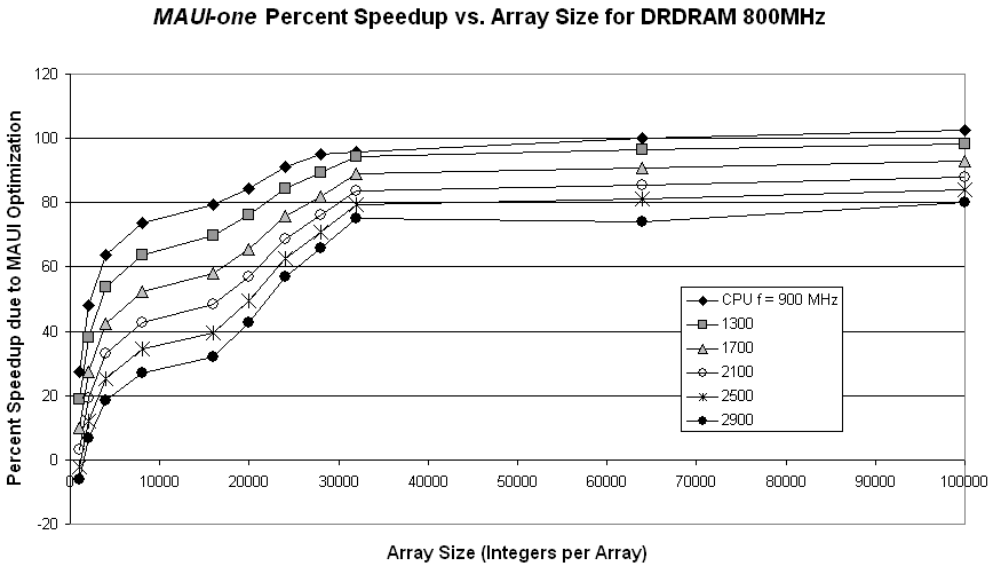

Figure C.8: Simulation results of *MAUI-one* when run with a 800 MHz *DRDRAM* memory system.

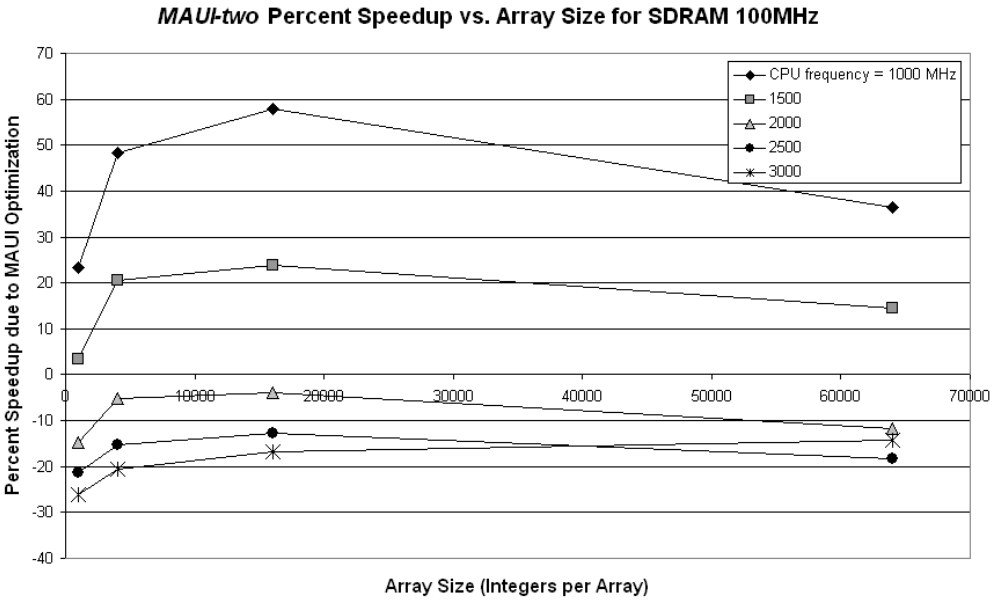

Figure C.9: Simulation results of *MAUI-two* when run with a 100 MHz *SDRAM* memory system.

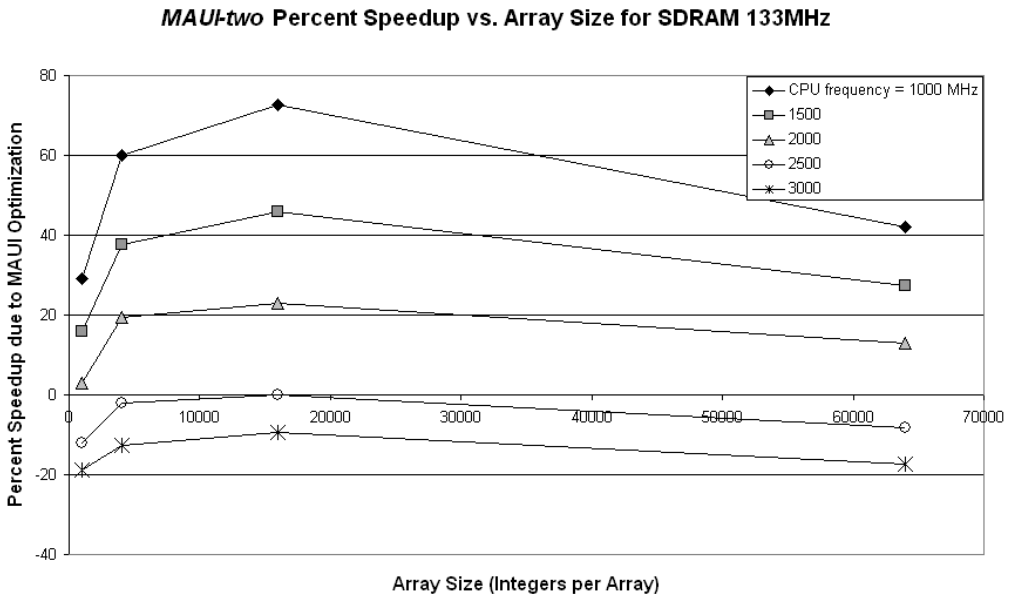

Figure C.10: Simulation results of *MAUI-two* when run with a 133 MHz *SDRAM* memory system.

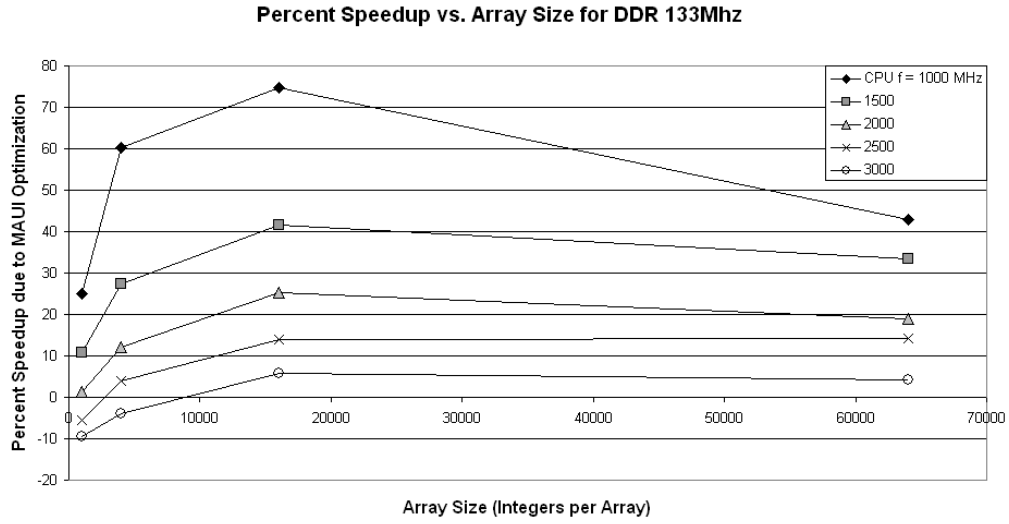

Figure C.11: Simulation results of *MAUI-two* when run with a 133 MHz *DDR-SDRAM* memory system.

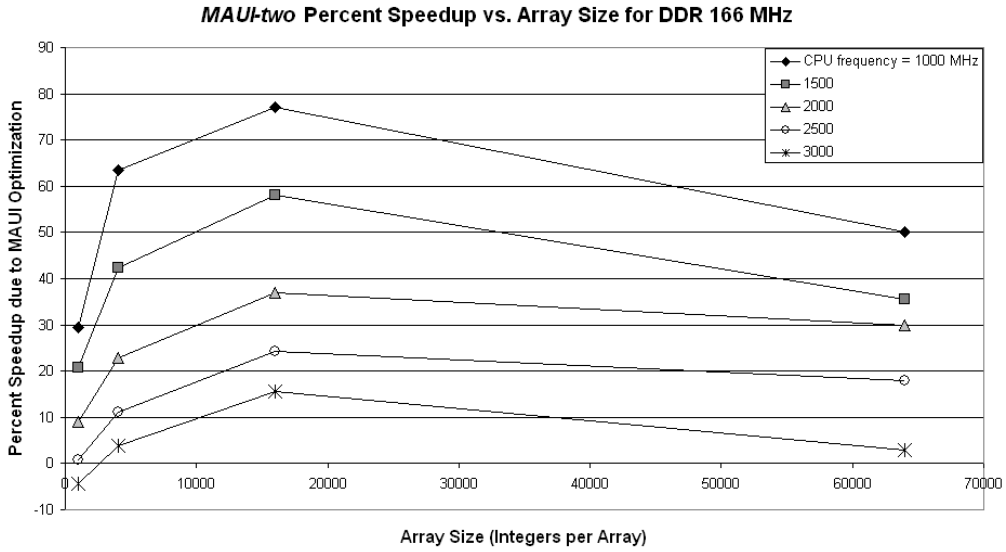

Figure C.12: Simulation results of *MAUI-two* when run with a 166 MHz *DDR-SDRAM* memory system.

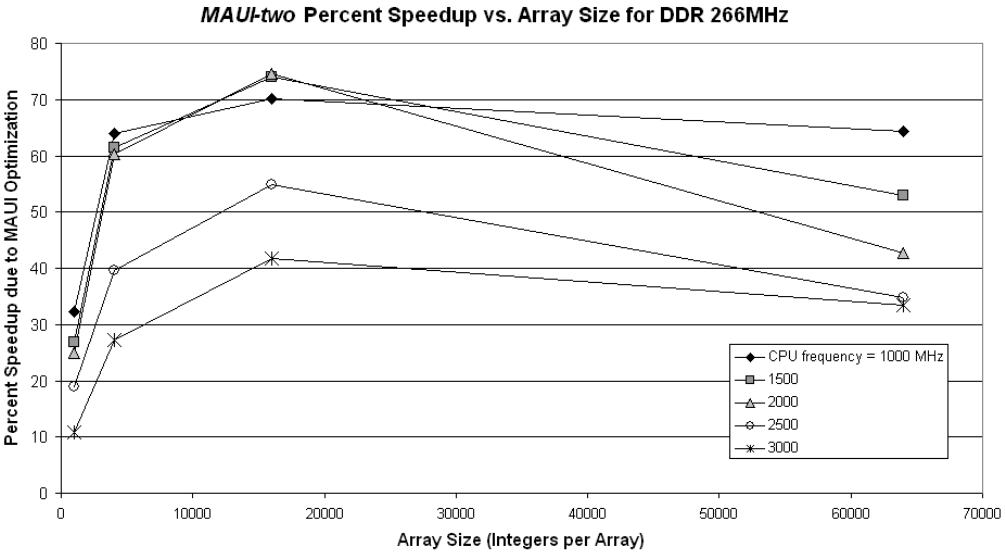

Figure C.13: Simulation results of *MAUI-two* when run with a 266 MHz *DDR-SDRAM* memory system.

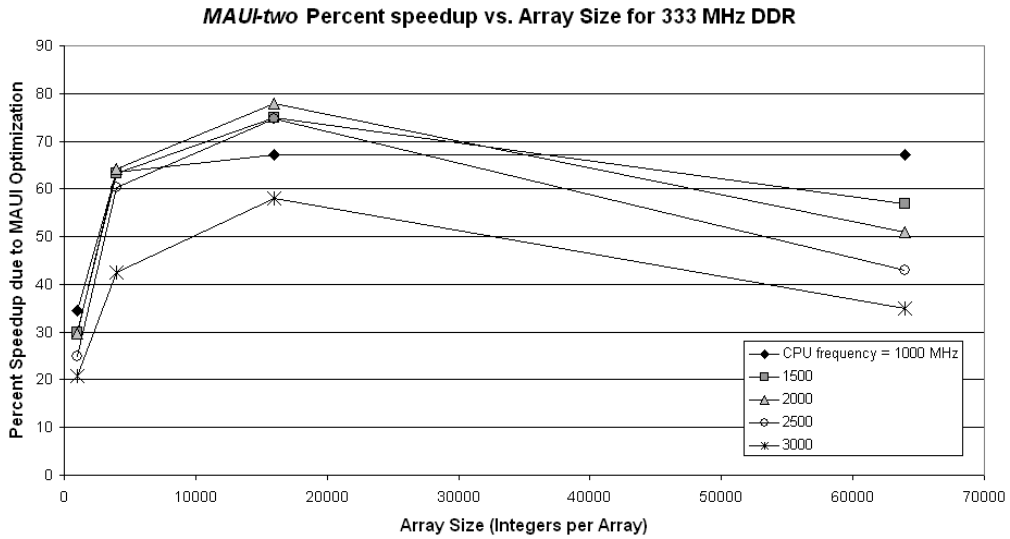

Figure C.14: Simulation results of *MAUI-two* when run with a 333 MHz *DDR-SDRAM* memory system.

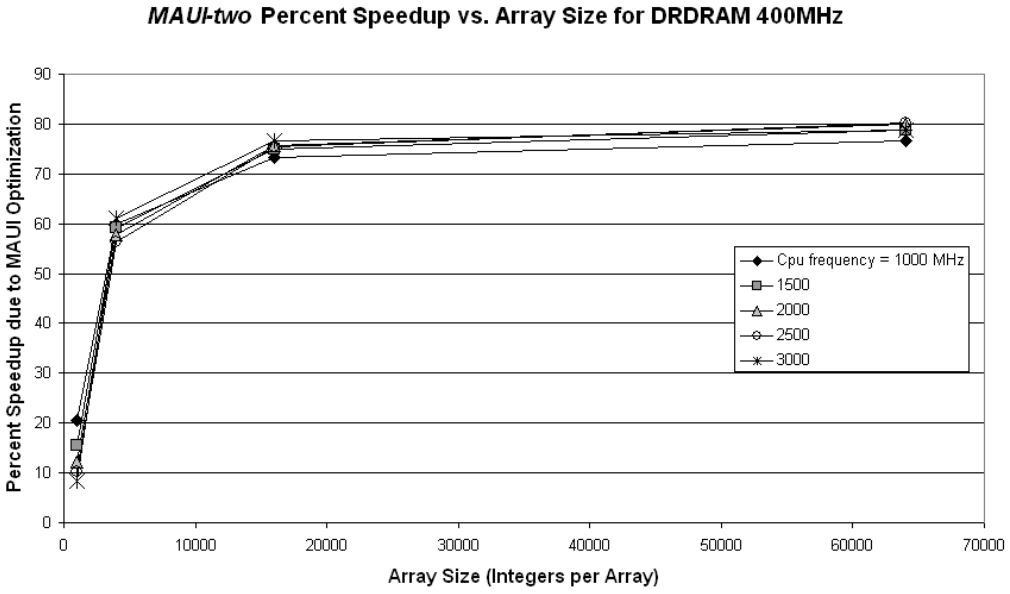

Figure C.15: Simulation results of *MAUI-two* when run with a 400 MHz *DR-DRAM* memory system.

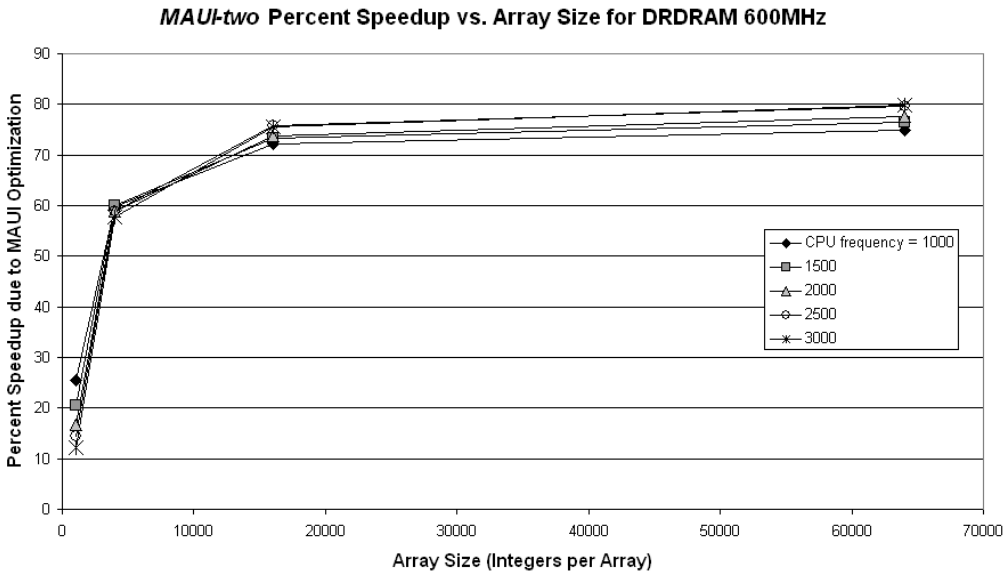

Figure C.16: Simulation results of *MAUI-two* when run with a 600 MHz *DRDRAM* memory system.

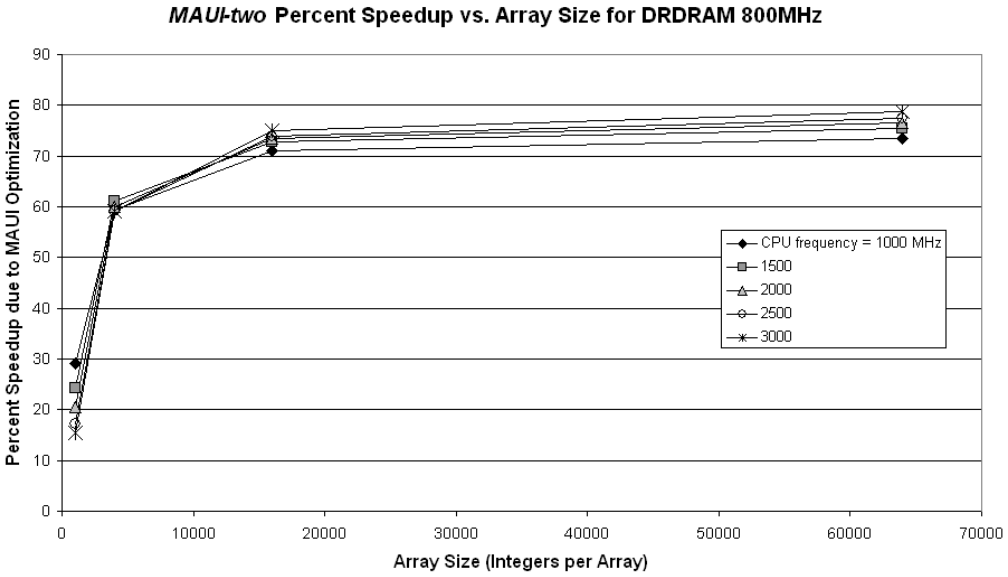

Figure C.17: Simulation results of *MAUI-two* when run with a 800 MHz *DRDRAM* memory system.

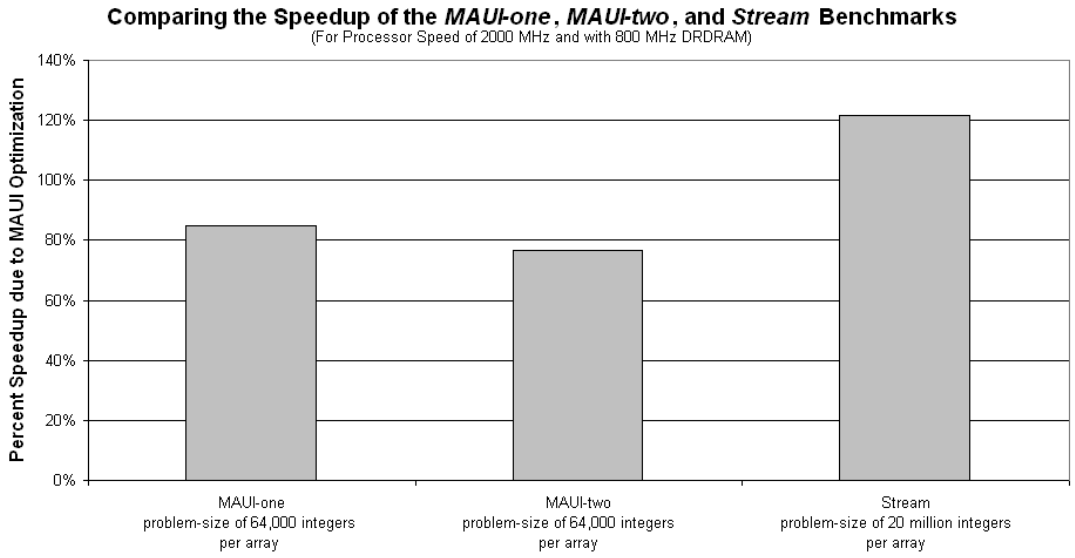

Figure C.18: Graph comparing the speedup of *MAUI-one, MAUI-two* and *Stream* due to MAUI optimizations.

BIBLIOGRAPHY

[1] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, June 1997.

[2] Doug Burger, James R. Goodman, and Alain Kägi. Memory Bandwidth Limitations of Future Microprocessors. In *Proceedings of the 23rd IEEE/ACM International Symposium on Computer Architecture (ISCA)*, pages 78–89, Philadelphia, Pennsylvania, May 1996.

[3] Douglas C. Burger, Alain Kägi, and James R. Goodman. The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors. Technical Report UWMADISONCS CS-TR-95-1261, University of Wisconsin Computer Sciences Department, Madison, Wisconsin, January 1995.

[4] Sourav Chatterji and Jason Duell. Performance Evaluation of Two Emerging Media Processors: VIRAM and Imagine. In *Proceedings of the Workshop on Parallel and Distributed Image Processing, Video Processing, and Multimedia*, pages 229–235, Nice, France, April 2003.

[5] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*, pages 273–283. Morgan Kaufmann, San Francisco, California, 1999.

[6] Brian R. Gaeke, Parry Husbands, Xiaoye S. Li, Leonid Oliker, Katherine A. Yelick, and Rupak Biswas. Memory-Intensive Benchmarks: IRAM vs Cache-Based Machines. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 30–36, Ft. Lauderdale, Flordia, April 2002.

[7] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer*, 28(4):23–31, April 1995.

[8] Mary Hall, Peter Kogge, Jeff Killer, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoov Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping Irregular Applications to DIVA, A PIM-based Data-Intensive Architecture. In *Proceedings of the ACM/IEEE International Conference on Supercomputing (ICS)*, number 56, Portland, Oregon, November 1999.

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, pages 11–17,390–504. Morgan Kaufmann, Menlo Park, California, 2003.

[10] David Judd, Katherine Yelick, Christoforos Kozyrakis, David Martin, and David Patterson. Exploiting On-Chip Memory Bandwidth in the VIRAM Compiler. In *Revised Papers of the 2nd International Workshop on Intelligent Memory Systems*, pages 122–134, Cambridge, Massachusetts, 2000.

[11] Chang Woo Kang and Jeffrey Draper. A Fast, Simple Router for the Data-Intensive Architecture (DIVA) System. In *Proceedings of the 43rd IEEE*

Midwest Symposium on Circuits and Systems, volume 1, pages 188–192, Lansing, Michigan, August 2000.

[12] Y. Kang, J. Torrellas, and T. S. Huang. Use IRAM for Rasterization. In *Proceedings of the 3rd IEEE International Conference on Image Processing (ICIP)*, volume 3, pages 1010–1013, Chicago, Illinois, October 1998.

[13] Yi Kang, Josep Torrellas, and Thomas S. Huang. An IRAM Architecture for Image Analysis and Pattern Recognition. In *Proceedings of the 14th IEEE International Conference on Pattern Recognition (ICPR)*, volume 2, pages 1561–1564, Brisbane, Australia, August 1998.

[14] Diana Keen, Mark Oskin, Justin Hensley, and Frederic T Chong. Cache Coherence in Intelligent Memory Systems. *IEEE Transactions on Computers*, 52(7):960–966, July 2003.

[15] Eric Larson, Saugata Chatterjee, and Todd Austin. MASE: A Novel Infrastructure for Detailed Microarchitectural Modeling. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software: ISPASS*, pages 1–9, Tucson, Arizona, November 2001.

[16] John D. McCalpin. A Survey of Memory Bandwidth and Machine Balance in Current High Performance Computers. *Newsletter of the IEEE Technical Committee on Computer Architecture (TCCA)*, December 1995.

[17] Micron Technology, Inc. Micron Datasheet: 256Mb: x4, x8, x16 DDR SDRAM. *http://download.micron.com/pdf/datasheets/dram/ddr/256Mx4x8x16DDR.pdf*, 2003.

[18] NVIDIA. Technical brief: NVIDIA nForce Integrated Graphics Processor (IGP) and Dynamic Adaptive Speculative Pre-Processor (DASP). *http://www.nvidia.com/object/dasp.html*, 2003.

[19] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Computation Model for Intelligent Memory. In *Proceedings of the 25th Annual IEEE International Symposium on Computer Architecture (ISCA)*, pages 192–203, Barcelona, Spain, June 1998.

[20] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. ActiveOS: Virtualizing Intelligent Memory. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 202–209, Austin, Texas, October 1999.

[21] Mark Oskin, Justin Hensley, Diana Keen, Frederic T. Chong, Matthew Farrens, and Aneet Chopra. Exploiting ILP in Page-Based Intelligent Memory. In *Proceedings of the 32nd Annual ACM/IEEE International Symposium on Micro-Architecture*, pages 208–218, Haifa, Israel, 1999.

[22] Mark Oskin, Diana Keen, Justin Hensley, Lucian-Vlad Lita, and Frederic T. Chong. Reducing Cost and Tolerating Defects in Page-based Intelligent Memory. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 276–284, Austin, Texas, 2000.

[23] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberley Keeton, Christoforos Kozyrakis, Randi Thomas, and Kathy Yekick. Intelligent RAM (IRAM): Chips that Remember and Compute. In *Proceedings of the 44th IEEE International Solid-State Circuits Conference (ISSCC)*, pages 224–225, San Francisco, California, February 1997.

[24] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A Case for Intelligent RAM. *IEEE Micro*, pages 34–44, March/April 1997.

[25] David A. Patterson. IRAM: A Microprocessor for the Post-PC Era. In *Proceedings of Tech. Papers, the IEEE International Symposium on VLSI Technology, Systems, and Applications*, pages 39–41, Taipei, Taiwan, June 1999.

[26] Yan Solihih, Jaejin Lee, and Josep Torrellas. Using a User-Level Memory Thread for Correlation Prefetching. In *Proceedings of the 29th Annual IEEE International Symposium on Computer Architecture (ISCA)*, pages 171–182, Anchorage, Alaska, May 2002.

[27] Yan Solihih, Jaejin Lee, and Josep Torrellas. Correlation Prefetching with a User-Level Memory Thread. *The IEEE Transactions on Parallel and Distributed Systems*, 14(6):563–580, 2003.

[28] Jinwoo Suh, Changping Li, Stephen P. Crago, and Robert Parker. A PIM-based Multiprocessor System. In *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium*, page 10004, San Francisco, April 2001.

[29] Kingston Technology. http://www.kingston.com/company/about.asp. *Kingston Technology homepage*, 2004.