Performance Characteristics of MAUI: An Intelligent Memory System Architecture

Justin Teller
University of Maryland
Department of Electrical and Computer Engineering
College Park, Maryland
teller.9@osu.edu

Charles B. Silio, Jr.
University of Maryland
Department of Electrical and Computer Engineering
College Park, Maryland
silio@Glue.umd.edu

Bruce Jacob
University of Maryland
Department of Electrical and Computer Engineering
College Park, Maryland
blj@eng.umd.edu

ABSTRACT

Combining ideas from several previous proposals, such as Active Pages, DIVA, and ULMT, we present the Memory Arithmetic Unit and Interface (MAUI) architecture. Because the "intelligence" of the MAUI intelligent memory system architecture is located in the memory-controller, logic and DRAM are not required to be integrated into a single chip, and use of off-the-shelf DRAMs is permitted. The MAUI's computational engine performs memory-bound SIMD computations close to the memory system, enabling more efficient memory pipelining. A simulator modeling the MAUI architecture was added to the SimpleScalar v4.0 tool-set. Not surprisingly, simulations show that application speedup increases as the memory system speed increases and the dataset size increases. Simulation results show single-threaded application speedup of over 100% is possible, and suggest that a total system speedup of about 300% is possible in a multi-threaded environment.

General Terms

Performance

Keywords

intelligent memory, memory architecture, SimpleScalar simulator, data-intensive calculations, MAUI memory architecture, SIMD processing, vector processing

1. INTRODUCTION

While processor performance has increased by about 58% annually since 1994, memory system performance has not increased as quickly as the processor's performance. Dynamic Random Access Memory (DRAM) latency has decreased by only about 7% annually, and DRAM bandwidth has increased about 15% 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, making the performance bottleneck worse [8].

Intelligent memory systems show promise in overcoming the memory system performance bottleneck by building computational ability into the memory system. Several intelligent memory systems have already been proposed, such as Active Pages [14], the Data IntensiVe Architecture (DIVA) [7], Intelligent RAM (IRAM) [17], the User-Level Memory Thread (ULMT) architecture [19], and others [3, 4, 6, 11, 16, 20, 21].

Many of these intelligent memory system architectures have shown impressive application speedup in simulation. In particular, The Active Pages project can improve performance by a factor of about 1000 times [14], a Vector IRAM (VIRAM) architecture can provide a 100% speedup [5], and the ULMT can provide up to a 58% speedup for some applications, despite only acting as a prefetching device [18]. Despite impressive simulation studies, none of the proposed intelligent memory system architectures has gained popular support for consumer computer systems. Except for ULMT, perhaps this is due to the expense of integrating logic and DRAM onto a single silicon die instead of using commodity DRAM.

This paper presents a new intelligent memory system architecture: 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. The MAUI architecture migrates computational power into the memory system, but 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 presented in Section 2. Section 3 discusses the details of an augmented memory system version of the SimpleScalar v4.0 simulation environment used to test the performance of the MAUI architecture and presents simulated performance results for the MAUI architecture. Section 4 presents conclusions.

2. THE MAUI ARCHITECTURE

Similar to the Active Pages and DIVA architectures, MAUI memory operations are explicitly invoked by the host processor, meaning that the processor's instruction set is aug-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSP '05, Chicago, USA
Copyright 2005 ACM 1-59593-147-3/05/06 ...$5.00.

mented 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, more complicated operations could be possible. However, other memory bound operations, such as pointer chasing, searching, and sorting are not presented in this paper. By providing the host processor with explicit control of specialized memory operations, the MAUI architecture resembles both the Active Pages and DIVA architectures[1]. While this paper only presents MAUI vector operations, more complicated and useful operations could be possible.

The placement of computational power within the memory controller decreases latency and increases bandwidth to memory when compared to the host processor. Additionally, the expression of massive Instruction Level Parallelism (ILP) allows for more efficient access to memory and reduced cache overhead. 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.

2.1 MAUI Software Interface

Conventional memory systems support only two commands from the processor, read and write[2]. 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. It is important to note that data movement is a very simple vector operation.

The MAUI commands are 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 and the scaling of a single vector.

For instance, implementing a block copy using MAUI commands requires the use of four commands to the MAUI 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 size of the memory block which will be copied. The command *maui-ADD-scalar* is used to copy data by setting the scaling value to zero: this is the execution command to begin copying the data. Figure 1 shows the pseudo-C code for a block copy function named *bcopy* implemented with the four previously mentioned MAUI commands.

Although MAUI operations can take a significant amount of time to complete, and the latency of a 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 logical ordering of memory accesses and MAUI operations is maintained automatically by the MAUI architecture while allowing independent memory accesses to proceed and complete before the MAUI operation has completed.

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

Figure 1: A MAUI implementation of bcopy.

2.2 MAUI Hardware

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 occurs only at the memory controller. 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.

2.2.1 The MAU

The Active Pages project demonstrated that the performance gain in using intelligent memory system architectures is due mostly to Instruction Level Parallelism (ILP) [15]. To exploit available ILP, the MAUI architecture performs vector computations on vectors as wide as a cacheline. 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 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 allows for significant exploitation of ILP.

2.2.2 The MAUI

The MAUI controls memory computations and acts as the intelligent memory system's interface to the rest of the computer system. The MAUI the heart of the MAUI intelligent memory system computational model. The MAUI coordinates its caches, includes dedicated registers to hold

[1]Note that it is easy to implement the MAUI instructions via I/O operations, which would be orthogonal to the instruction set.

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

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. A block level schematic of the MAUI architecture is shown in Figure 2. Notice that all of the data flow in the MAUI architecture passes through the MAUI.

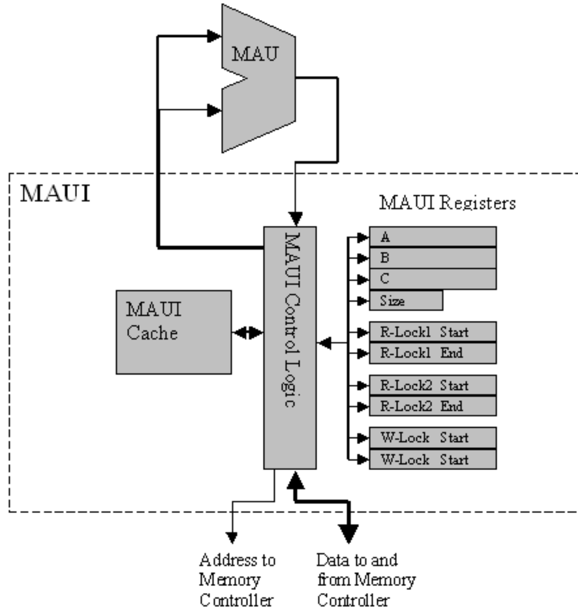

Figure 2: The block diagram of the MAUI architecture.

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 2 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 a MAUI execution command. When the MAUI receives an execution command, it begins the execution of that command. Generally, the MAUI begins 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 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 of traditional memory accesses and MAUI operations, the MAUI architecture introduces the concept of locking memory. The MAUI maintains two types of memory locks, *Read* and *Write* locks. 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 yet over-written.

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.

2.3 Possible Drawbacks

One possible performance pitfall for the MAUI augmented memory system is the cost of maintaining cache coherence. By operating on data within the memory system, we run the risk of changing data that is also 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 [12]. Because software driven cache coherence results in a more simple hardware implementation, hardware cache coherence is not explored for the MAUI architecture. However, hardware cache coherence has strengths, such as a reduction to the cache-processor bandwidth used. Quantifying the effect of software versus hardware driven cache coherence within this architecture is left to future work.

Another issue is introduced when examining the nature of virtual memory. When a virtual virtual memory is trans-

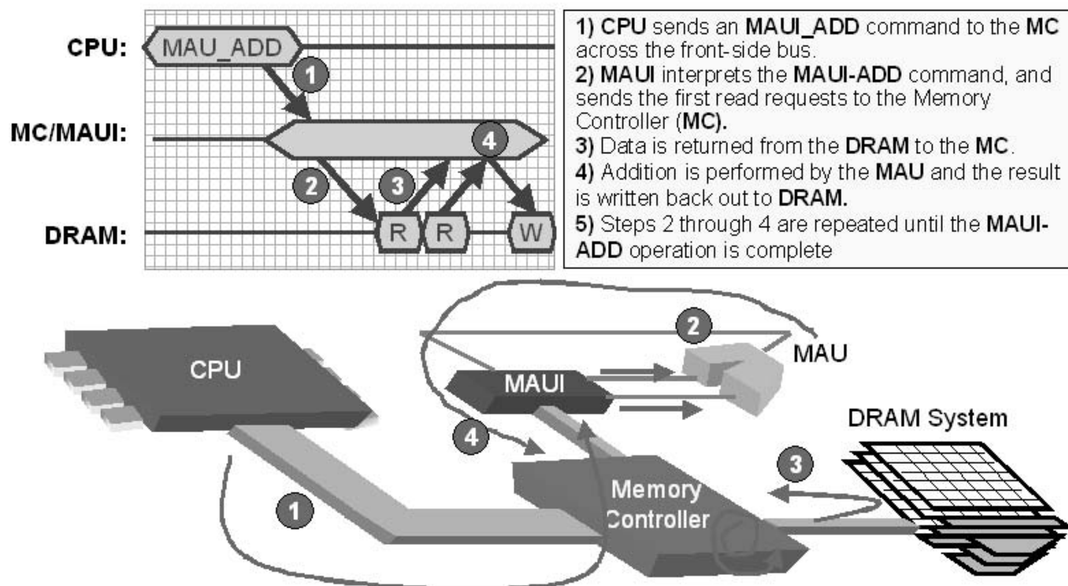

Figure 3: Illustration of the MAUI add operation. The MAUI has already been setup with the vector size and the source and destination addresses.

lated to physical addresses, it is clear that a contiguous segment of virtual memory does not always translate to a contiguous segment of physical memory [8]. The existence of virtual memory does not align well with the assumptions the MAUI makes. The MAUI has access only to the physical addresses, and expects the source and destination vectors to have consecutive addressing. Therefore, when a vector crosses a page boundary it is possible for the parts of the vectors that fall in different pages to be mapped to non-consecutive physical memory pages. 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 should not 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.

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 are not 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. The operating system usually manages virtual memory, so an obvious solution to the virtual memory problem is to have the operating system handle the correctness of MAUI commands. Therefore, the programmer would issue an operating system call asking to perform the MAUI vectors operation. The operating system 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. Unfortunately, operating system calls may entail significant overhead, canceling out any performance advantage from using MAUI operations. As the largest performance gains are for large problem sizes, it is expected that the operating system call overhead will be small compared to the total running time in these cases. Other, more complicated solutions could be devised, such as having the MAUI hardware keep track of the virtual memory mapping. However, other solutions are not explored in this paper.

3. MAUI SIMULATION

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 MAUI architecture calls for additional instructions to be added to the host processor's instruction set. 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 were hand optimized and written partially in assembly language [22].

The processor configuration used in each simulation is noted in Table 1. Every simulation used the same processor configuration, unless otherwise noted. In all simulations, performance speedup is calculated as:

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

where $\text{sim-cycle}_{non-MAUI}$ is the number of processor cycles SimpleScalar reported that the benchmark took to complete using only the processor and sim-cycle_{MAUI} is the number of processor cycles SimpleScalar reported that benchmark took to complete when using the MAUI hardware.

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 1: Simulated processor configuration.

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. *MAUI-one* performs a single vector operation, while *MAUI-two* performs two independent vector operations.

MAUI-one 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, *MAUI-one*'s simulations can provide cues to which vector operations should be off-loaded to the MAUI hardware for performance gains.

MAUI-two 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, while simultaneously stressing the memory system.

The final benchmark, *Stream* tests the performance of the memory system by performing vector additions, multiplications, scaling, and multiplication-accumulation on extremely large vectors [13]. The purpose of the *Stream* benchmark was to test how the MAUI enhanced intelligent memory system affects total memory system performance. In the MAUI optimized version of *Stream*, several operations were performed using the MAUI hardware, while other operations remained within the processor.

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. A large problem size of two million integers is standard for the implementation of the *Stream* benchmark, to stress the memory system accordingly.

The *Stream* benchmark uses over 7MB of memory per array, so no single vector used in the *Stream* benchmark can fit in the caches. By making *Stream's* dataset too large to fit in the cache, *Stream* provides an indication of total memory system performance. There are many real-world examples of programs whose datasets are 7MB or larger, and whose operations fit well into a vector computing paradigm; examples include audio and image processing, video compression and decompression, and scientific computing and visualization [9].

3.1 Simulation Results

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 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, 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 is repeated for each combination of processor speed and problem size.

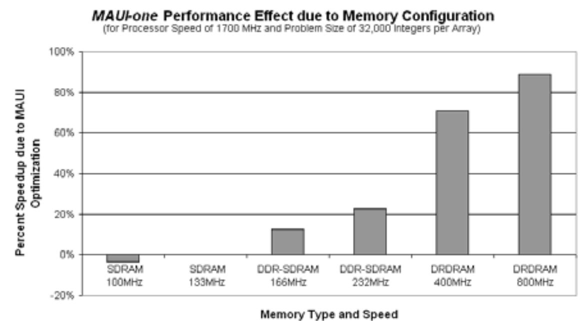

Figure 4: Graph illustrating the effect memory configuration has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark simulated with a processor speed of 1700 MHz and a problem size of 32,000 integers per array.

Figure 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 MAU and MAUI are only operating at 100 MHz.

The effect of processor speed on the speedup due to the MAUI architecture is shown in Figure 5. Figure 5 shows that, as the processor speed increases, the 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.

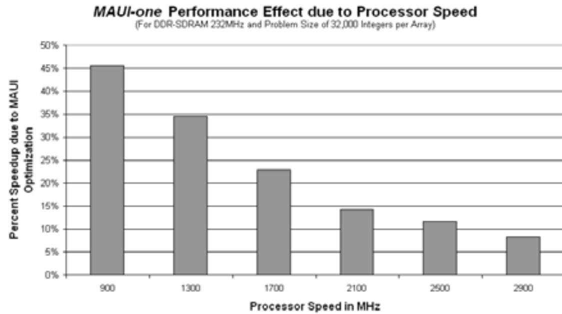

Figure 5: Graph illustrating the effect processor speed has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark.

The effect problem size has on the speedup due to the MAUI architecture is shown in Figure 6. Examining Figure 6, one can deduce that, as the problem size increases, the percent speedup due to optimization for the MAUI architecture also increases. The general reason for the speedup of *MAUI-one* was that the MAUI instructions allow for the expression of a large amount of memory operation ILP directly in the code. Expressing this ILP greatly increases the efficiency of memory accesses. By not bring this data into the cache, there is also significantly lower cache overhead. A lower cache overhead would have an even larger impact on more complex benchmarks, by allowing the non-MAUI portions of the benchmark to occupy more of the cache.

For smaller datasets, the data is loaded into the cache by the initialization, and the data remains there for the entire simulation. *MAUI-one* shows no speedup for these small problem sizes because when the dataset fits in the cache, the processor has a faster connection to the dataset than the MAUI hardware has to main memory. 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. The largest problem size simulated had 100,000 integers per array, meaning that the total size of memory utilized was slightly more than 1MB. Data sets this large and larger are present in many multimedia applications, as well as scientific simulations and visualization applications. Once the dataset no longer fits in the cache, we see that speedup saturates, and we expect the performance of MAUI operations to be similar for datasets larger than 1MB.

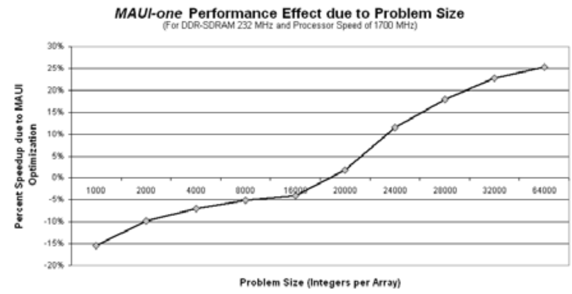

Figure 6: Graph illustrating the effect problem size has on the speedup due to the MAUI architecture for the *MAUI-one* benchmark.

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 7 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 7, 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.

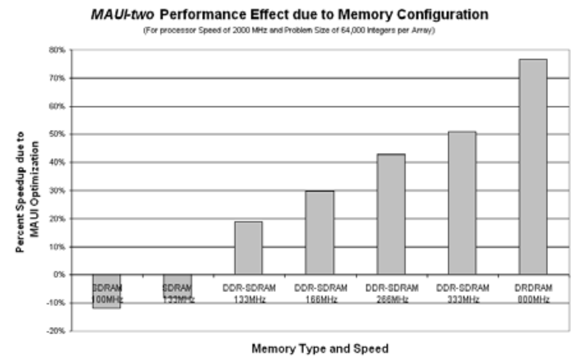

Figure 7: Graph illustrating the effect memory configuration has on the speedup due to the MAUI architecture for the *MAUI-two* benchmark.

Figure 8 shows the effect processor speed has on the speedup of *MAUI-two* due to MAUI optimizations. The trend shown in Figure 8 is the same as the trend shown in Figure 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.

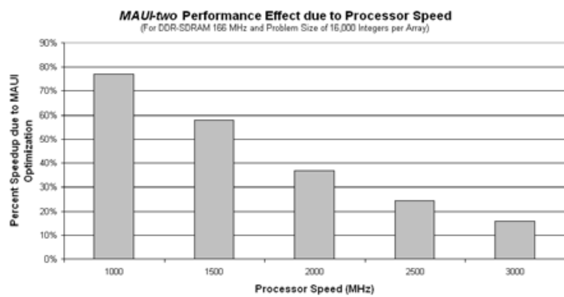

Figure 8: Graph illustrating the effect processor speed has on the speedup due to the MAUI architecture for the *MAUI-two* benchmark.

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 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 9.

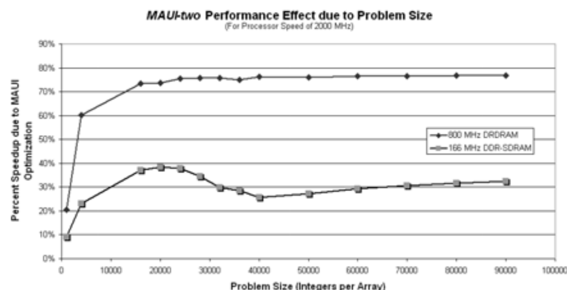

Figure 9: Graph illustrating the effect problem size has on the speedup due to the MAUI architecture for the *MAUI-two* benchmark.

There is one significant difference when comparing the effect problem size has on the speedup due to MAUI optimization for *MAUI-one* (Figure 6) to the effect problem size has on the speedup due to MAUI optimization for *MAUI-two* (Figure 9). Looking at Figure 9, 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 for which the *MAUI-one* benchmark experiences the performance "knee". However, for *MAUI-two* this translates to a decline in the speedup, instead of the increased speedup, as is seen in the *MAUI-one* benchmark. 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 has only 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 9 because the bandwidth available for that memory system type is significantly greater than that of 166 MHz *DDR-SDRAM*.

The speedup of the *MAUI-two* benchmark is due to exploiting parallelism. 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 did not compete with the MAUI hardware for memory access.

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. 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 [13].

Simulating the *Stream* benchmark showed a 121.5% speedup due to MAUI optimization. Figure 10 compares the speedup of *MAUI-one*, *MAUI-two*, and *Stream* for identical memory configurations and processor speeds. There are two reasons for *Stream's* speedup. First, three vector operations are performed with the MAUI hardware. As shown in simulations of *MAUI-one*, each of 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* is because the final vector operation is performed using the processor, in parallel with the MAUI 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 operation takes longer than the preceding *maui_add* operation, and the final vector operation competes for memory bandwidth with the MAUI.

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-

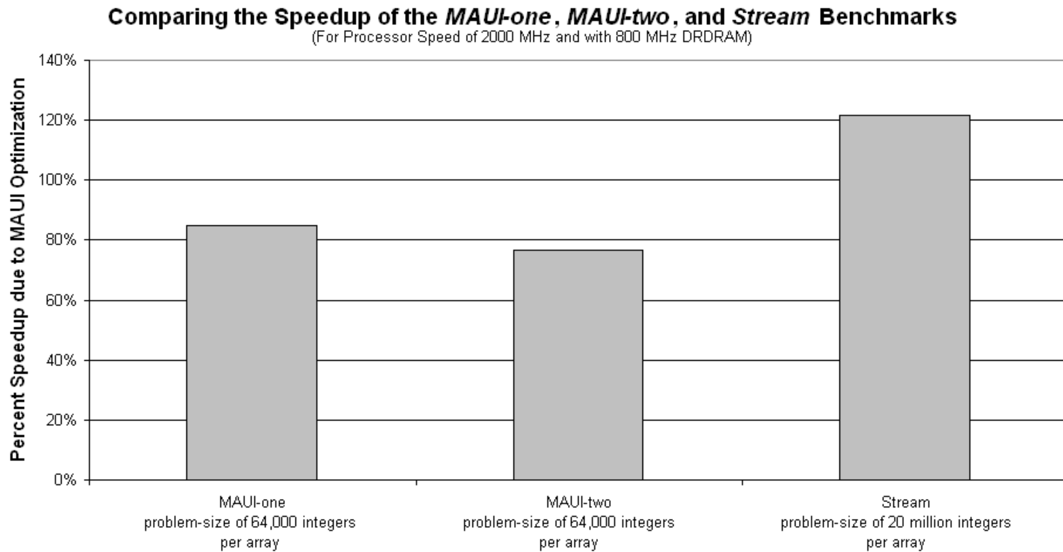

Figure 10: Graph comparing the speedup of *MAUI-one, MAUI-two* and *Stream* due to MAUI optimizations.

bound benchmarks can exceed that predicted by the simplistic *MAUI-one* and *MAUI-two* benchmarks.

4. CONCLUSIONS

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 typically fall into one of two categories: either they migrate computational power into the DRAM system, or they migrate DRAM into the main processor [2].

This paper has presented a new intelligent memory system architecture which takes a slightly different approach. The MAUI architecture integrates additional computational power into the same chip as the memory controller. 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. Additionally, because the MAUI hardware is a separate processing element from the processor, further application speedup is possible by exploiting parallelism. However, recent trends place the memory controller onto the same die as the processor, canceling the bandwidth and latency advantage for the MAUI hardware. The MAUI architecture still has value, by expressing significant ILP directly in the code. Expressing the ILP in the code allows for more efficient access of memory and reduced cache overhead. Furthermore, some recent architectures, such as the Cell processor, are shrinking their instruction window in favor of more computational parallelism [10]. The MAUI architecture can provide architectures resembling the Cell processor the opportunity to hide memory latency without the pipelining and other overheads associated with large instruction windows.

Simulations 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. Using a 2GHz processor coupled with 800 MHz DRDRAM enables around an 80% speedup for both *MAUI-one* and *MAUI-two*. A system of this nature is fairly similar to modern desktop computers. Performing three of *Stream*'s vector operations using the MAUI hardware resulted in a 121% speedup over a normal organization. *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.

Unfortunately, the overhead associated with managing virtual memory and using the MAUI architecture was not explored in this paper. For the large problem sizes that experience significant speedup using the MAUI architecture, the context switch overhead is expected to be significantly less than the time required to perform the MAUI operation. Therefore, it is expected that the operating system overhead for MAUI calls would have a negligible impact on performance for most problem sizes that exhibit a performance increase.

There are a number of possible real-world applications that may benefit from MAUI hardware. Examples include audio and image processing, video compression and decompression, and scientific computing and visualization. Furthermore, the operating system may need to copy large sections of data from one place to another in memory. One example occurs in a in a cluster environment. In this particular parallel architecture, remote messages are copied into buffers within a particular process. Copying data within memory is one domain where the MAUI architecture can benefit the operating system.

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 11.

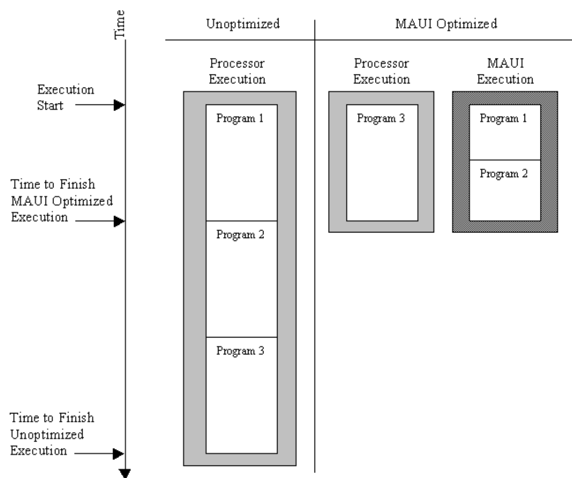

Figure 11: Illustration of how the MAUI architecture could increase total system performance by up to 300% on a multiprogrammed computer system. In this example, Program 1 and 2 are both memory-bound programs that take the same amount of time to complete as Program 3 when executed by the processor, but each take half that time when executed by the MAUI architecture.

4.1 Future Work

There is still room for significant research with the MAUI project. First, the MAUI architecture operates only within a very constrained SIMD paradigm. Additional support for more general purpose vector processing would make the MAUI architecture significantly more powerful. Future possibilities for operations include searches, scatter-gather operations, pointer chasing, or other memory bound operations which express significant ILP.

The second research direction would be to quantify the effect of the operating system on MAUI performance. If the operating system signifies significant performance overhead, other methods of managing the nature of virtual memory would need to be explored. These may include exposing the nature of virtual memory to the MAUI hardware, possibly by having the operating system post changes to the virtual memory to hardware.

The next direction is for further research into which applications the MAUI hardware would be most useful. 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. Lastly, increased software support for the MAUI architecture, such as a MAUI aware compiler, will simplify the creation of MAUI optimized software.

5. REFERENCES

[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] John Carter, Wilson Hsieh, Leigh Stoller, Mark Swanson, Lixon Shang, Erik Brunvand, Al Davis, Chen-Chi Kuo, Ravindra Kuramkote, Michael parker, Lambert Schaelicke, and Terry Tateyama. Impulse: Building a Smarter Memory Controller. In *Proceedings of the 5th IEEE Int. Conference on High Performance Computer Architecture*, pages 70–79, Orlando, Florida, January 1999.

[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] 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.

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

[7] 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)*, Portland, Oregon, November 1999.

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

[9] Mark Hereld, Rick Stevens, Justin Teller, Wim van Drongelen, and Hyong Lee. Large Neural Simulations on Large Parallel Computers. To Appear. In *The Joint Meeting of the 5th International Conference on Bioelectromagnetism and the 5th International Symposium on Noninvasive Functional Source Imaging within the Human Brain and Heart (BEM NFSI 2005)*, Minneapolis, Minnesota, May 2005.

[10] H. Peter Hofstee. Power Efficient Processor Architecture and The Cell Processor. In *Proceedings of the 11th IEEE International Symposium on High-Performance Computing (HPCA-11 2005)*, pages 258–262, San Francisco, California, February 2005.

[11] Yi Kang, Wei Huang, Seung-Moon Yoo, Diana Keen, Zhenshou Ge, Vinh Lam, Pratap Pattnaik, and Josep Torrellas. FlexRAM: Toward and Advanced Intelligent Memory System. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*, pages 193–201, Austin, Texas, October 1999.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Michael Taylor and et. al. The RAW Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, pages 25–35, March/April 2002.

[22] Justin Teller. Performance Characteristics of an Intelligent Memory System. Master's thesis, University of Maryland, August 2004.