EFFECT OF MEMORY ACCESS AND CACHING ON HIGH PERFORMANCE COMPUTING

by

JAMES GROENING

B.S., Kansas State University, 2010

A THESIS

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2012

Approved by:

Major Professor
Dr. Dwight Day

Copyright

JAMES GROENING

2012

Abstract

High-performance computing is often limited by memory access. As speeds increase, processors are often waiting on data transfers to and from memory. Classic memory controllers focus on delivering sequential memory as quickly as possible. This will increase the performance of instruction reads and sequential data reads and writes. However, many applications in high-performance computing often include random memory access which can limit the performance of the system. Techniques such as scatter/gather can improve performance by allowing nonsequential data to be written and read in a single operation. Caching can also improve performance by storing some of the data in memory local to the processor.

In this project, we try to find the benefits of different cache configurations. The different configurations include different cache line sizes as well as total size of cache. Although a range of benchmarks are typically used to test performance, we focused on a conjugate gradient solver, HPCCG. The program HPCCG incorporates many of the elements of common benchmarks used in high-performance computing, and relates better to a real world problem. Results show that the performance of a cache configuration can depend on the size of the problem. Problems of smaller sizes can benefit more from a larger cache, while a smaller cache may be sufficient for larger problems.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my committee members Dr. Dwight Day, Dr. Don Gruenbacher, and Dr. John Devore for their guidance and support through the thesis process. In particular, I would like to thank Dr. Day for his encouragement and consultation. I would also like to thank the support of Sandia. I am especially grateful for the assistance of Doug Doerfler.

Chapter 1 - Introduction

The “memory wall [1] [2]” is a problem that has been around for a long time. While memory speeds have continually increased, they have not been able to keep up with microprocessors. Microprocessor performance has increased at a rate of 60% per year, while DRAM performance has increased by only 10% [3]. This wall is also caused by the Von Neumann bottleneck [4]. The CPU must access both program and data memory through the same bus. This leads to a limited throughput of memory to the CPU; therefore the CPU will have idle cycles while it waits for the memory to be accessed. In addition to this, the memory performance is highly dependent on how it is accessed [5]. The increase in CPU speed along with the increase in memory size has only made the problem worse.

Several methods have been implemented in order to alleviate this problem. The integration of CPU and memory onto a single chip should lead to lower latency and increased bandwidth [3] [6]. However, because memory and processor technology are so different, tradeoffs must be made, such as smaller amounts of memory and a less complex processor architecture. A new type of memory controller that intelligently accesses memory can increase performance for certain applications [7]. But this adds complexity and latency for the memory controller. Changes to the hierarchical structure of memory access have also been proposed [8]. Caches are an important part of the hierarchical memory structure. Cached memory can dramatically reduce the access time of data. There are several different cache architectures.

Cache Structure

The cache of a CPU is designed to provide data quickly to the processor in order to speed up execution. This way the processor is not idle while waiting for data from main memory. The

cache is limited in size because of cost and power considerations and cannot contain all of main memory. Therefore, cache controllers have different algorithms for deciding where to store data in the cache [9]. Different procedures are also used for writing to cache and main memory.

A cache is generally divided into blocks of memory called cache lines. Cache lines have a set size, but the size may vary from cache to cache. When a CPU loads a word from memory, the cache is checked first. If a copy of the data resides in the cache, the data will be loaded from the cache. This is considered a cache hit. Otherwise the data will be loaded from main memory and the cache will be updated with the new data. This is considered a cache miss. In general, for a cache miss, an entire cache line is replaced. If there is a cache miss then there must be an algorithm for determining which cache line the newly retrieved memory will fill. The associativity of the cache determines where data will be stored in cache.

A direct-mapped cache is the simplest algorithm for deciding cache location. In a direct-mapped cache, every location in main memory has one cache line it is associated with. The location is determined by the address of the data. The address can be divided into a tag, index, and block offset as in Figure 1.1. In this case the address contains 28 bits, for 256MB of memory.

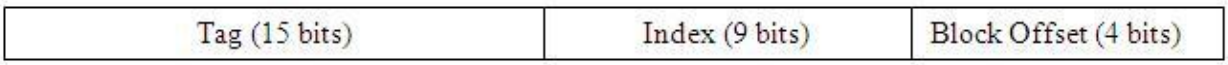

Figure 1.1 Example of a 28-Bit Memory Address

The block offset simply determines the data desired within the cache line. In this example the block offset is 4 bits, meaning each cache line contains 16 bytes. The index specifies which cache line the address is associated with. The index along with the block offset determines where

the data should be stored in the cache. The index of 9 bits, along with the 4-bit block offset, indicates a cache size of 8kB. Since multiple memory segments can be stored in the same cache line, there must be a way to tell which memory segment is currently stored in each cache line. That is the purpose of the tag. The tag is made of the remaining bits of the address. The cache will store a list of tags on the memory segments currently stored. When memory is accessed the cache will check the tag table to see if the memory is already cached.

Another algorithm for determining where a memory location is stored in cache is 2-way set associative. Instead of each memory address associating with one cache line, each memory location is associated with 2 cache lines. If both associated cache lines are already full then a decision must be made about which cache line will be evicted. One of the easiest ways to decide this is using the least recently used algorithm (LRU). The LRU algorithm also only requires one additional bit for each set of cache lines. 4-way set associative and higher associative caches also exists and work on similar principles.

Access Character of SDRAM

Another import aspect of memory management is the way it is accessed. Most systems today use some form of SDRAM. Most SDRAM adheres to the JEDEC standard [10]. A typical SDRAM DIMM contains several SDRAM chips. Each SDRAM chips is divided into memory banks. Banks must be active in order to access the data within them. Each bank consists of rows and columns. As with banks, rows must be active in order to access the data within them. Once a bank and row are active then a column may be accessed. This will result in the access of one bit. Several SDRAM chips can be used to access multiple bits at one time. Each access has quite a bit of delay from accessing bank, row, and column. There are techniques to decrease the access time of memory.

In page mode memory, a row can be held active so multiple columns can be accessed within the row, called a burst. A burst read will access the specified data and the sequential data following this address. This can be very useful for instruction reads and sequential data access. If an address in a different row or a different bank is needed then the current banks and row must be closed before the next access.

Memory can also be interleaved [11]. In noninterleaved memory composed of multiple banks, data will be stored sequentially in one bank then continued on into the next bank. Figure 1.2 shows noninterleaved memory.

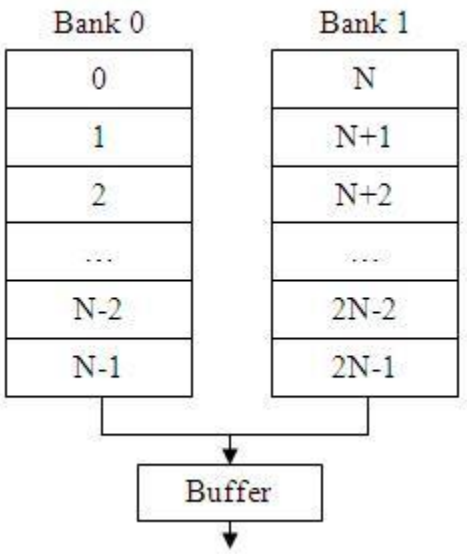

Figure 1.2 Noninterleaved Banks

After the request for data 0 is sent, the system must wait until the access is complete before it accesses data 1. While the first access is slower than subsequent accesses, there is still some latency between accesses. Figure 1.3 shows the timing for noninterleaved memory access.

The timing in Figure 1.3 and Figure 1.5 are not real timings and are meant to show the concept of non-interleaved and interleaved memory.

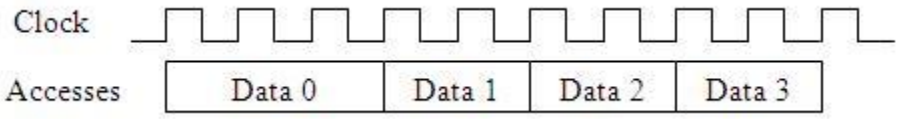

Figure 1.3 Noninterleaved Access Timing

In interleaved memory, sequential data is stored on alternating banks, with multiple banks open. Multiple buffers must be used on interleaved memory in order to access the different banks. Figure 1.4 shows how interleaved memory is organized.

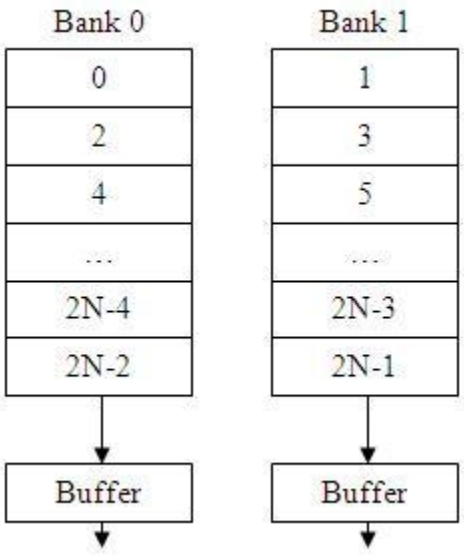

Figure 1.4 Interleaved Banks

While the memory from one bank is being accessed the request for the next address can be sent to the second bank. For example, after the request for Data 0 is sent, the request for Data

1 can be sent without waiting for Data 0 to finish. This results in a faster overall access of memory. Figure 1.5 shows the timing for interleaved access.

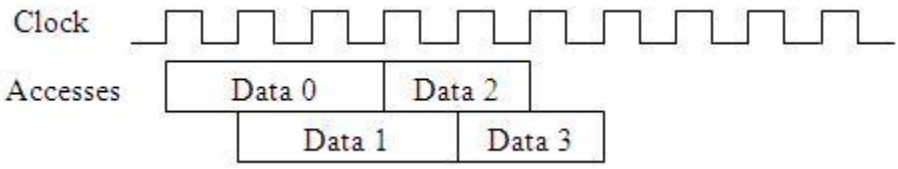

Figure 1.5 Interleaved Access Timing

As state earlier, these are not real timings, but they do show the concept of interleaved memory, and how much it can help with performance for sequential access.

Scatter-Gather

Scatter-gather is a technique that can be used to gather data from or scatter data into multiple locations. One example of scatter-gather is its use in vector processing for sparse matrices [12]. A sparse matrix is primarily composed of zero values. It is usually stored in a compacted form. The indices of the nonzero elements can be stored in order to more easily access the nonzero elements of the sparse matrix. The scatter-gather operation works well with this implementation. The gather operation will use the index vector to load the correct data from the sparse matrix. After the data is loaded the operation can be carried out by the vector processor. The values can then be stored using the scatter operation. The scatter operation uses the same index vector to update the computed values back in memory. Many recent supercomputers have these scatter-gather operations. These methods provide a faster alternative to looping through the sparse matrix in a traditional fashion.

The scatter-gather operations can also be applied at the memory control level [7]. The scatter-gather operations are used to access physical memory through indirection vectors. Normal memory access will return a burst of sequential words. For sparse matrix operations, only one of these words is used, resulting in wasted memory bandwidth. Implementing the scatter-gather operations at the memory controller level helps to reduce this wasted bandwidth. This memory controller is also able to reorganize memory addresses so that scatter-gather operations will fill a cache line resulting in more cache hits. These changes result in a significant improvement in sparse matrix calculation times. This improvement comes at the cost of increased memory controller and compiler complexity.

Chapter 2 - Cache

For this project the Virtex-5 FXT FPGA ML507 Evaluation Platform was used. This board uses the XC5VFX70TFFG1136 chip. This board was chosen based on its availability and speed. A 64-bit hardware timer was used in order to time the different operations, and an FPU was added to speed up floating point operations. A 32-bit hardware cache monitor was used in order to evaluate the cache performance. There are many different caching parameters for the MicroBlaze soft processor. The MicroBlaze is able to have an instruction cache and a data cache [13]. Each can be turned on and off independently and each can have their own settings.

Instruction Cache

The instruction cache can be used to increase performance when the instructions are located in off-chip memory. The instruction cache can be set to cache a range of the addresses within the memory address space. The instruction cache cannot cache memory located within the Local Memory Bus (LMB) address range. The base address and high address parameters change the cacheable region of memory. The addresses must be of size 2^N, where N is a positive integer. Each address is divided into two parts, the cache address and the tag. The size of the cache can be from 64 bytes to 64kB; therefore the cache address can be between 6 and 16 bits. The rest of the address is stored as the tag address. If the size of the cache is below 2kB then distributed RAM is used to implement the tag lookup. A parameter can be set to always use distributed RAM for the Tag lookup if the cache size is 8kB or less for 4 word cache lines or 16kB or less for 8 word cache lines.

Figure 2.1 shows how an instruction address is decoded. For an instruction fetch, the instruction cache will detect if the address is within its cacheable range. The cache will then check if the tag for the cache location matches. If the tag does match then it is a cache hit and the

cache will read the instruction from cached memory. If not, then it is a cache miss and the cache will request the instruction from memory over the instruction CacheLink interface. The cache line will be updated with the new memory when it becomes available.

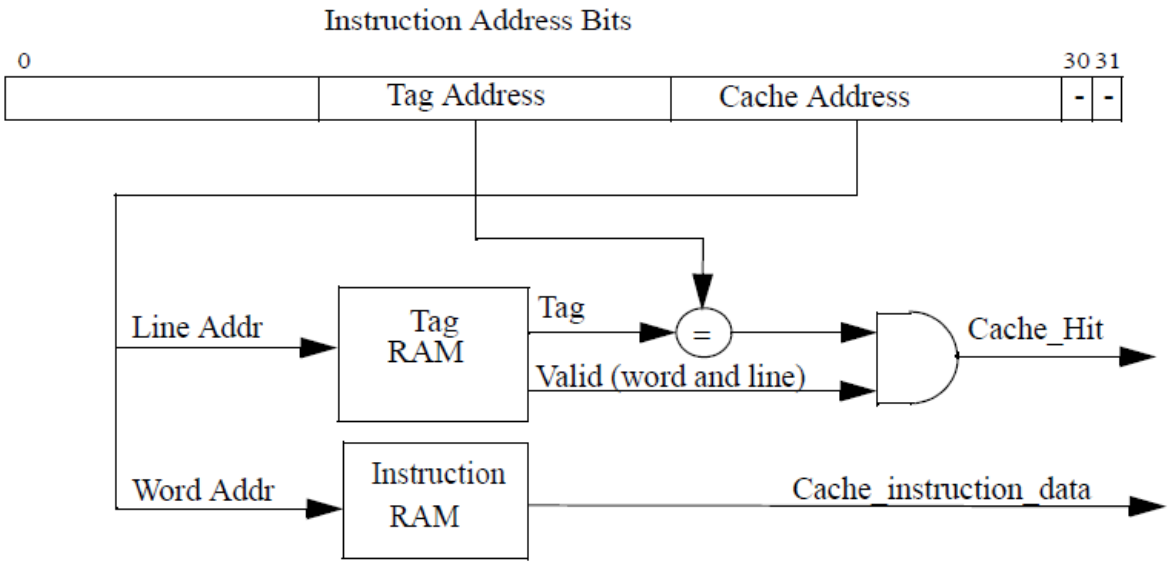

Figure 2.1 Instruction Cache Organization

Stream buffers and victim caches can also be used to increase performance of the instruction cache. If enabled the stream buffer will speculatively fetch cache lines that sequentially follow the last instruction that was fetched. If the next instruction is located in the stream buffer, the CPU will not have to wait for the instruction to be loaded from memory. This can be helpful, especially in sequential code with no loops. The victim cache will store evicted cache lines and serves as a second chance cache. The victim cache can be specified to hold 2, 4, or 8 cache lines.

The instruction cache can also be enabled and disabled in software by setting the Instruction Cache Enable (ICE) bit in the Machine Status Register (MSR). The Write to

Instruction Cache (WIC) instruction can be used to invalidate cache lines. This can be used periodically along with parity checks to avoid random bit errors.

Data Cache

The data cache can also improve performance. The data cache is very similar to the instruction cache, but it caches data instead of instructions. The data cache checks for a load instruction to determine if the memory access is for data. Figure 2.2 shows the structure of the data address is decoder.

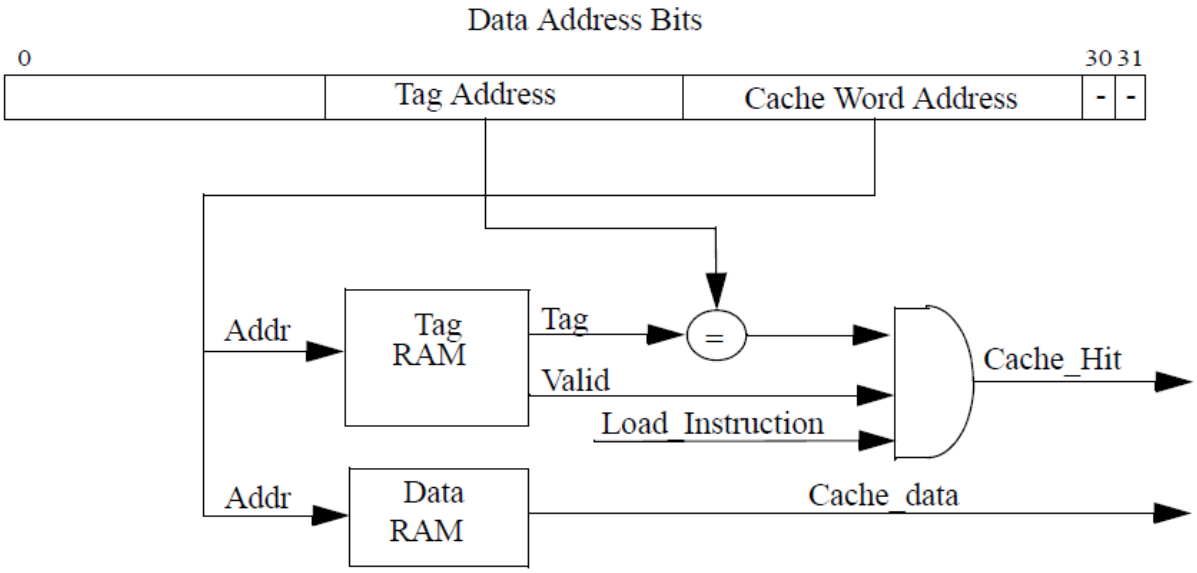

Figure 2.2 Data Cache Organization

The data cache does, however, differ from the instruction cache is a few ways. It does not include the option for a Stream Buffer, but it does have the option for a Victim Cache. The victim cache can only be enabled when the write-back policy is used.

The data cache can have two different write policies: write-back and write-through. The write-back policy will update the data in cache and only update memory when needed. If there is

a cache miss and the data already contained in the cache is dirty (it differs from main memory) then the memory will be updated with the cached value. The cache will then be updated with the new data. The write-through protocol will always update both cache and external memory, so the data in the cache will never be dirty.

Chapter 3 - Benchmarks

There are several benchmarks used for high-performance computing. While all measure performance in a similar manner, they all use different computations that will yield a difference in performance. Some of the benchmarks used in high-performance computing are LINPACK, STREAM, and RandomAccess [14]. We use HPCCG from the Mantevo project.

LINPACK Benchmark

Floating-point performance is important in HPC. Floating-point operations generally take more clock cycles to compute than other operations. Most CPUs have a dedicated floating-point unit (FPU) to deal with floating-point operations. Floating-point numbers are used in many HPC applications and are one of the limiting factors for performance. Floating-point numbers are used in HPC applications because they represent real-world elements better than integers. Floating-point numbers also have a much wider range of values than integers. The LINPACK benchmark offers a good way to test the floating-point operation performance of supercomputers. The LINPACK benchmark is used to build the Top500 list of the world's most powerful supercomputers. The LINPACK benchmark is based on the LINPACK library, which was used to analyze and solve linear equations and linear test-squares problems [15]. The benchmark has changed over the years, such as additional testing for parallel computering, but the idea of testing a systems floating-point performance still remains the same.

STREAM Benchmark

As discussed the memory wall can be a problem on any system. The STREAM benchmark assesses the performance of machines in terms of sustained memory bandwidth. The STREAM benchmark executes four simple sequential vector operations [16]. The vector length

can be set so the cache cannot hold the entire vector. This will cause the system to have many cache misses, and spend most of its time waiting for those misses to be satisfied. The STREAM benchmark will therefore show the systems sequential memory access performance.

RandomAccess Benchmark

Memory has been designed for sequential access. This helps when reading instructions and data in many applications. Some applications however will access data in a seemingly random way. The RandomAccess benchmark measures performance of random memory access. The RandomAccess benchmark will access a table at pseudo-random indices [17]. If the table is large enough then there will be cache misses and the time taken will be dominated by these misses.

HPCCG

The benchmark used for this project was HPCCG from the Mantevo project. We used this benchmark because it represents a real world problem. “HPCCG is intented to be the ‘best approximation to an unstructured implicit finite element or finite volume application, but in 8000 lines or fewer’” [18]. HPCCG incorporates many of the elements of the other benchmarks. It includes floating-point operations, sequential memory access, and random memory access. Floating-point numbers are used in the calculations. The vectors are accessed in a sequential manner, and HPCCG uses a sparse matrix, which will cause some seemingly random access. HPCCG reads a sparse matrix, right side vector, solution vector, and initial guess then solves the conjugate gradient and prints the results. A few alterations were made in order to get better and faster results. The MicroBlaze can have a 32 bit FPU. Because of this single-precision floating-point numbers were used instead of double precision. This greatly decreased the amount of time taken to solve the problem. Since different matrix sizes were used as input, the number of

iterations used to converge to the same residual would also be different. Because of this a fixed number of iterations were used. The residual at the end of each test is therefore different, but the same number of operations is used for the different sizes of matrices. Data for the different components of the conjugate gradient solve was also captured. With the conjugate gradient solve there are a number of dot products, weighted-vector adds, and sparse-matrix vector multiplications. Data cache hit rates and times were recorded for each of these parts. Using all of this data we can see how a real-world problem, HPCCG, is affected by different cache configurations.

Chapter 4 - Results

The system used for this project was the ML507 from Xilinx. Although the Virtex 5 device on the board contains a hard-wired PowerPC, we used the MicroBlaze soft processor because of its flexibility. The PowerPC has a fixed cache size of 32kB, while the MicroBlaze has a variable cache. The MicroBlaze's cache has a variable size, cache line length and write policy, which the PowerPC does not have. We were unsuccessful in implementing the write-back policy; however we were able to look into the effect of cache line length and cache size on the performance of HPCCG. We believe the write policy will have little effect on the performance of the system.

Setup

For all of the results presented we are using a MicroBlaze soft processor running at 125MHz. The MicroBlaze also contains a 32-bit FPU to improve performance. A 4kB local memory is used to contain the boot loop. All other code and data are stored on the 256MB DDR2 SDRAM SO-DIMM. A UART was used for transmitting the results. The instruction cache for the MicroBlaze was set at a size of 2kB and cache line length of 4 words for all tests. The cacheable range for the instruction cache was the address range of the DDR2 memory. Stream buffers and victim caches were not used for the instruction cache. We kept the instruction cache consistent for all tests because the instruction cache consistently had hit rates around 99%. This gave us very little interesting data for the instruction cache. We focused on reconfiguring the data cache instead, which gave varied results. The data cache varied in size and cache line length. The cacheable range for the data cache was also set to be the address range of the DDR2 memory.

Two custom Intellectual Property (IP) cores were used to monitor performance. A 64-bit timer was designed in order to keep track of the computation time for the different tests. A 64-bit timer is needed because a 32-bit timer will roll over after 34.36 seconds. A 64-bit timer will provide more than enough time, rolling over after 4676.34 years. The timer simply consists of a counter, which will increment at the positive edge of the clock. Since the bus interface between the timer IP and MicroBlaze is 32-bits wide, two successive reads are needed in order to read the lower and upper halves of the 64-bit counter. Because of this an error could occur when reading the second set of 32 bits. For example, the lower 32 bits are read first. Before the upper 32 bits are read they are incremented, reporting an incorrect time to the processor. To prevent this error the upper 32 bits are registered when the lower 32 bits are read. When the processor first reads the timer, the lower 32 bits are sent over the bus, and the upper 32 bits are stored in a separate register. When the processor reads the timer again the registered upper 32 bits are sent over the bus. The software must then convert the counter reading into a time. This can be easily done by dividing by the bus frequency. The timer can be reset by writing to the timer IP, but this feature is not used in this project.

The other custom IP is a cache counter. The cache counter will track cache accesses and hits. The MicroBlaze can be set up with a Trace bus, and the relevant signals from the Trace bus routed to the cache counter IP. The cache counter is able to keep track of cache requests and hits for both the instruction and data caches. For the data cache, the cache counter IP will first check if the address being accessed is within cacheable range. The data request counter is incremented if there is a data request in cacheable range. The data hit counter is incremented if there is a data hit for a data request in cacheable range. The counters for the instruction cache are similar; however it is assumed that the instruction request is within cacheable range since all of the code

is stored on the DDR2 SDRAM. This IP will also keep track of request and hits when there is a valid instruction. This is useful for the data cache counters, since a data request should only happen with a valid instruction.

In order to access the different counters the user must specify which counter value he wants. This can be done by writing to the cache counter IP. Table 5.1 shows what values will be accessed from reading the IP after a value is written.

Value Written to IP	Value Read from IP
0	Data Requests
1	Data Requests during Valid Instructions
2	Data Hits
3	Data Hits during Valid Instructions
4	Instruction Requests
5	Instruction Requests during Valid Instructions
6	Instruction Hits
7	Instruction Hits during Valid Instructions
All Other Numbers	0

Table 4.1 Cache Counter Read Output

For example, if a 0 is written to the IP then the total number of data requests will be sent upon a read. Some of these counts were only used during testing and only a few of them are used in the actual results. The data requests during valid instructions, data hits during valid instructions, instruction requests, and instruction hits are the only values used in the results.

A few changes needed to be made in order to run the chosen benchmark (HPCCG). The MicroBlaze is limited to a 32 bit FPU; therefore, it was decided that single-precision floating-point numbers would be used. This greatly reduced the amount of time needed to run the tests. Similar relationships and trends should be seen for double-precision floating-point numbers.

Since there is not a system timer, the mytimer function had to be rewritten to read the count values from the timer IP and convert then to a time. The conversion of the time from a count to a floating-point value of time will add a small amount of overhead into each call to the mytimer function. Instead the function could be changed to return the counter value to store for later computation. However we determined that the complexity of changing the behavior of the function was not worth the small amount of error the overhead would incur.

The Benchmark, HPCCG, also uses command line arguments. This is not possible for the MicroBlaze. Therefore, we set up the variables at the beginning of the code according to which test we wanted to run. The program took the problem size and the type of sparse-matrix storage that would be used as arguments. The type of sparse-matrix storage was compressed row storage by default, and was not changed in this study. In order to run tests efficiently we created a loop that would run HPCCG repeatedly for different problem sizes. The problem size varied from 1x1x1 to 64x64x64. Running larger test would have been preferred, but there was not enough SDRAM for sizes larger than 64x64x64. For each successive test the problem size was doubled. First the x-dimension was increased, then the y-dimension, and finally the z-dimension. So the problem went from 1x1x1 to 2x1x1 to 2x2x1 to 2x2x2 to 4x2x2 and so on until the problem size reached 64x64x64.

Finally, the cache counter also needed to be read in order to keep track of the cache hit rates. Timers had already been set up to keep track of the total times and the times for the major components of the conjugate gradient solve: dot product (DDOT), weighted-vector add (WAXPBY), and sparse-matrix vector multiply (SparseMV). The timer calls were already wrapped around each of these individual function calls, as well as the total conjugate-gradient solve. Calls to read and store the cache counters were simply added around these timer calls. A

loop reads and stores all the cache counters before the start-timer call. After the function completes and the stop-timer call, a loop reads all the cache counters and computes the difference and adds that to the total number. The rates are later calculated when the results are sent out via the UART.

Cache Line Length

The MicroBlaze has two options for cache line length: 4 or 8 words. A word is 32 bits, so cache lines can be either 128 or 256 bits. In order to test this we used a 2kB cache with the write-through policy. All other cache options were left on the default settings. Figure 4.1 shows the total data cache hit rate for various problem sizes. The 8-word cache lines outperform the 4-word cache lines up until a certain problem size. At this point the hit rates for both 4-word and 8-word cache lines drop; however the hit rate for the 8-word cache lines drop below the 4-word cache line.

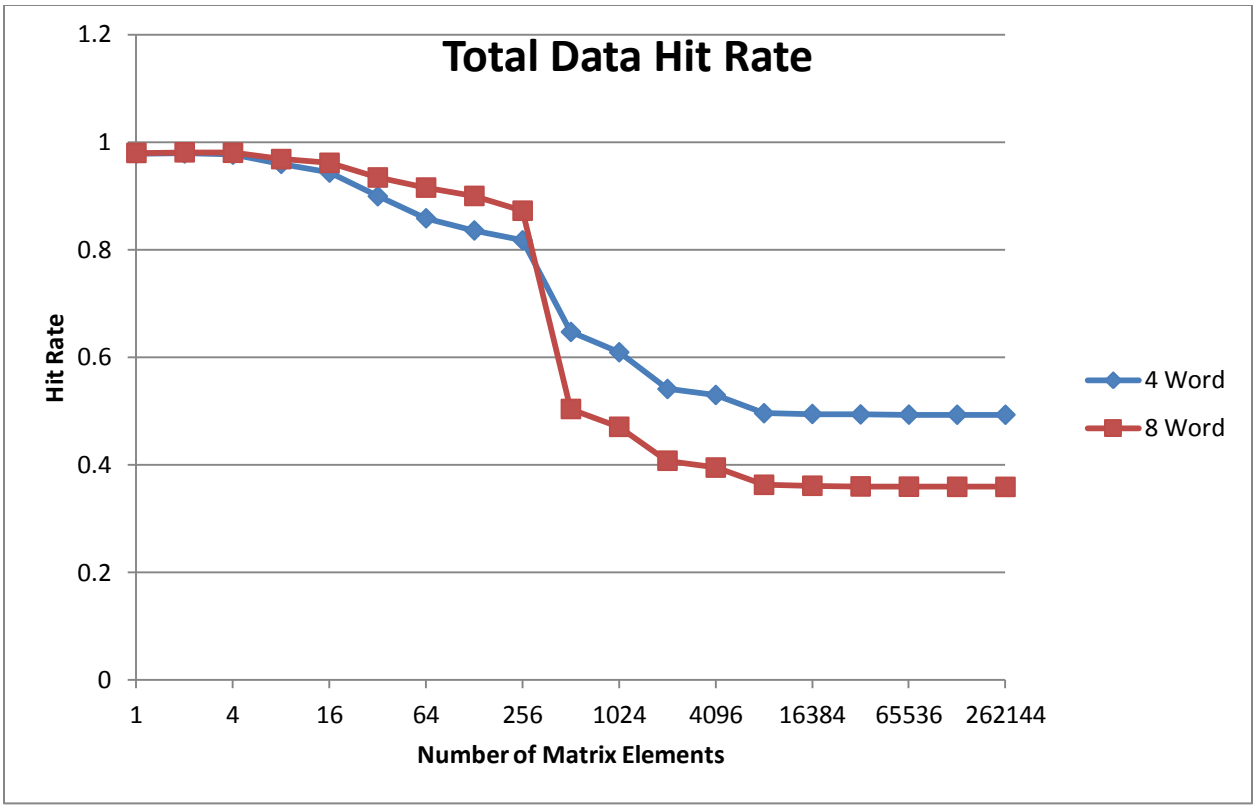

Figure 4.1 4-Word vs. 8-Word Cache Line (2kB, Write Through)

The hit rates for DDOT, WAXPBY, and SparseMV have similar trends shown in Figure 4.2. The difference between the 4-word and 8-word cache lines is more evident in the WAXPBY and SparseMV.

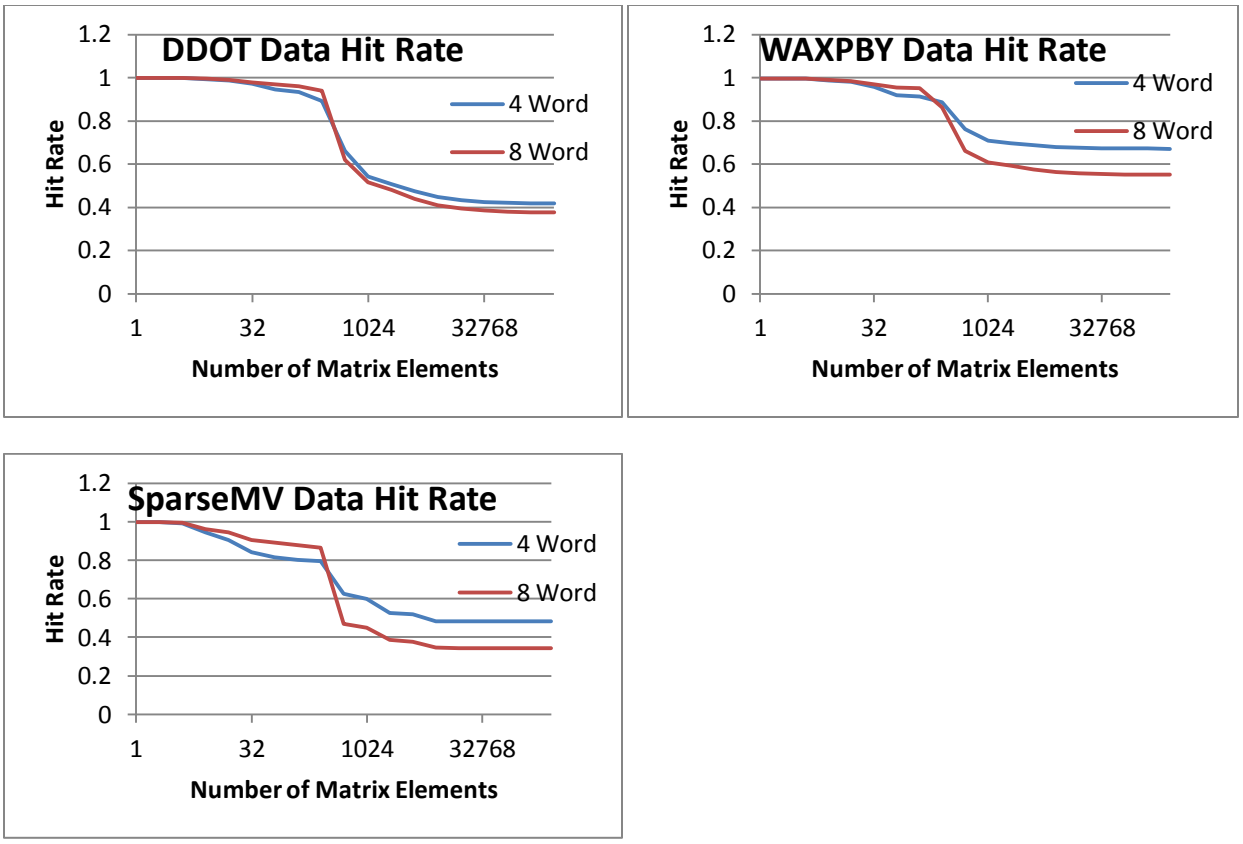

Figure 4.2 DDOT, WAXPBY, and SparseMV Hit Rates

Figure 4.3 shows the times for the total conjugate-gradient solve and for each of the individual operations. Although the times for 8-word cache lines are better for a smaller problem size, it does not show well on the linear scale. It is important to note that as the problem size increases, performance of the 4-word cache line becomes better than performance of the 8-word cache line. The 8-word cache lines retain more data from a read to main memory. Because of this the 8-word cache lines perform better at smaller problem sizes. Most of the 8 words in the cache line will get hits. When the problem size gets large enough, data will no longer fit into cache. This will cause cache misses no matter what the cache line size. Since 4-word cache lines are smaller, there is room for more nonsequential data to be in cache at one time. This is believed to be why 4-word cache lines to get more hits than 8-word cache lines.

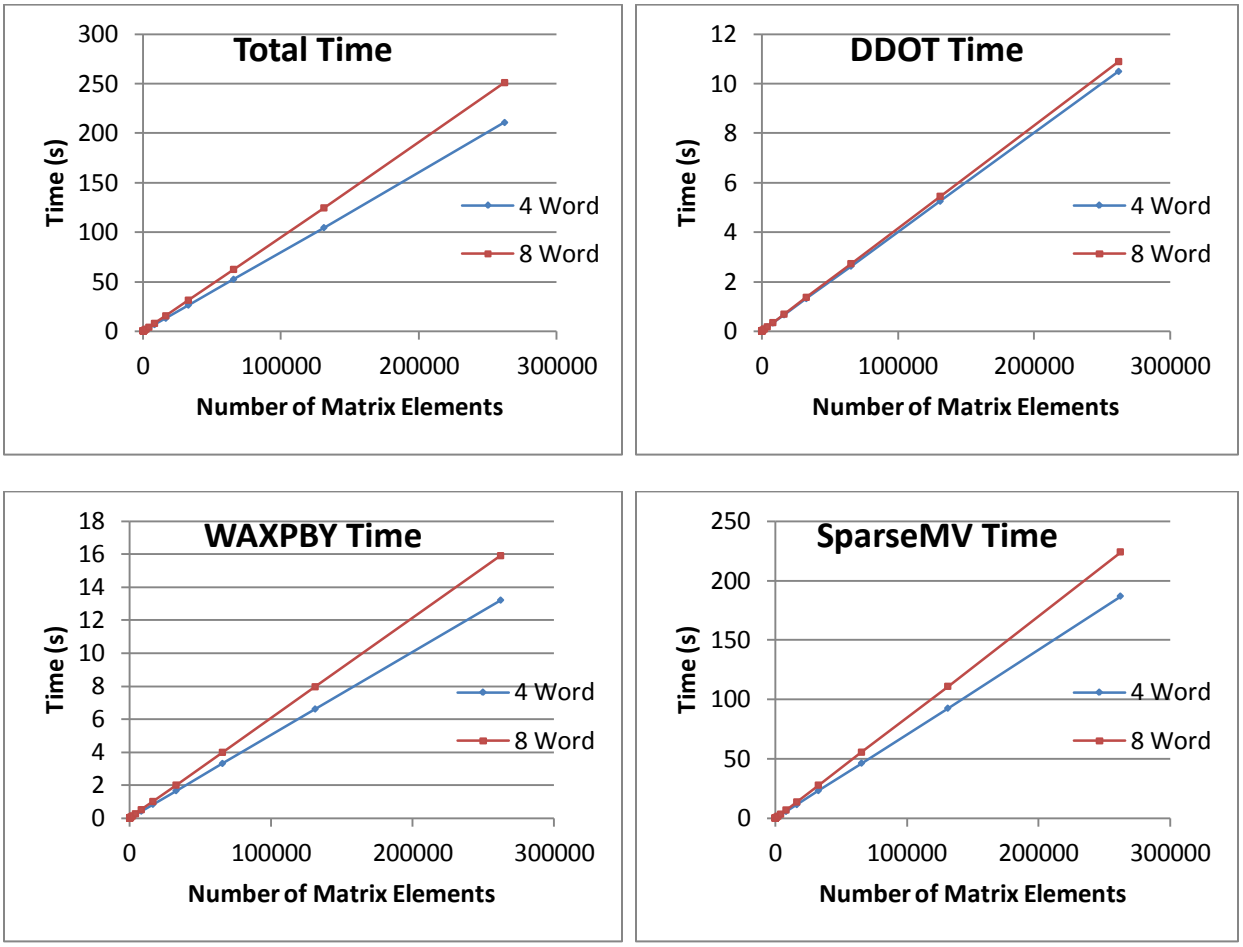

Figure 4.3 Total, DDOT, WAXPBY, and SparseMV Times

Since most testing time is spent solving the larger problems, we used 4-word cache lines for further tests. This will give us a lower total run time for the benchmark.

Individual Operation Performance

Next we compared the data hit rates of the different operations. Figure 4.4 shows the total, DDOT, WAXPBY, and SparseMV hit rates for different problem sizes. Again, a 2kB data cache was used. There is a significant drop in data hit rate once the problem reaches a certain size. This drop happens for all of the different operations. The DDOT has the biggest drop in hit

rate. This could be due to the dot product operation having no reuse once the problem size gets too large. In other words, data brought into the cache will be replaced before it is used again. The WAXPBY and SparseMV operations have some reuse, so the drop in hit rate is not as severe. The fact that the total hit-rate curve follows the SparseMV curve so closely shows that most of the data accesses in the conjugate-gradient solve come from the SparseMV operation.

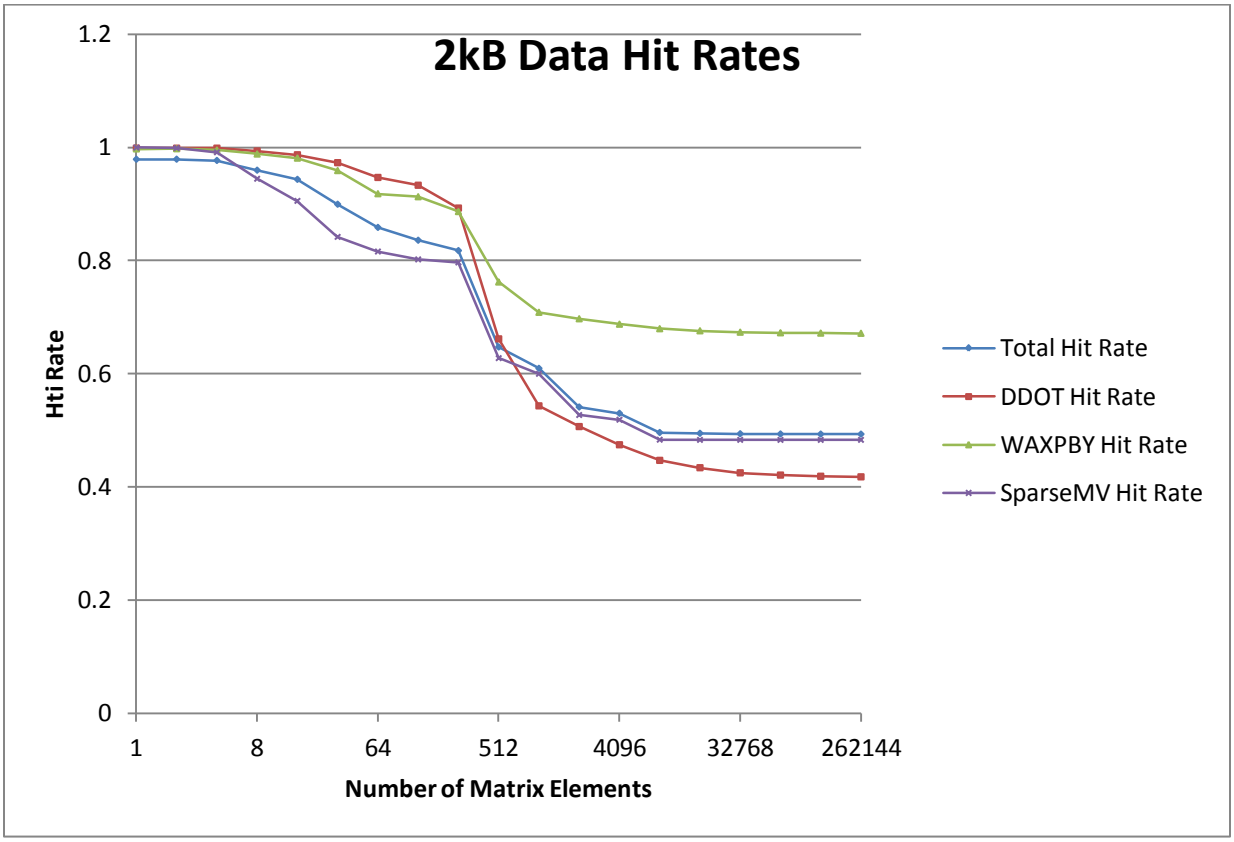

Figure 4.4 Data Hit Rates for Individual Operations

Cache Size

Finally we looked at the performance of different sized caches. Figure 4.5 shows the total data hit rates for sizes from 64B to 64kB. At the smallest problem sizes the hit rates are tightly

grouped once the cache size reaches 512 bytes. As the problem size increases hit rates for the smaller caches drop off. Finally at the largest size the hit rates become tightly grouped again for most of the cache sizes. At the largest problem size the 32kB and 64kB caches do not fit into this tight grouping. We believe that if test were running on larger problem sizes the 32kB and 64kB cache hit rates would eventually drop to a level similar to the other cache sizes. We were not able to verify this because amount of memory on the system limited the size of problem we could run.

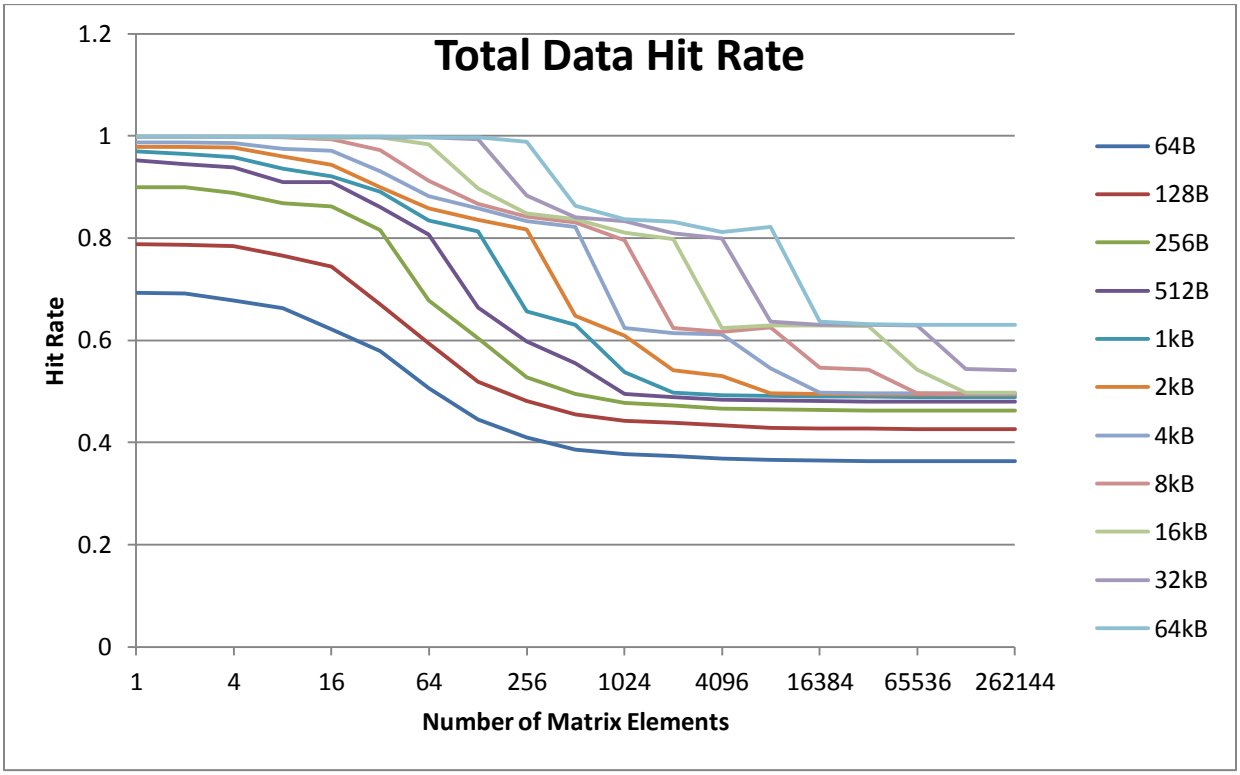

Figure 4.5 Total Data Hit Rate for Varying Cache Size

The data hit rates for DDOT and WAXPBY show slightly different trends. The DDOT and WAXPBY trends for small problem sizes are similar to the trends for the entire conjugate-gradient solve. The DDOT and WAXPBY have a significant drop in hit rate when the problem size reaches a quarter of the cache size. This drop in hit rate is most likely due to an array or

matrix that will no longer fit in cache. When accessing this data the cache must evict cache lines once it becomes full, increasing the number of misses and lowering the hit rate. This drop in hit rate is less evident for small caches. In the program there is data that increases in size with increased problem size and static data that stays at a constant size for every problem size. We believe that when the cache size is small enough, the static data has more of an effect on the hit rates and the decreases in hit rate become more gradual.

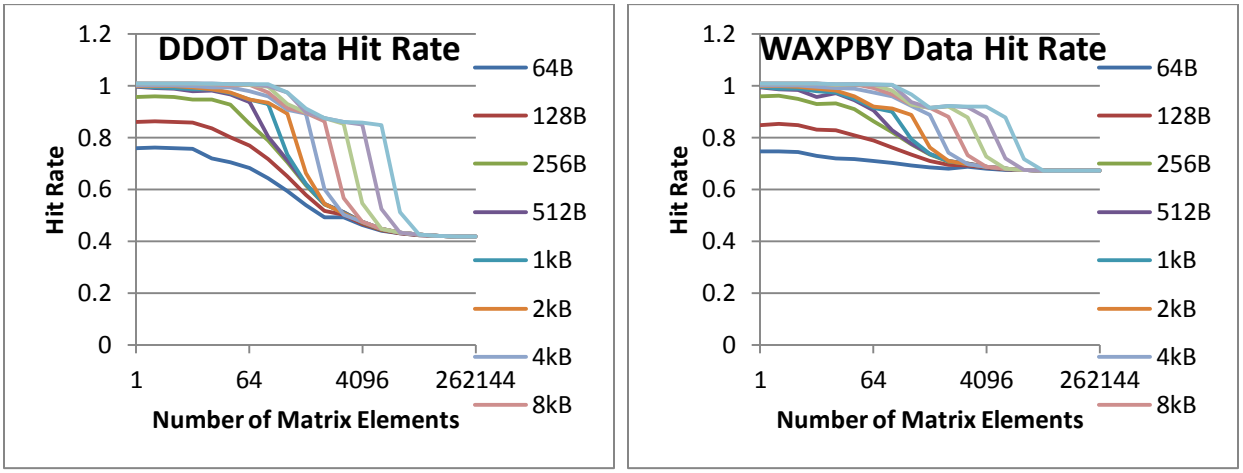

Figure 4.6 DDOT and WAXPBY Data Hit Rates for Varying Cache Sizes

Figure 4.7 shows the data hit rates for the SparseMV function. The trends in the total hit rate follow the SparseMV hit rate closely. We expect this since the number of data accesses in SparseMV is far greater than the data accesses in either DDOT or WAXPBY. For SparseMV there are three different "levels" for the hit-rate curve. In between these levels there is a significant drop in hit rate.

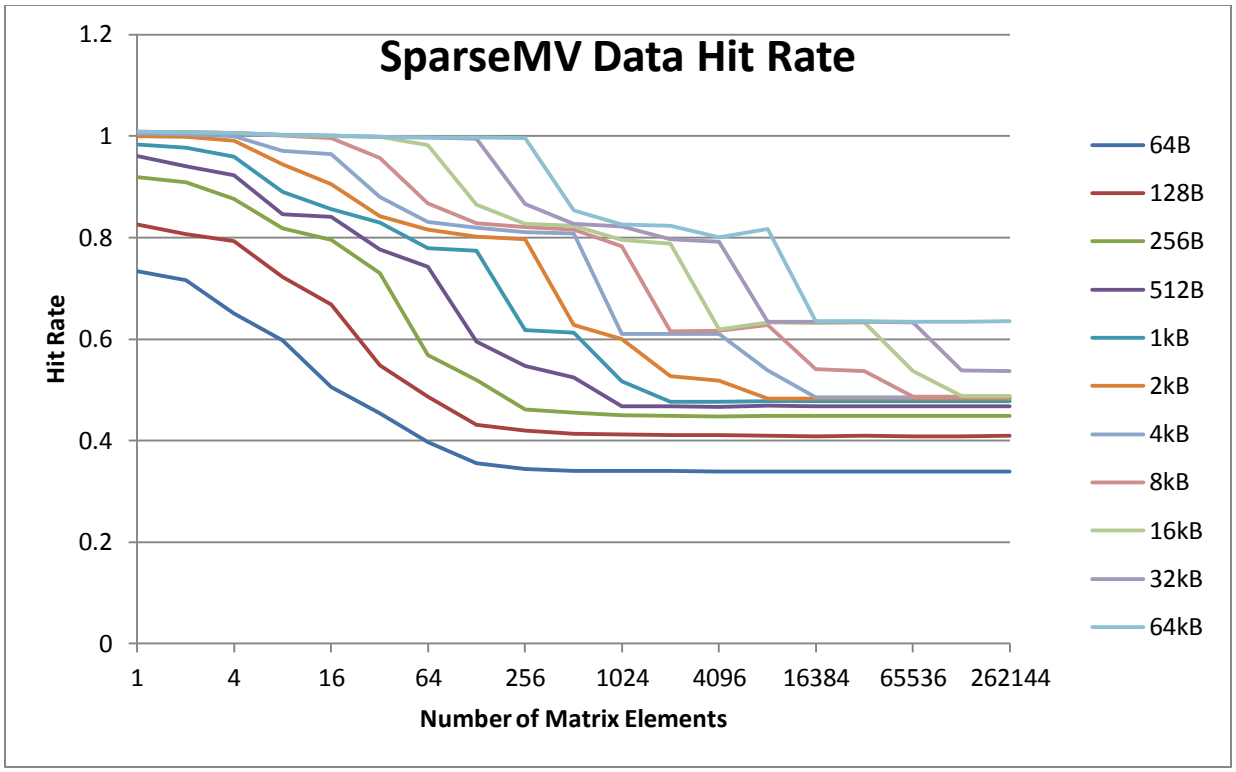

Figure 4.7 SparseMV Data Hit Rates for Varying Cache Sizes

We believe the first drop in hit rate occurs when most of the data will no longer fit in the cache. Most of the data accessed comes from five different variables: r, p, Ap, x, and A. The variables r, p, Ap, and x all have a length equal to the number of matrix elements, n. A is the sparse matrix that the program generates. Within the sparse matrix there are several vectors accessed. There are three vectors that are n-elements long. There are two more vectors accessed which have a length equal to the number of nonzero elements in the sparse matrix, nnz. Since each number is 32 bits, we can multiply the total by 4 to get the number of bytes. To calculate the total number of memory needed we can simply compute:

Table 5.2 shows the total memory needed for the different problem sizes. After computing the total memory needed for the different problem sizes, we see that the first

significant drop in hit rate occurs when the total memory can no longer fit in cache. For example, the 16kB cache has a data cache hit rate of at least 98% for problem size 1 through 64. When the problem size increases from 64 to 128 the data hit rate drops from 98.2% to 86.5%. Table 5.2 shows that the total memory needed for a problem size of 64 is 12.8kB which will fit into the 16kB cache. For a problem size of 128 the total memory needed climbs to 25.6kB which will no longer fit into the 16kB cache. When the cache is full, some old data will be evicted to make room for the new data. Some of the data could be the data needed for the SparseMV operation. This will lead to fewer cache hits. The calculation for total memory does not take into account the static memory that is needed for every problem size. This is likely the reason the drop in hit rate becomes less significant for small cache sizes. The access of the static data becomes dominant over the dynamic data for small problem sizes.

n	1	2	4	8	16	32	64	128	256	512	1024
nnz	7	16	36	130	262	550	1414	2830	5710	12622	25246
Total Memory	84	184	400	1264	2544	5296	13104	26224	52848	115312	230640

n	2048	4096	8192	16384	32768	65536	131072	262144
nnz	50590	105886	211774	423742	866110	1732222	3464830	7003774
Total Memory	462064	961776	1923568	3848688	7846384	15692784	31388656	63370224

Table 4.2 Total Memory Needed for Different Problem Sizes

The second drop in the hit rate for SparseMV occurs for almost of the cache sizes. This drop occurs when the problem size reaches a quarter of the cache size. This is consistent with DDOT and WAXPBY data hit-rate drops. Since the memory required for many of the data arrays accessed is four times the size of the problem, we thought that the drop in performance could be due to one of these arrays no longer fitting into cache. Several tests were designed to test this theory. None of them proved that this was the case. In order to fully understand what is

occurring, further study must be performed in order to determine which portion of data is no longer getting cache hits. In order to do this, the code and cache requests and hits must be looked at closely. This is also the case for the third drop and final plateau.

The last performance indicator we looked at was the time it took for HPCCG to complete for the different cache sizes. We were also able to include results for no cache. Although this test did not have a data cache, it still had an instruction cache in order to compare it to the rest of the tests. Figure 4.8 shows the total time taken to solve the conjugate gradient for varying cache sizes. The number of matrix elements is shown on a linear scale in contrast to the base-2 logarithmic scale used for the hit rate. This better shows the linear trend for time increase as the problem size increases. However this hides what is occurring for the smaller problem sizes. Figure 4.9 shows the times for some of the smaller problem sizes. When comparing these two figures to hit rate, a significant drop in hit rate corresponds to a significant increase in slope of the time taken. From these figures we can also see that having a cache of any size helps significantly. This is likely because of how tightly coupled SDRAM and the cache are. When the processor requests data the memory controller must bring the data in from the SDRAM. If there is a cache, the cache line will be filled and the data given to the processor. If the next data request is sequential to the previous request there will likely be a hit in the cache and SDRAM will not need to be accessed. If there is no cache the SDRAM will need to be accessed again. This can be seen in the hit rates in Figure 4.5. Even for a 64 byte cache and a problem size of 256k, the hit rate is about 36%. The large difference in time shows that SDRAM access time makes up a large portion of the total time.

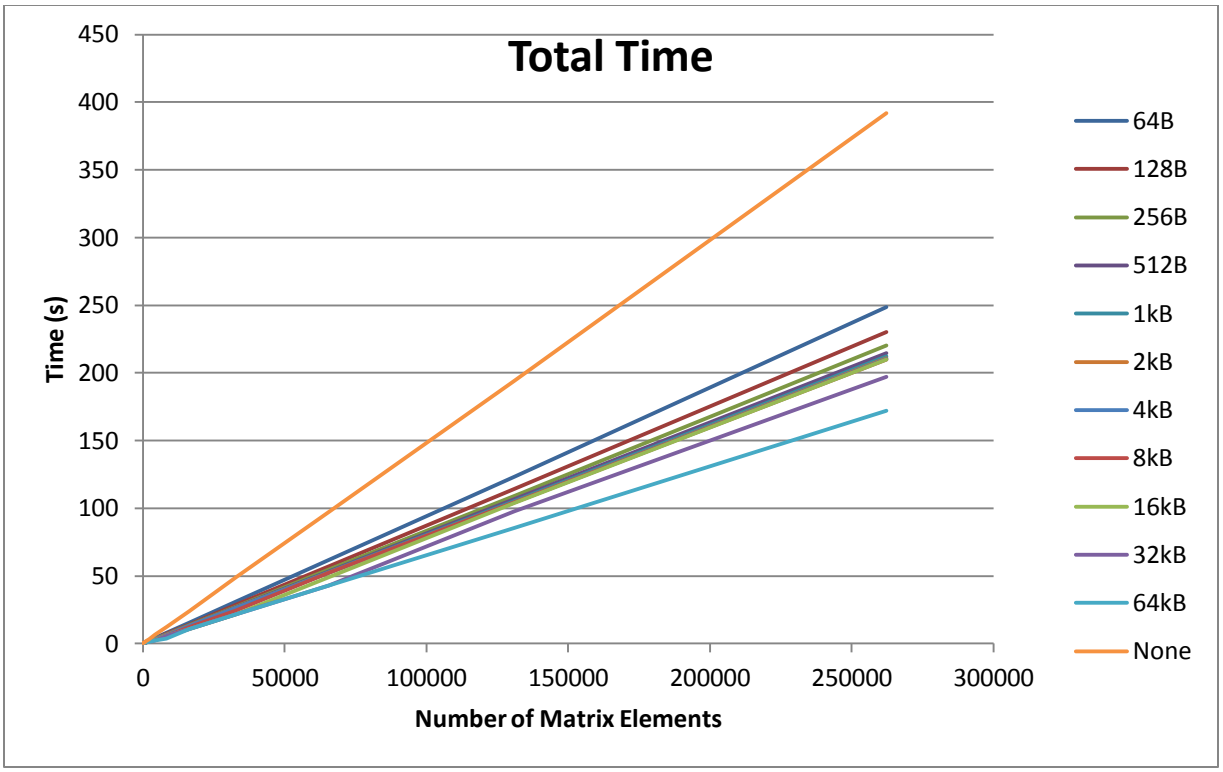

Figure 4.8 Total Time for Varying Cache Sizes

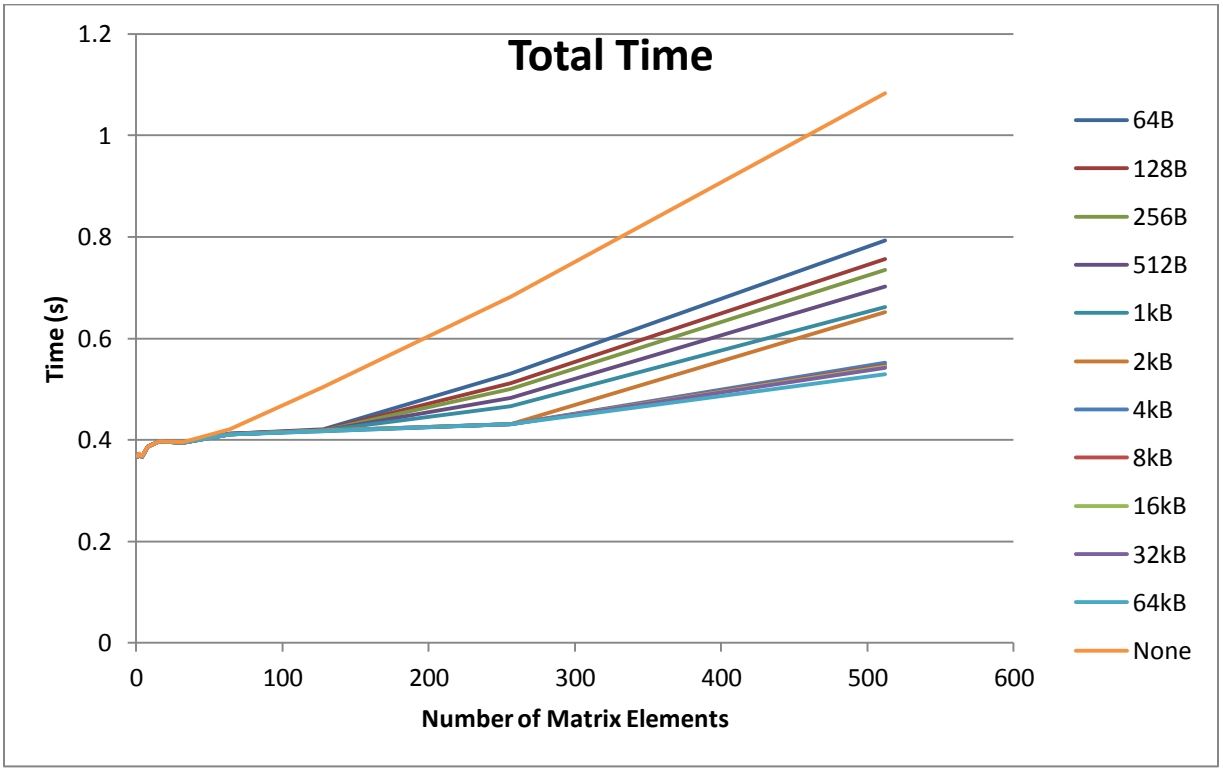

Figure 4.9 Total Time for Varying Cache Sizes (Limited Problem Size)

Figure 4.10 shows the times for DDOT, WAXPBY, and SparseMV. Again the times correlate to the hit rate. The DDOT and WAXPBY have similar trends. At the larger problem sizes the times taken are all about the same as long as there is a cache. No cache shows a significant increase in time. The SparseMV time is similar to the total time. This shows that the time taken to compute is dominated by the time taken to compute the SparseMV.

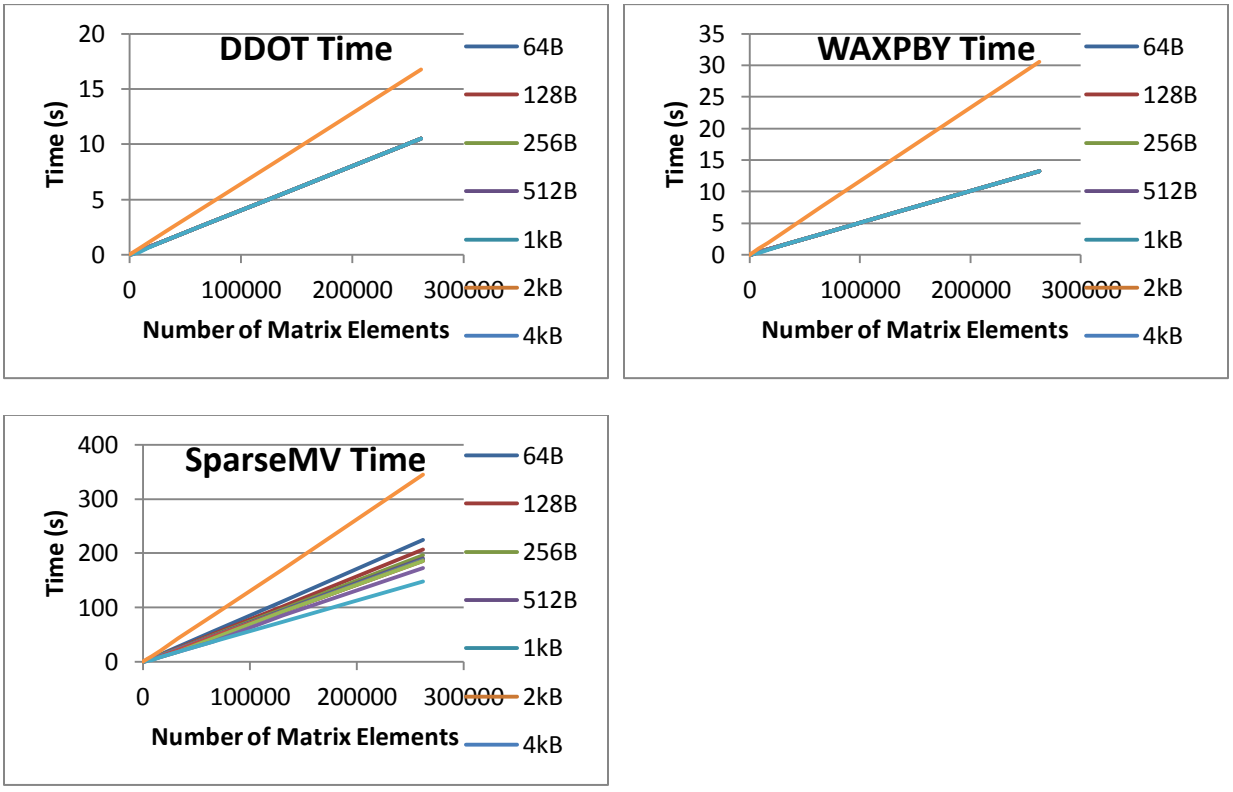

Figure 4.10 DDOT, WAXPBY, and SparseMV Times for Varying Cache Sizes

Chapter 5 - Conclusions

There have been many attempts to overcome the memory wall. The use of an effective cache is a popular way of mitigating this problem. Different caching schemes can be effective for different problems. This project looks into the effect of cache for high-performance computing problems. The access character of SDRAM, which is optimized for sequential data, can cause many of these programs to perform poorly. FPGAs provide a way to look at different cache configurations using a real system. Many systems are tested using a synthetic benchmark. This project uses a real-world problem, conjugate gradient, to characterize the performance of different cache configurations.

The performance of the MicroBlaze is limited, but the flexibility of configuration makes it a good test platform. Hardware can also be added to monitor performance, alleviating some of the work of the processor. From the two options for cache line length, we determined that a cache line length of 4 words performed better than a cache line length of 8 words. A cache line length of 8 words would only be better if operating on very small problem sizes.

After looking into the different cache sizes, the best size is dependent on the problem size. If working on a very small problem size many of the cache sizes' performance was similar. Although the 64kB cache always performed the best, the 512B cache had similar performance for problem sizes up to 16. Because smaller caches mean less cost and lower power consumption a 512B cache may be the optimal size for problems containing 16 or less elements. As the problem size increases the performance drops off. The performance drops off at different points for different cache sizes. This makes it difficult to determine an optimal cache size. Unless the problem size is defined for all programs that will run on the processor it may be best to choose a cache size that performs adequately for most problem sizes. A cache size of 2kB or 4kB has

good performance for a problem size between 16 and 16384. Finally, at the largest problem sizes the difference in performance for the different cache sizes becomes minimal. The 32kB and 64kB cache sizes performance does not fall to the level of the smaller caches most likely because we were not able to run problems of a sufficient size. If the performance of these two sizes were to drop to the same level as the other caches, there would be little difference between the 512B cache and the 64kB cache. As with the smaller problem sizes, the optimal cache size for large problems may be around 512B.

Future work can be done to better characterize the results of this study and to further investigate the performance of different cache parameters. The plateaus and drops for data cache hit rate need to be better defined. A deeper look into what data is causing the cache misses could show why those plateaus occur and what could be done to improve the cache performance. The caches of the MicroBlaze were direct-mapped. Further research could be done to investigate the effect of set-associative caches.

References

[1] W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, Vol. 23, No. 1, pp. 20-24, Mar 1995.

[2] M.V. Wilkes, "The Memory Wall and the CMOS End-Point," *Computer Architecture News*, Vol. 23, No. 4, pp. 4-6, Sep 1995.

[3] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K., "A Case for Intelligent RAM," *Micro, IEEE*, vol. 17, no. 2, pp.34-44, Mar/Apr 1997.

[4] Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, Vol. 21, No. 8, pp. 613-641, 1978.

[5] Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D., "Memory Access Scheduling," *ISCA*, 27, 2000.

[6] Nowatzyk, A., Fong Pong, Saulsbury, A., "Missing the Memory Wall: The Case for Processor/Memory Integration," *1996 23rd Annual International Symposium on Computer Architecture*, pp. 90-101, May 1996.

[7] Lixin Zhang, Zhen Fang, Parker, M., Mathew, B.K., Schaelicke, L., Carter, J.B., Hsieh, W.C., McKee, S.A., "The Impulse Memory Controller," *IEEE Transactions on Computers*, vol. 50, no. 11, pp. 1117-1132, Nov 2001.

[8] Mengxiao Liu, Weixing Ji, Zuo Wang, Jiaxin Li, Xing Pu, "High Performance Memory Mangagement for a Multi-core Architecture," *Ninth IEEE International Conference on Computer and Information Technology*, vol. 1, pp. 63-68, Oct 2009.

[9] D.A. Patterson and J.L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface," Morgan Kaufmann, Burlington, MA, 2009.

[10] *DDR2 SDRAM Specification*, JESD79-2F, Nov 2009.

[11] D. Adams. (1993, Nov 6). *Dale Adams on Interleaved Memory on Centris 650 & Quadra 800* [Online]. Available: http://www.ralentz.com/old/mac/hardware/dale-adams/memory-quad8-cent650.html

[12] J.L. Hennessy and D.A. Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann, San Francisco, CA, 2007.

[13] *MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 13.4*, UG081, Jan 2012.

[14] (2012, May 4). *HPC Challenge* [Online]. Available: http://icl.cs.utk.edu/hpcc/

[15] J. Dongarra. (2007, May 8). *Frequently Asked Questions on the LINPACK Benchmark and Top500* [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/faq-linpack.html#_Toc27885709

[16] *STREAM Benchmark Reference Information* [Online]. Available: http://www.cs.virginia.edu/stream/ref.html

[17] D. Koester and B. Lucas. *RandomAccess Rules* [Online]. Available: http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/

[18] M. Heroux and R. Barrett. *Mantevo-Packages* [Online]. Available: https://software.sandia.gov/mantevo/packages.html