CS505 Project Part 1 Report

Bill Katsak, wkatsak@cs.rutgers.edu
Binh Pham, binhpham@cs.rutgers.edu
Rick Ramstetter, rick.ramstetter@gmail.com

1 Project Description

"Based on our lectures on caches, code up a 2-level cache simulator and hack Bochs to fit your cache simulator inside. In other words, you need to hack the Bochs source code to find where the memory accesses are conducted. When you isolate these accesses, you need to route them to your cache simulator and figure out whether these accesses would results in cache hits and misses. While Bochs is a purely functional simulator (meaning that it does not track timing information), your cache simulator should faithfully provide some notion of cache miss latencies. Once your simulation platform is up and running, run gcc, mcf, and libquantum from the SPEC2006 suite (these are provided to you as the rest of the document will explain) on Bochs with your cache simulator. Your job is to show, for each of these benchmarks, the impact of cache parameters such as total size, line size, associativity, and miss latencies on cache miss rates and average memory access times. In doing so, you should be able to provide some intuition about how the nature of the benchmarks (are they memory-intensive, cpu-intensive etc.) affects your cache hit rates. As part of your 3-page writeup, you must explain very briefly your coding strategy, and your cache parameter variation results for the three benchmarks. Remember that the bulk of your writeup should focus on your results." [**?**]

2 Introduction

Our project implements a series of delays atop Bochs, and we attempt to use these delays to simulate the caching system described in 1. As Dr. Abhishek Bhattacharjee stated in class, our caching system does not need to (and, indeed, does not) actually hold data. Rather, because Bochs is a functional only simulator, we have assumed that memory accesses are always immediately available (in particular, we assume this holds true for unmodified Bochs).

3 Trade-offs explored

In testing the performance of our caching system, we altered three cache parameters: (1) cache size, (2) associativity, and (3) block size. A summary of our expected results after changing each of these parameters follows:

1. Cache size: We expect that increasing cache size will always decrease miss rate. We anticipate that the 2:1 associativity rule (presented in class) will be followed. Thus, we expect a 64kb one-way cache to perform as well as a 32kb two-way cache.

```
memory_access_delay(address)  {
   /* do nothing if the address is in L1 cache */
   if (in_L1_cache(address)) return;

   /* small delay if the address is in L2 cache */
   if (in_L2_cache(address)) {
       delay(10);
       bring_address_to_L1_cache(address);
       return;
   }

   /* huge delay if the address is not in either L1 or L2 */
   /* this is our simulated memory access delay */
   delay(1000);
   bring_address_to_L2_cache(address);
   bring_address_to_L1_cache(address);
   return;
}
```

Figure 1: Psuedocode explaining delays introduced by our caching system. A call to our actual code's functional equivalent of the above *memory_access_delay()* is made wherever Bochs would normally write to or read from memory. In our code, these occur in the few files under the "bochs-2.4.5/memory" folder.

2. Associativity: We expect that increasing associativity will always decrease miss rate. We anticipate that the 2:1 associativity rule (presented in class) will be followed. Thus, we expect a 64kb one-way cache to perform as well as a 32kb two-way cache.

3. Block size: We anticipate that an increased block size will aid performance in benchmarks which have heavy spatial locality characteristics. We anticipate that a decreased block size will aid performance in benchmarks which have heavy temporal locality characteristics, as this will allow for more cache lines.

For each of the benchmarks tested, we used the "test" data set. Bochs exhibited extremely poor performance on three test systems (including two with a Core i7 processor), and thus we were unable to utilize other data sets without incurring significant time overhead. We chose instead to focus our time on understanding the internals of Bochs and evaluating the results of our benchmark tests.

4 Benchmark performance

In this section, we test the three relevant SPEC benchmarks (429.mcf[**?**], 403.gcc[**?**], and 462.libQuantum[**?**]) under our caching system. We expect their performance relative to one another to be roughly equivalent to the relative performance values found by [**?**]. The parameters we vary during testing are listed at 3.

4.1 Environment and assumptions

For comparison's sake, we first run all benchmarks under a Bochs simulator which always delays. With these results, we hope to establish the baseline performance of a CPU completely lacking a caching system (and thus requiring a memory fetch on every read).

Our Bochs modifications continually print (every $100,000$ memory accesses) a total of the number of cache misses occurring within that most recent period. In addition to establishing a cache miss rate, this allows us to filter cache misses associated with the operating system running inside Bochs. Before executing any benchmark, we take note the number of cache misses that have occurred (e.g. at the fault of the operating system) and, after the benchmark concludes, we filter these misses. Unfortunately, this yielded poor results for the $462.libQuantum$ benchmark, which has an overall low ratio of memory access instructions to total instructions.

We do not (ever) vary the L1 cache size or associativity, which is held at a direct-mapped 64kb. We assume that the L1 cache block size is always equivalent to the L2 cache block size.

4.2 Results for varying L2 cache size

See figures 2, 3, 4, and 5.

Test name	With 256kb L2	With 512K L2	With 1024K L2
403.gcc	1.4%	0.7%	0.6%
462.libQuantum	0%	0%	0%
429.mcf	13.6%	12.4%	9.7%

Figure 2: Cache performance varying the L2 cache size. Expressed as a percentage of memory accesses generating a cache miss. Associativity is held constant at 2-way, block size is held constant at 64 bytes.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	2	131	12
462.libQuantum	0	2	1
429.mcf	108	765	26

Figure 3: Cache misses, memory reads, and memory writes for a **256kb**, 2 way associative cache with a block size of 64 bytes.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	2	267	28
462.libQuantum	0	1	0
429.mcf	91	717	19

Figure 4: Cache misses, memory reads, and memory writes for a **512kb**, 2 way associative cache with a block size of 64 bytes.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	1	159	64
462.libQuantum	0	0	0
429.mcf	77	763	29

Figure 5: Cache misses, memory reads, and memory writes for a **1024kb**, 2 way associative cache with a block size of 64 bytes.

4.3 Results for varying associativity

See figures 6, 7, 8, 9 and 10.

Test name	2-way L2	4-way L2	8-way L2	16-way L2
403.gcc	1.4%	2%	1.8%	1.4%
462.libQuantum	0%	0%	0%	0%
429.mcf	13.6%	13.4%	13.6%	11.3%

Figure 6: Cache performance varying the cache associativity. Expressed as a percentage of memory accesses generating a cache miss. Cache size is 256K, block size is held constant at 64 bytes. We include the data for a 16-way associative cache to highlight an interesting trend discussed in this paper's "Problems Encountered" section.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	2	131	12
462.libQuantum	0	2	1
429.mcf	108	765	26

Figure 7: Cache misses, memory reads, and memory writes for a 256kb, **2 way** associative cache with a block size of 64 bytes.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	3	133	12
462.libQuantum	0	1	0
429.mcf	104	752	24

Figure 8: Cache misses, memory reads, and memory writes for a 256kb, **4 way** associative cache with a block size of 64 bytes.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	4	150	14
462.libQuantum	1	0	0
429.mcf	103	734	22

Figure 9: Cache misses, memory reads, and memory writes for a 256kb, **8 way** associative cache with a block size of 64 bytes.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	2	133	14
462.libQuantum	0	1	0
429.mcf	103	869	3

Figure 10: Cache misses, memory reads, and memory writes for a 256kb, **16 way** associative cache with a block size of 64 bytes. We include the data for a 16-way associative cache to highlight an interesting trend discussed in this paper's "Problems Encountered" section.

4.4 Results for varying block size

See figures 11, 12, 13, and 14.

Test name	64 byte block size	32 byte block size	16 byte block size
429.mcf	1.4%	2.2%	2.96%
462.libQuantum	0%	0%	0%
403.gcc	13.6%	17.5%	16.6%

Figure 11: Cache performance varying the block size. Expressed as a percentage of memory accesses generating a cache miss. Associativity is held constant at 2-way, cache size is held constant at 256K. Notably, we vary the block size of both the L1 and L2 caches equivalently.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	2	131	12
462.libQuantum	0	2	1
429.mcf	108	765	26

Figure 12: Cache misses, memory reads, and memory writes for a 256kb, 2 way associative cache with a block size of **64 bytes**.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	3	124	11
462.libQuantum	0	1	0
429.mcf	143	790	27

Figure 13: Cache misses, memory reads, and memory writes for a 256kb, 2 way associative cache with a block size of **32 bytes**.

Test name	Cache misses	Memory reads	Memory writes
403.gcc	4	125	10
462.libQuantum	0	1	0
429.mcf	128	749	20

Figure 14: Cache misses, memory reads, and memory writes for a 256kb, 2 way associative cache with a block size of **16 bytes**.

5 Problems encountered

Members of our group spent a significant amount of time attempting to create a virtual machine image for Bochs. This endeavor entailed approximately 50 man-hours, and ultimately proved to be entirely futile upon our switch to the newly recommended and newly provided "d.img."

On all tested hardware, performance under Bochs was significantly worse than performance under competing simulation software (Qemu without the Kernel Virtual Machine (KVM) module, for example).

Further degrading the performance of Bochs, we were forced to manually flush our logfiles (e.g. by making a call to the syscall "fflush()"). This was done because, on all tested hardware, Bochs *always* exits

with a segfault. We do not know the cause of this segfault, however it prevents cleanup calls to "fclose()" from being properly executed.

We noticed a performance degradation from a 4-way to an 8-way cache and from an 8-way to a 16-way cache. We were at first confused by these results, and indeed attempting to debug the performance degradation led us to an error in our initial cache implementation. However, even after correcting errors in our cache, we continued to see a performance degradation. We talked to Dr. Abhishek Bhattacharjee, and discovered that for the $403.gcc$ and $429.mcf$ benchmarks, this behavior is actually expected. The severe increase in associativity brings about a severe decrease in the number of cache lines, and thus the very random behavior of these two benchmarks causes a performance degredation. If we were to increase the size of the cache while also increasing the associativity (e.g. holding the number of indexes constant), such a performance degredation would not occur.

6 Conclusion

For the most part, our actual results (as per 4) match our expected results (as per 3). For example, we anticipated that benchmarks with heavy spatial locality properties would benefit from an increased block size, and this was demonstrated. In our tests, *429.mcf* benchmark exhibited the worst caching performance, and indeed this correlated with the results presented in [**?**].

We have found that the 2:1 cache rule[**?**] is true in that our 4 way cache at 256k performed roughly equivalently to our 2 way cache at 512k, both with equivalent block sizes.

We assumed that an increase in cache associativity would always yield an increase in caching performance3, and we found this assumption to be invalid5.