Lucas Johnson

Jesse Leonard

**Lab 9 - Cache**

**Q1. Use Cachegrind to determine the miss-rates of an 8KB, direct-mapped cache with the following block sizes: 32 bytes, 64 bytes, 128 bytes, and 256 bytes. To do so, use commands that look like this: valgrind --tool=cachegrind --D1=8192,1,{block} ./a.out where {block} ranges from 32 to 256. When the program runs correctly there will be lots of lines specifying the cache performance. The one you are interested in is D1 miss rate -- the miss rate for the L1 data cache. This value is rounded to 0.1%. If you want to see the result with more precision, you can divide the number of L1 data misses by the total number of L1 data accesses. After you have run Cachegrind for each block size, save the miss rate from each run. List the miss rate for each block size tested.**

A1: 32bits: ==3580035== D1 miss rate: 3.1% ( 3.1% + 13.8% )

64bits: ==3580524== D1 miss rate: 1.6% ( 1.6% + 9.5% )

128bits: ==3580666== D1 miss rate: 0.8% ( 0.8% + 8.6% )

256bits: ==3580842== D1 miss rate: 0.4% ( 0.4% + 8.4% )

**Q2. Based on your observations, determine a formula for the miss rate in terms of block size. The formula will not be exact, but it should track the miss rate quite closely.**

A2: Let’s call block size ‘b’

We’ll call the miss rate ‘m’

m ̴ 1.024/b

**Q3. Your formula above should have an intuitive explanation (i.e., it should not just be a line fit to data). Explain what is happening in the cache during each memory access to produce the results you observed.**

A3: The reason that the miss rate decreases as the size of the cache increases is because the cache can grab longer blocks of memory from the RAM, which are more likely to have the values that we are looking for. This is due to principles of spatial locality, predicting that we are likely to use values in memory that are close to one another. Because we tested for this pattern using block sizes that are doubled from the previous run, we observe that the miss rate roughly halves due to the two-times increase in block size.

**Q4. Now, write a C program for which the miss rate is considerably higher for a 64-byte block than for a 32-byte block.**

Graphical user interface, text, application, email

Description automatically generated

**Q5. Plot the miss rate as the cache size increases from 1K to 512KB. (Use powers of two for the cache size. Choose any block size you like.) Running cachegrind on this input should take 15 to 20 seconds on EOS/Arch and should generate about 5.6 million memory accesses. If you are noticing a faster run time or fewer memory accesses, then you are doing something wrong. The most common problem is a mis-configured input file. Similarly, if the program doesn’t terminate after 45 seconds or so, then you probably forgot the input file. Attach your plot to your lab report.**

**Q6. When (for what cache size) does the miss rate reach zero?**

**Q7. Why do you think this cache size produces a 100% hit rate? (Hint: Why is the input file named input\_5e4?)**

**Q8. Determine the optimal block size for insertion sort. Run runInsertionSort using input\_5e4 as input given cache sizes of 2KB, 8KB, and 32KB and block sizes from 32 to 512 (powers of two only). Present your results using a graph with block size on the x-axis and the miss rate on the y-axis. Please generate one graph with three lines: One each for each cache size. Your graph should have a form similar to**[**Figure 8.18**](https://cis.gvsu.edu/~kurmasz/Teaching/Courses/W22/CIS351/Labs/Cache/effectsOfBlockSize.jpg)**in Harris and Harris (2nd edition). There are a couple ways to generate these graphs. One option is to place the data points into a spreadsheet that can generate graphs. Another is to use a tool called *[gnuplot](https://www.gnuplot.info/)*. To use gnuplot, place your data in a plain text file similar to [sample\_gnuplot\_data](https://cis.gvsu.edu/~kurmasz/Teaching/Courses/W22/CIS351/Labs/Cache/sample_gnuplot_data). This sample file also contains the gnuplot commands that will generate the graph.**

**Q9. Generate a similar plot for runMyQsort. Use the same cache and block sizes. (To compile the quick sort runner, run make runMyQsort.)**

**Q10. When comparing the plots, you will notice that the optimal block size for quick sort is smaller than for insertion sort. Why is that? (Hint: Think about how each algorithm accesses the array as it runs.)**