**Cache Analysis**

**Program modifications**

Several modifications to the provided program template were made. First, implementation of the different replacement strategies was implemented, by wrapping the update of the cache counter in an if clause. Second, Execution of the program was wrapped in two separate loops, one for the file (gcc.trace vs swim.trace) and one for the replacement strategy, resulting in four datapoints being collected on every execution.

Introduction

The purpose of this analysis is, broadly, to determine what effects, if any, various configuration settings have on overall cache performance, as measured by hit rate. The first analysis section focuses on the effect of cache size, varying that value, and determine what effects are produced on the various associativity settings. The second analysis will compare LRU, FIFO, and direct mapping across different cache sizes to determine if one option outperforms any of the others. The final analysis attempts to determine what the optimal block size is, by fixing the cache size and testing out different block size configurations. The primary metric that I will use to compare the different results will be hit rate.

**Initial Analysis**

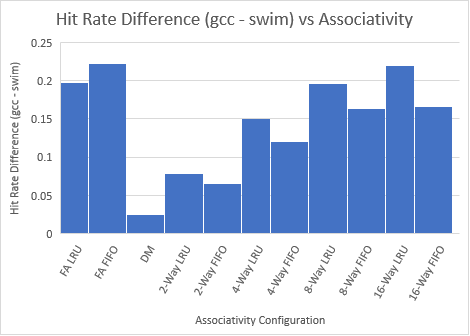
**Test setup:** Block size exponent was fixed at 6, cache size exponent was varied from 10-15, inclusive. Test runs were executed on both gcc and swim datasets, and on both LRU and FIFO replacement strategies.

The first set of analyses compared the average hit rate on both the gcc.trace and swim.trace files while varying the cache size exponent, keeping the block size exponent constant. That resulted in this chart:

I took two things away from this analysis, the first was just how close in performance the various different cache settings were, especially at larger cache sizes. In an effort to make the differences more pronounced, I replaced the Y-axis with the negative log of the miss rate, resulting in this graph:

The second takeaway was how surprisingly well direct mapping performed. This was admittedly somewhat unexpected, my initial hypothesis was that higher levels of associativity would result in higher hit rates, as they have more options on where to store any given block of data, but these plots showed the opposite. After digging into the data, it was apparent that performance was very dependent on which dataset (gcc.trace or swim.trace) was used to run the simulation. Separating this data resulted in the following two graphs:

On the gcc.trace dataset, direct mapping consistently performed near the bottom of the, but in the swim.trace dataset, direct mapping was consistently the top performer. Additionally, high-associativity configurations significantly underperformed low-associativity configurations while utilizing smaller cache sizes on the swim.trace dataset, and the fully associative FIFO configuration in particular significantly underperformed other configurations even with higher cache sizes. To highlight the differences between the results of the gcc.trace and swim.trace analyses, I plotted the differences between the performances of the various configurations, resulting in this graph:



Direct mapping had by far the smallest performance drop off when moving from the gcc.trace to swim.trace datasets. And an increasing drop off was positively correlated with an increased level of associativity.

Overall, I identified several major takeaways from this first analysis. The first was that the overall average difference in performance was most notable at lower cache sizes and that this difference decreased significantly as the cache size increased. Next, and possibly the most important takeaway, was how dependent the performance was on the testing data. This highlighted the importance of ensuring that testing data is a representative sample of the expected workload. The final takeaway was the most surprising for me, that a more complex configuration and higher associativity is not automatically better. Direct mapping was the least effected by the different datasets, and at least with this test data, was the most flexible and highest performing overall. But again, it is likely that this result is dependent on the testing data, and we must keep this in mind when designing caches in the real world.

**Replacement Strategy**

**Test setup:** This analysis used the same data collected from the first analysis.

Moving on to replacement strategy, I first looked at the performance of the two strategies averaged across the various configurations as well as direct mapping, resulting in this graph:

Once again, direct mapping came out on top. Taking what was learned in the previous analysis, I separated the results out by dataset, resulting in the following two graphs:

And once again there were significant differences between the datasets, with direct mapping being the worst option on the gcc.trace data, and wildly outperforming on the swim.trace data. However, two things stood out in this analysis. First, was that replacement strategy became less important as cache size increased, with the hit rates converging on both datasets, and the differences being almost negligible on the gcc.trace data. The second takeaway was the LRU consistently outperformed FIFO. While the overall difference was small, the trend was clear. This is likely due to LRU being more in line with the principle of temporal locality.

**Block Size**

**Test setup:** Cache size exponent was fixed at 12, and the block size exponent was varied from 2 – 10, in increments of 2, inclusive. A second data collection was run afterwords, with the cache size exponent fixed at 16, and the block size exponent was varied from 2-14, in increments of 2, inclusive. In both runs, just like in the previous analysis, data was collected for each combination of dataset and replacement strategy.

The final set of analyses was done to determine if there was an optimal block size. Starting with a fixed cache size exponent of 12, and keeping in mind the importance that the dataset played in the previous rounds of analysis, I plotted the average performance of various block size exponents on the different datasets, resulting in this graph:

Performance on the datasets increased until it reached a maximum and then decreased, implying that there is a goldilocks zone for the block size, and suggesting that it may be somewhat independent of the dataset being used. Interestingly, the peak performance occurred when the block size exponent was exactly half of the cache size exponent. To see if this trend continued, I ran the same experiment with a cache size exponent of 16, resulting in our final graph:

The trend is gentler this time, but still present. Performance on the gcc.trace data did still peak at the anticipated value of 8, but on the swim.trace data the peak occurred at a cache size exponent of 6. Meaning that while this setting is not as data independent as I had hoped, a block size exponent around half of the cache size exponent does seem to be a good place to start.

**Major Takeaways Summary**

-Specific configuration settings had the most variance in low cache size situations

-Higher cache sizes resulted in increased performance, and the performance differences between configurations shrinking significantly

-Test results were highly dependent on the dataset used

-More complex associativity configurations were not automatically better

-Replacement strategy performance converged even more than other configuration settings

-LRU outperformed FIFO in most configurations, but the differences were small

-A good starting block size exponent was around half of the cache size exponent

**Conclusion**

Overall, several major trends appeared throughout the testing. The first is probably the least surprising, that hit rates improve as cache size increases. This makes intuitive sense, as all things being equal, a larger cache has more places to store any given item, making it more likely that the cache will contain it when requested. The second was the replacement strategy did not have the influence that I had expected. The difference in performance between LRU and FIFO was largely negligible, and this difference decreased as the cache size increased. However, the results with the available test data showed that LRU consistently outperformed FIFO, though not by much. We also determined that the optimal setting for the block size exponent is likely around half of the cache size exponent. Finally, we found that the optimal associativity configuration is highly dependent on the test data, with lower associativity outperforming on one, and the reverse being true on the other dataset. This setting will likely need to be tuned to the real-world data, highlighting the importance of using representative data in these tests.