COMP 7300: Lab Assignment-1
Exploring Caching Performance

Samir Hasan (szh0064@auburn.edu),
Steffi Mariya Gnanaprakasa Paul Arasu
(smg0033@tigermail.auburn.edu)

Group ID: 25

September 10, 2013

Introduction

In this project, we have tried to look into the impact of cache and block size along with two block replacement policies, FIFO and LRU, on the miss rate of the cache. We also look at how temporal and spatial locality may affect the performance of a cache drastically. For the purpose of this study, we have built a cache simulator in java. The simulator program creates a cache, generates 100,000 random memory references and logs the final results (hits, misses, etc.) that we eventually plot as illustrations. The last section of this report contains steps to run our simulator and reproduce these results.

Our simulation has been setup as follows:

- **Cache size:** 4K, 16K, 64K and 256K
- **Block size:** 16B, 32B, 128B and 256B
- **Address:** 32-bits

Part A

In this section, we present our simulation results on the test data. Since our performance metric in this study is **miss rate**, we have implemented a simple cache model, where we keep a count of memory references undergoing a hit or a miss in the cache. During a hit, we update the **lastVisitedTime** for a block in the cache, whereas for a miss event, we assign a new block and initialize its creating and visited times accordingly.

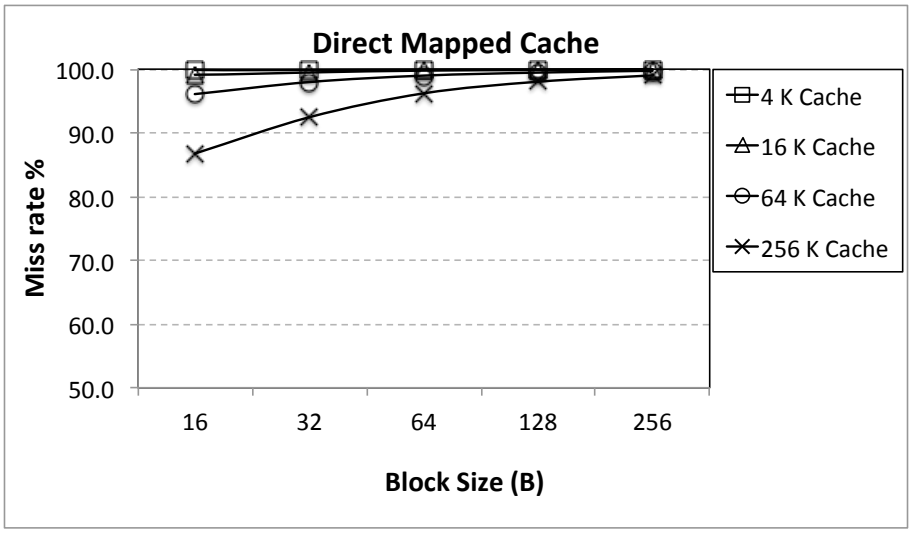
We have simulated a direct mapped cache as well as a fully associative cache that uses FIFO block replacement strategy. The last part of the section makes a comparative analysis of the two caches.

Direct Mapped Cache

In the direct mapped cache, each memory block is assigned a single block in the cache. The lower order bits of the block address identifies the index into the cache where the memory block may be

placed. Since it is a one-to-one mapping, we do not require any block replacement strategy, so that when conflicts occur, we will always have a miss.

We got the following results from our simulation.



From the figure above, we notice a few trends. Firstly, the miss rate increases with increasing block sizes for all the cache sizes. Secondly, bigger caches incur lesser miss rates, at least initially when the block sizes are small. With increasing block sizes, all the caches eventually reach very close to a 100% miss rate.

The performance we have achieved is not realistic. First of all, we expect overall miss rates to be much lower than that portrayed in this graph. Also, with a higher block size, our results deviate from the usual trend, which is miss rate decreasing with increasing block size (up to a certain extent). But we have achieved a completely opposite trend. The results however confirm the fact that bigger caches have lower miss rates, which is in agreement with our findings.

The reason behind these discrepancies are easy to see. In order to simulate memory accesses, we have generated 100,000 32-bit **random** memory (word) address or $(100,000 * 4)/16 \approx 25,000$ block addresses. Since we have used a uniform distribution function, we expect the numbers to be mostly unique. If we now look at our chosen cache sizes, for example, a 4 K cache with block size of 16 bytes, we can see that the cache can only store about $(4 * 1024)/16 \approx 256$ blocks. Thus, the cache has a very small number of slots compared to the total number of unique memory accesses $(256/25000 \approx 1\%)$. In this case, we have about 99% miss rate due to the small size. This explains why we have such high miss rates for our caches.

Apart from that, we also have a increasing trend for miss rates with larger block sizes. Since our cache sizes remain the same, if we increase the block size by some factor, the number of blocks that the cache can have is reduced by the same factor. Increasing the block size shows improvements in cases where spatial locality is exploited. Since our address generation is purely random, the addresses do not have spatial locality. Thus, when we decrease the number of blocks of the cache by making each block bigger, our cache suffers from misses even more since the memory addresses are uniformly random.

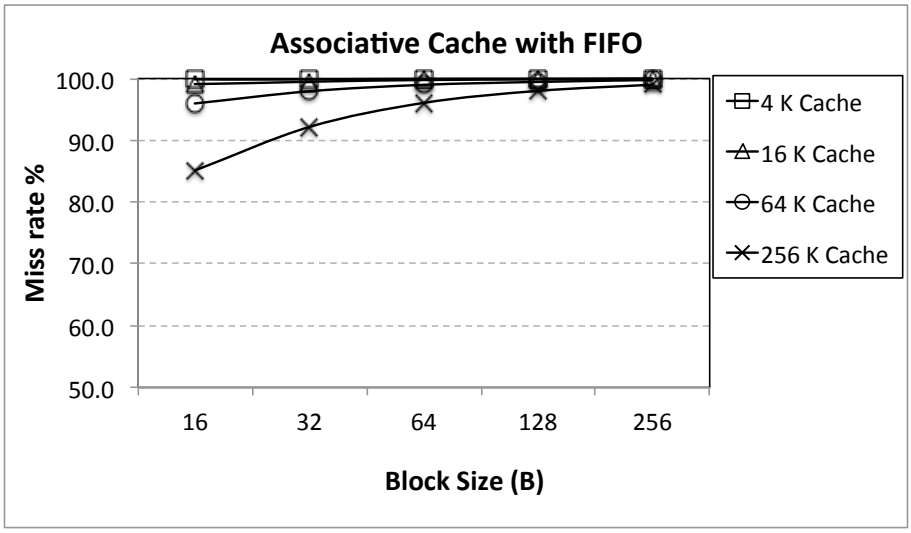
The reasons explained above also holds for the trends that we will see in the other caches as well.

Fully Associative Cache with FIFO

The fully associative cache can maps a memory block to any block in the cache. When there is a miss, we first look for any invalid or empty block in the cache to keep it. If we do not get one,

we use a block replacement policy to remove one from the current cache. In this section, we have used the First In First Out (FIFO) strategy. This means that when we have to throw a block out, we choose the one that was the earliest to have entered in the cache.

Our simulation results are shown below.



From the figure above, we see that we have achieved a trend very similar to that for direct mapped cache. We expected lower miss rates for the fully associative cache than direct mapped, since it makes maximum use of the space available. The reason behind such results are very similar to those described for the direct mapped cache. We have simulated too many unique memory references such that our small caches were exhausted, so we gained no significant utility by using the cache. Moreover, since the random references neither use temporal nor spatial locality, it is completely unpredictable as to what to put into the cache. Thus, both of our caches exhibit the same poor miss rates.

Conclusion: Comparing the Caches

In comparing the caches, we can use two metrics such as the miss rate and delay.

Miss Rate

Fully associative cache has a lower miss rate than direct mapped cache. This results from the fact that a memory block can be placed to any empty or invalid location in the cache, whereas in direct mapped cache, we can only choose one single spot for the memory block. So, even if there were plenty of space in the cache, a direct mapped cache will always try to put the block to its corresponding position in the cache, but nowhere else. This results in higher misses incurred by the direct mapped cache.

Access Time

The direct mapped cache has a lower access time than fully associative cache. Since we can instantly figure out where a block might be in the cache directly from its address, it is very fast. On the other hand, the associative cache has no idea whether the block is present in the cache and if so, where. As such, it queries into all the blocks in the cache to find a block matching the address referenced. This incurs a significant delay in the access times for looking up an entry in the associative cache.

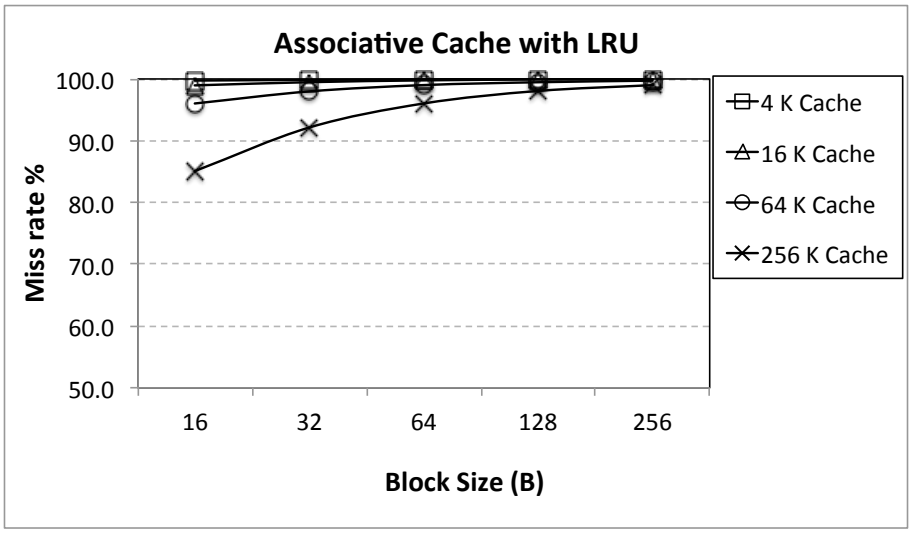
Part B

In this section, we implemented the fully associative cache with a different block replacement strategy, the Least Recently Used (LRU) strategy. According to this policy, we choose to throw away the block that has not been referenced for the greatest length of time.

Fully Associative Cache with LRU

Our goal in this experiment is to compare the miss rates for a fully associative cache when using LRU as the replacement strategy.

We got the following graph from our results.



Interestingly, we get almost the same graph as we did with direct mapped and fully associative FIFO caches. The results are misleading, mainly for the fact that we are not using temporal and spatial locality while generating the memory references. The reasons for the trend in this graph is exactly the same as that for the previous two graphs. Also, no difference in performance between FIFO and LRU is evident in these graphs.

Conclusion: Comparing the Strategies

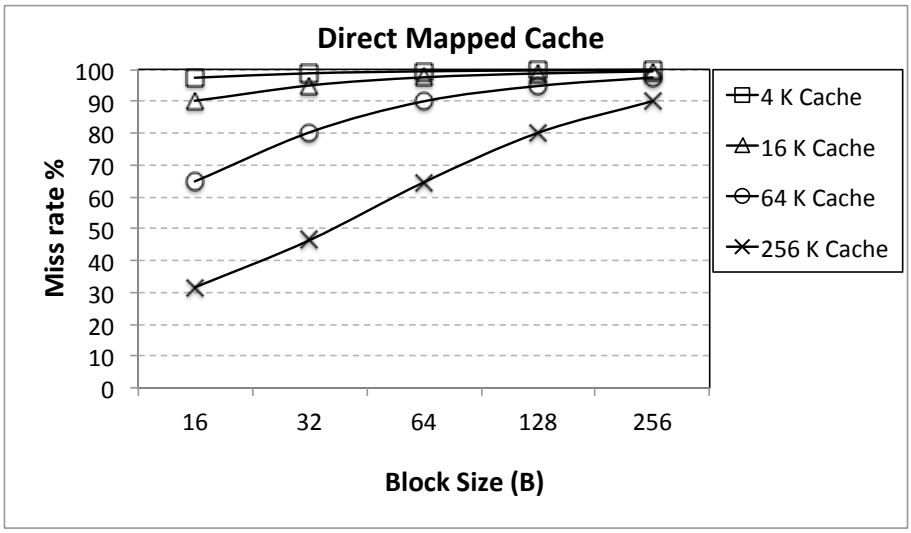
Although the real-world performance is very different for LRU and FIFO, we have not been able to see any in our experiment. We got almost the same graphs for both the strategies, making it impossible to perform a comparative analysis on their performance. The reason behind our unexpected results is, again, the memory access pattern that we generated. We have generated too much unique addresses that is way beyond the capacity of the cache. Moreover, the LRU strategy is based upon temporal locality, but we did not implement that in our system. As such, we did not notice any improvement in the miss rate for LRU. In a real scenario, LRU outperforms FIFO due to temporal locality.

Validation

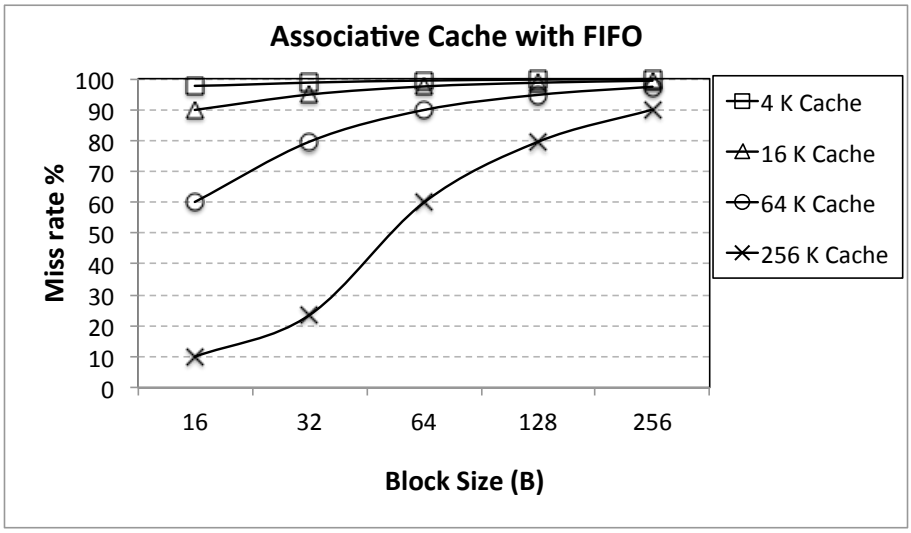
In the previous sections, we saw misleading results due to an unrealistic means of accessing memory. As a results, we could not achieve any confirmatory results in order to understand trends for miss rates. In this section, we use a slightly realistic input set, validate our reasons that we showed for the previous graphs, and produce results that can help realize trends for miss rates better.

In our new experiment, we have ensured that the 100,000 memory accesses occur randomly, but this time the addresses are picked from a smaller predefined pool of 10,000 addresses. This helps introduce a temporal locality into our system. With this setup, we get the following trends for the different cache models.

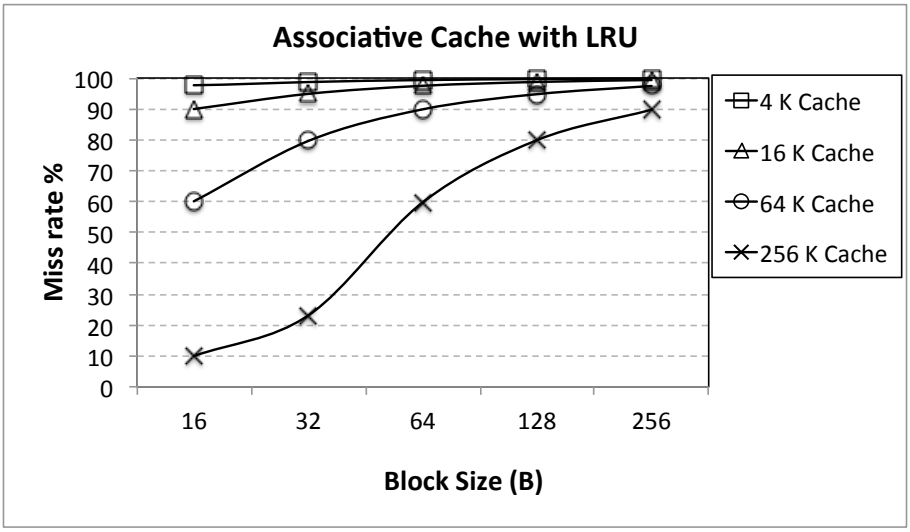
Direct Mapped Cache



Fully Associative Cache With FIFO



Fully Associative Cache With LRU



From the figures above, it is evident that large caches with smaller block sizes perform better in our system where there is no spatial locality. We have better results in that the initial miss rates are now much lower than what we had in the previous section. Without spacial locality, we expect to have an increasing trend of miss rates with block size. These graphs hint on the fact that beyond a specific block size to cache size ratio, the miss rate degrades drastically (e.g., from block size 32 to 64 for Fully Associative Cache). This validates our reasoning for the previous graphs.

Conclusion

In our experiment, we have tried to see how caches perform in a random sequence of memory accesses. Our results helped in understanding the principles (temporal and spatial) that caches are built around. We have had unexpected results due to using an unrealistic memory access pattern, and ran our experiments on a slightly better input set to validate our reasons behind the findings. We expect that if we can simulate a more real-world scenario, we will have the desired results.

Running the Simulator

The simulator is a Java application. It runs on JDK 1.7 or 1.6. It is in the form of an eclipse project, but can also be run from the command line. The main class is in **Program.java** All the files are under the **CacheSimulator/src** directory.

At end of the simulation run, a detailed statistics is displayed on the screen. Near the end of the output, there is a comma spearated list of numbers that can be pasted into Excel and generate graphs easily. The columns for the list are cache size, block size and miss rate %.

Running the code

1. In the terminal (command prompt) window, type: **cd CacheSimulator/src**
2. Compile the java files: **javac *.java**

3. Run the Program class: **java Program**

Inside the code

Program The main entry point for the simulator. Here, the initial cache and block sizes, as well as displaying the statistics are handled.

Cache An abstract class that defines the interface and basic functions for the caches in the system. Defines the abstract function **fetch**.

DirectMappedCache Extends **Cache**, and implements the abstract method **fetch**. Calling **fetch** is equivalent to the CPU querying the cache for an address.

FullyAssociativeCache Extends **Cache**, and implements the abstract method **fetch**. It also takes a **ReplacementStrategy** in its constructor

ReplacementStrategy Interface for LRU and FIFO implementation, defining only one method interface, **getIndexForAddress**

FIFOReplacementStrategy Implements the **ReplacementStrategy**. Chooses the next block to remove in FIFO strategy

LRUReplacementStrategy Implements the **ReplacementStrategy** by employing LRU

CacheBlock Model class representing a single block in the cache