Cache Simulator

Vince Coghlan and Zach Vogel

May 5, 2014

1 Introduction

Cache configurations are not a one size fits all job. Some cache's are better than others for specific applications. For example, a multiway cache will benifit a system that has limited space, as hit rates will increase. A direct-mapped cache will be better for a larger cache, since our hit times will be very fast. Our simulations are an attempt to discover some of these facts. We created a simulator that will preform instruction, read, and write operations on any associativity or size cache. This cache system included a hierarchy of a level 1 instruction cache, a level 1 data cache, and a unified level two cache. The caches are all write-allocate, write-back, meaning writes will immidiately write to the cache, and any data in the cache that has been written must be accounted for and lazily written back down to level two and eventually main memory. All caches maintain their own Least Recently Used (LRU) policy in order to preform the best that they can given information at any point in execution. We measured many metrics, such as hit time, miss time, cost, CPI, ideal times, and kickouts. Although many things are idealized, this simulator is very similar to what you would see in a modern computing machine.

2 Design

There were several design considerations of note that must be addressed. The first concern of ours, was one of memory. The stack was stored as an array of structures in heap memory. This was mainly for speed reasons. An array is generally faster than a linked list when we dont know where in the list we are going. The second consideration was how to design the requests. We found it to be simplist to split the simulator, much like a computer does, into the CPU portion, and the cache portion. The CPU decides how many references it needs, and decides what to do when it gets a hit, miss, or dirty kickout. The cache portion, on the other hand, was a simple machine. It took the address and the cache it was looking in and that is essentially it. It returned a hit, miss, or the tag of a dirty kickout that

it received. The CPU simulater then decides what to do with this information. The last important consideration was the LRU system. We knew that the fastest solution would be to provide each block with a number that would represent it's place in the LRU. This would be a constant time solution. This, however, led to some difficulties in implementation when we had variable sized caches. We instead went with a linked list structure. This made it easy to program, and we knew that for every cache other than the fully associative, this time would be relatively minimal.

The use of the code is quite simple. The inluded makefile will build the program on our versions of Mac OSX 10.9 and Arch Linux. just typing make is sufficient, while make clean will remove the compiled files. The libraries used are quite simple. libc and the POSIX standard library are all that is required. Once built the program will be called cachesim. At that point it can be run with an option describing some features. cachesim -v will run in "verbose" mode, where each cache reference and clock cycle will be spelled out step by step. Besides helping in debugging, it is quite relaxing to watch all the text fly by. cachesim -h will print out the usage: "usage: cat <traces> | ./cachesim [-hv] <config_file>". Note that the traces must be sent in through stdin, any other form of stdin will work as well, not just cat. The config file takes the form of a series of lines, each with a parameter, followed by a space, then a value. For example, a line could be "l1dassoc 4". That would set the level 1 data cache to a 4-way set associative cache. A full list can be seen in the code, or in the example config files provided.

3 Simulation

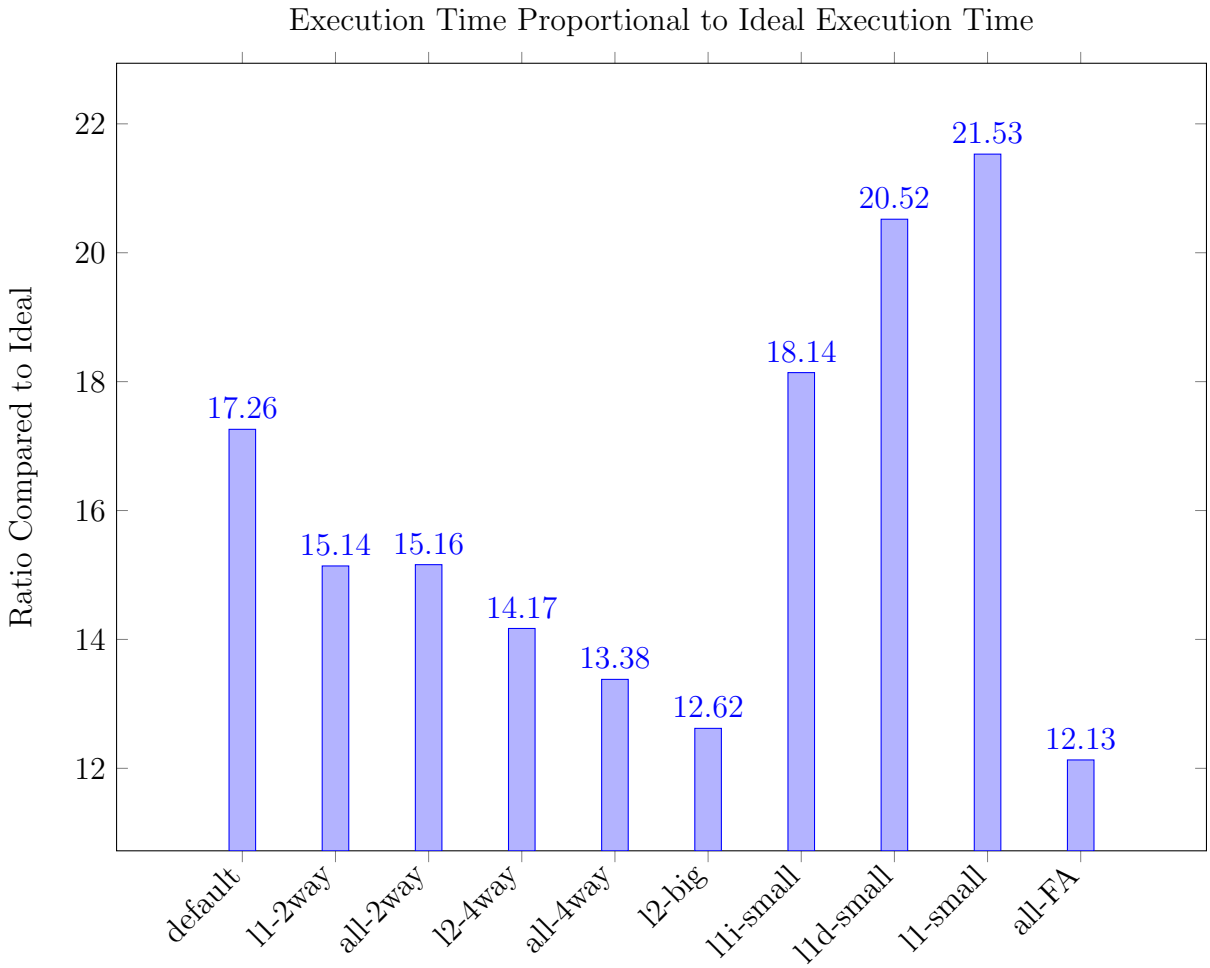
The simulations were run on multiple cores of a PC running Debian Linux virtualized in a windows enviornment. In total they were run in a little under 20 hours. The simulations included a number of traces that tested a variaty of typical operations a computer would run. Not suprisingly the simulations of the fully associative caches took almost twice as long as the direct mapped ones. The LRU's linked list had to be traversed and reorganized on almost every reference. This meant slow references, as well as a much less efficient real-life cache. We simluated each of the 5 traces on 10 different caches. Additionally, the omnetpp simulation was run on systems with multiple memory chunk sizes from cache to main memory. We will preform cost benifit analysis on this later.

4 Results

4.1 Cache Configuration

The best way to present this data, in our minds, is to represent it as a ratio of the real time over the ideal time. Since many of the traces vary in ideal execution time, but are

mostly proportional to the stack parameters, the best option to initally analyse the data is to average out the values from each group of traces and present the ratio over ideal time. This can be seen below.

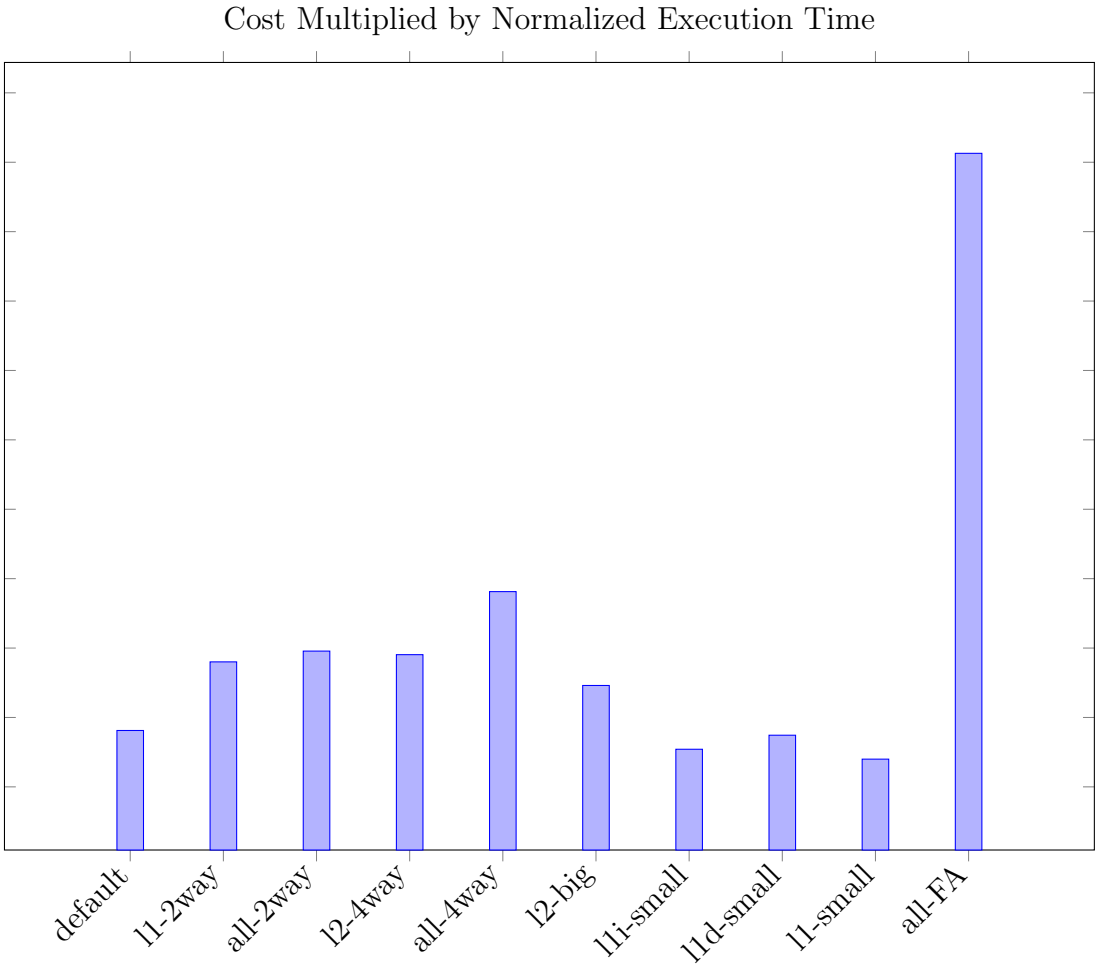


The two fastest caches are all-FA and l2-big and, unsuprisingly, the slowest caches are the ones with smaller l1 cache sizes. We know that a fully associative cache will gaurantee the highest hit rate. If we had a very high miss penalty this would be our best bet, since we would always hit. The larger L2 cache will also guarantee a low l2 miss rate which means, less accesses to main memory, the slowest part of the process.

There are reasons why we dont use these caches for our everyday computing. The fully associative simulation does not take into account the amount of time it takes to move down the LRU list and update it on every reference. It is also extremely expensive, as we will show later on. The large L2 cache is sufficient to allow more hits of older data. This is great, but comes at the cost of a larger cache.

4.2 Cost

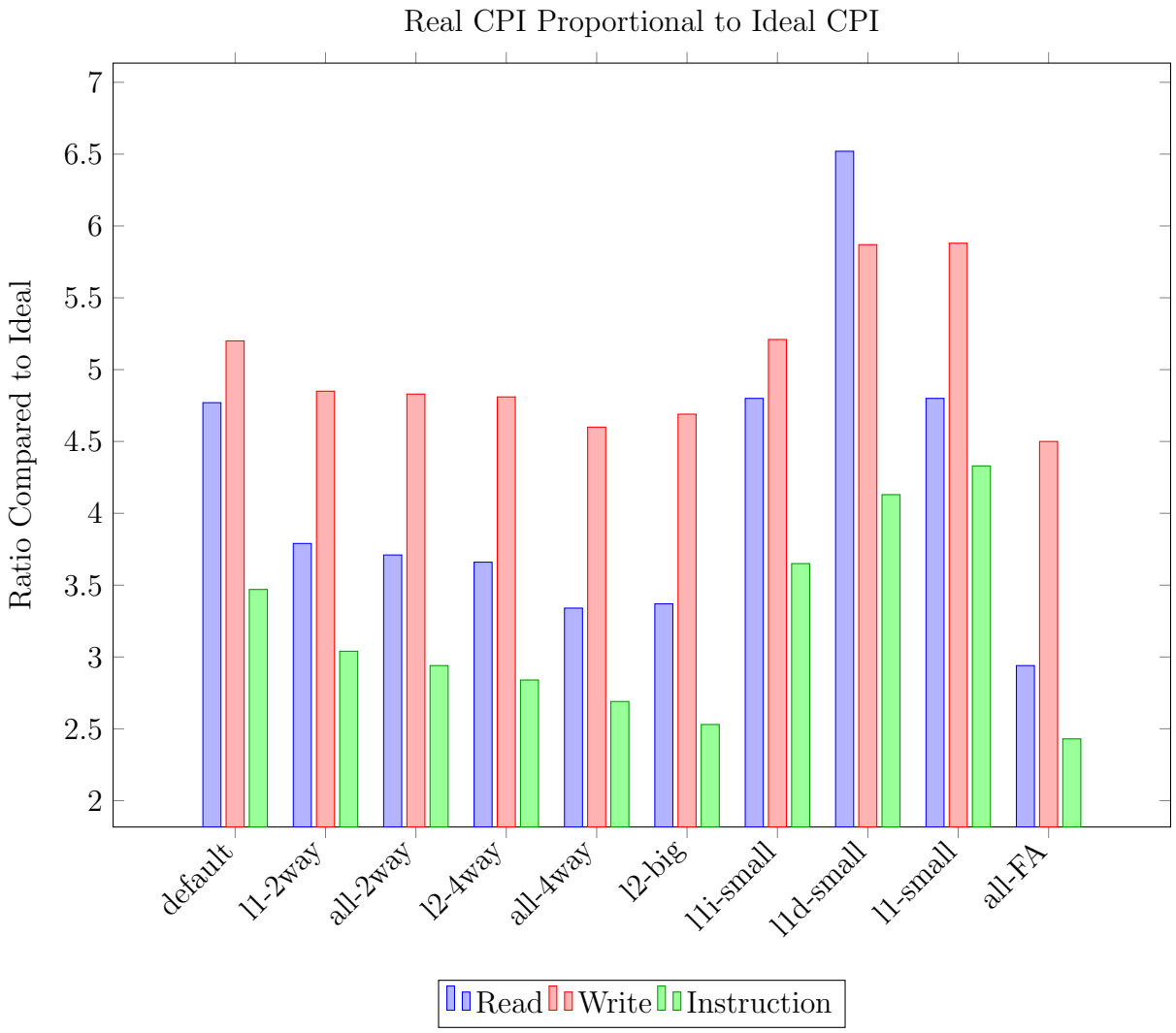
Cost considerations are very important when considering which cache to choose. If we manufacture processors, what matters to us is the bottom line, profit. The cost multiplied by the preformance as measured above can give us a good idea of the "best" cache system.



In this graph, it is clear how poor the fully associative cache actually is. We would much prefer to use any of the others. The low cost of the small caches makes them ideal for the situations we put them through. Part of the reason for this is the unrealistic cache times that we gave the system. We assumed that a L1 cache will have the same hit and miss time. This means that a small, low associative L1 cache is best, and a large L2 cache is best. This is also why our default cache preformed so well. Miss rates were not as important in the cache as things like the avoidance of going to main memory.

4.3 CPI

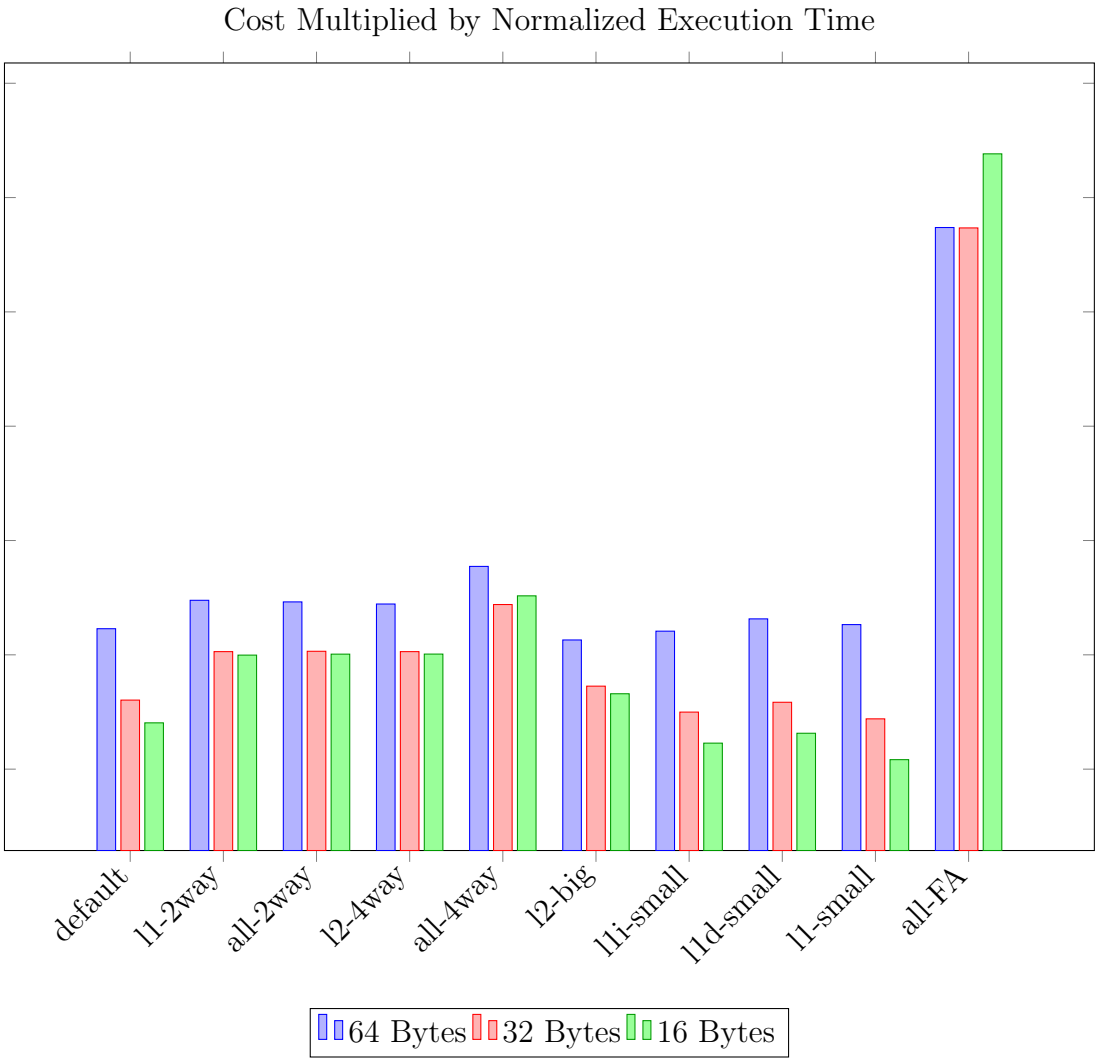
CPI is important to know if you are going to be preforming a specific operation with the machine. A mathematical program may use many instruction references but low reads and writes. A text editor may do nothing but writes. This means we must know the cycles per instruction. Once again this value was normalized with the ideal CPI then averaged over all traces.



As expected, running nothing but reads and writes would be a bad idea on a small data cache system. The most suprising result, however, is that it is worse for almost everyone involved. Even instructions get worse CPI on a l1d-small cache than they do on a l1i-cache. This is because of all the dirty blocks in L2 cache. When we eventually read our instruction from L2, there is a high probability of a dirty kickout. The fully associative cache is the clear winner, but once again, the data must be ignored, since there is no cost effective way of implementing this cache.

4.4 Memory Chunksize

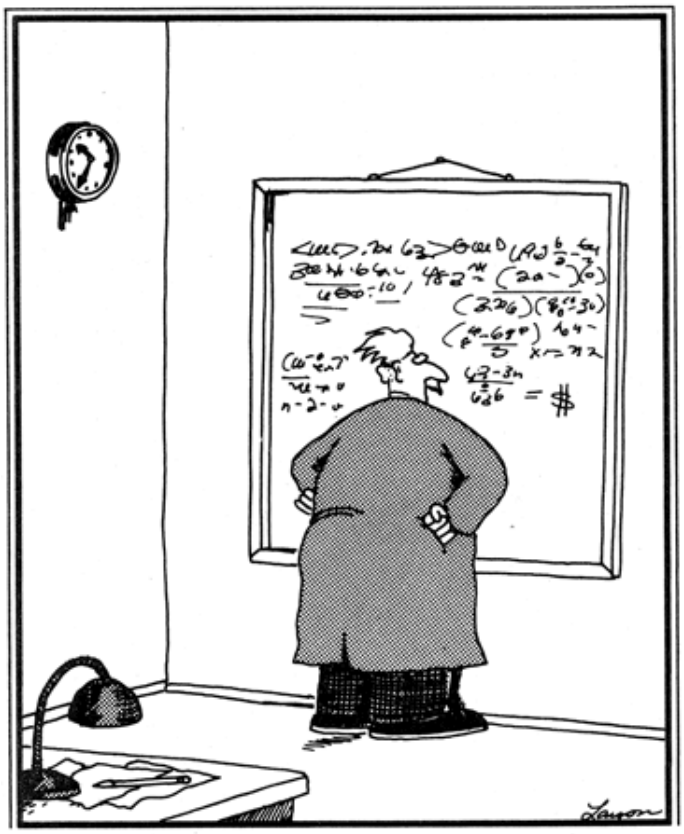
The last component we explored was the memory chunksize between L2 cache and memory. One would imagine performance to increase as we increase the size of the chunksize. More importantly cost increases as we increase the chunksize. We can once again graph the cost multiplied by the normalized "cost" that we used previously.



As in so many of the other categories, youre design choice is going to be based off of what you want. Everything is a tradeoff. 16 Bytes is generally better for most of the caches, but for most of them, 32 Bytes is almost as good. In the actual design of a architecture, it may be easier to design a 32 byte bus.

5 Conclusion

Our data shows that the best memory model, using these idealized values given to us, would be two small, direct mapped caches and a unified direct mapped L2 cache with a 16 byte chunksize. This is a very good allegory to everything we haved learned this year, the whole architecture design industry revolves heavily around cost. A cache that is too expensive may not sell, and your business is gone. It is not just about finding a balance between cost and performance, but about pushing the limits of how cheap you can make something. It is the sad truth that I wish I didnt know about the business of modern computing. A business is exactly what it is.



Einstein discovers that time is actually money.