CS 352 Project: Cache Simulator

Deborah Hawkins (rowehldr)
Rishi Dewan (rrd328)

In this project, we simulated an L1 cache and ran a set of experiments to determine the optimal configuration of capacity, associativity, and blocksize to reduce the miss rate and the number of evictions.

Experiences During Development

Program Structure

We structured our code so that both main memory and cache memory would have the same public interface--methods for reading a word of data, for writing a word of data, and for printing the updated contents. However, main memory can exist independently, whereas a cache requires access to a main memory and consists of a collection of sets. Each set in turn consists of a collection of blocks. Although blocks could be further separated into a collection of words, there would be no advantage to creating a separate class for them rather than just placing them in an array.

In this structure, the cache is responsible for determining if a read or write is a miss, and handling allocations and writebacks if necessary. For reads and writes that it knows are hits, control is passed to the set. Although fully-associative and direct-mapped caches could both be implemented as special cases, they can also be fully simulated by this generic cache structure.

The set is responsible for maintaining the blocks it contains. This includes seaching for a block that matches a certain tag, keeping track of the least recently used block (described under Challenges in Development), and passing reads and writes through to the appropriate block.

The block is responsible only for it's own contents, which include an array of contents that can be read or written, as well as the tag, the valid bit, the dirty bit, and an int that defines when the block was last accessed. Words could have had their own class within this structure, but it would provide no additional functionality.

Finally, conversion between hexadecimal and decimal is handled by a separate library class called IO.

Address Partitioning

When given a particular configuration, we partitioned the 32-bit address into tag, index, and offset values. In order to determine these, we first converted the capacity and blocksize to word amounts rather than bytes, and then specified the number of sets by `capacity / (associativity * blocksize)`.
From these values, we calculated:

- `tag = address / (sets.length * blocksize)`
- `index = ( address / blocksize ) % sets.length`
- `offset = address % blocksize`

Challenges in Development

During development of our cache simulator, we came across several challenges. The use of word addresses rather than byte addresses was the first source of confusion. However, after recognizing this and storing the capacity and blocksize variables in terms of their size in words rather than bytes, this served to simplify the code.

In the cache design in figure 5.17 of the Patterson text, there are two outputs for the cache--one which specifies if the access was a hit and another which returns the data. Since in Java we're limited to one return value, we decided to make this into a two-step process, involving first determining whether the word exists in the cache and then performing the appropriate access. This also allows for an easy implementation of the counter to track the number of read misses and write misses, as well as leaving control to the cache for allocating the needed block and writing back when necessary.

In addition, an associative cache uses a parallel search for a block within a specific set. Parallelization techniques weren't available to us within this simulation so searching for a block is linear.

Since caches generally employ only an approximation of LRU, we also had to decide how to implement a full version of LRU. The two main options were to use a linked list and move accessed items to the end (to get the least recently used at the beginning) or to use a counter that assigned an integer to each block when it was used. We chose the second option, as it seemed closer to the approximation a real cache would use. With this option, we're restricted to less than 2^{31} accesses to prevent overflow, but none of the simulations performed approach that amount.

Testing Plan

Rather than running all possible combinations of parameters, we decided to test only combinations with a minimum capacity of 16 KB and a minimum blocksize of 64 bytes, since we know that larger capacities reduce conflict misses and that larger blocksizes reduce cold misses.

In addition to looking just at the miss rate or just the number of evictions, we also created a custom statistic which takes both into consideration. The memory access rate (MAR) is the total number of memory accesses, due either to misses or to evictions, divided by the number of cache accesses.

Top Configurations

Ranked below are the five best cache configurations ranked by MAR using the stress test provided:

Configuration	**Capacity (in KB)**	**Associativity**	**Block Size (in bytes)**	**Number of Sets**	**MAR (%)**	**Total Misses (%)**	**Evictions**
1	64	16	512	8	5.322	2.734	2588
2	64	4	512	32	5.328	2.738	2590
3	64	8	512	16	5.340	2.744	2596
4	64	1	512	128	5.342	2.746	2596
5	64	2	512	64	5.348	2.748	2600

From these results, it appears that the maximum capacity and maximum blocksize are both necessary in the optimal configuration. However, the outlier configuration with a capacity of 32 KB, a blocksize of 512 bytes, and an associativity of 16 ranked third place in terms of miss rate. It's only when taking into consideration the 2660 evictions it created that its weakness is found.

Another interesting note is that with the capacity and blocksize constant, the best associativities have a generally decreasing trend but also exhibit some randomness: 16, 4, 8, 1, 2. This may be related to the dataset used in testing.

Missing Factors

There were several factors not considered in this simulation but which would also be important in selecting a configuration for a real cache. These include how the configuration would affect the cache access time (hit time), complexity, and die area used.

First off, increasing the capacity directly affects the die area used, which leads to a higher cost.

However, in an L1 cache in particular, the focus is usually on the cache access time. If speculation is used, a direct-mapped cache (associativity 1) can create even more speed-up. Since there's only one option for each hash, the value can start being used even before checking whether the value is valid and then squashed later if it isn't.

Reducing the complexity of the system (ratio of the blocksize to the capacity), either by reducing associativity or by reducing the number of sets, leads to a lower cost in the amount of wires. On the other hand, increasing the associativity of the cache increases the amount of comparators needed. This increase in the complexity not only requires more die area for the extra transistors but may also increase time because the length of the wires would also be increased. In addition, more comparators means more power consumption by the cache.

Finally, in the case where a quick time to release for our new cache is essential, it's necessary to have as little complexity as possible. Reducing associativity or the number of sets generally also reduces testing time so the product will get to the market more quickly.

Conclusion

This project has given us more insight into caches in respect to the Patterson text and the lectures. Figuring out a way to implement such properties as associativity, write back, and allocation has allowed us to do extra research and experimentation that has helped to tie together everything we've learned. It was interesting to find that associativity wasn't as important as blocksize or capacity in optimizing our cache, although without other considerations the maximum of everything capacity, blocksize, and associativity gives the best results.