Caching Algorithm Performance Analysis

Bonnie Ishiguro, Apurva Raman, and Shruti Iyer
Computer Architecture Fall 2016

Cache Replacement Policies

Least Recently Used and Random Replacement are the standard policies available for simulation in MARS. We added Most Recently Used and First In First Out.

Least Recently Used (LRU) replaces the block in the cache that was accessed the least recently. This algorithm keeps track of when each block was accessed using “age bits”, and tracking this can be expensive. When an item is accessed, LRU places it on the top of the cache. When the cache limit is reached, the items at the bottom of the cache are removed first.

Random Replacement (RR) randomly chooses which block in the cache to replace. It is very simple as it does not keep track of anything, and is therefore used in ARM processors.

Most Recently Used (MRU) replaces the block in the cache that was most recently accessed. This algorithm is designed for programs where older items are more likely to be accessed.

```
case MRU:
      int mostRAT = 0; // keep track of most recently accessed block
      for (int block = first; block <= last; block++) {
            if (blocks[block].mostRecentAccessTime > mostRAT) {
            mostRAT = blocks[block].mostRecentAccessTime;
                  replaceBlock = block; // at the end of the loop, this block
                  should have the highest mostRecentAccess time value
           }
      }
      break;
```

First In First Out (FIFO) replaces the block in the cache that was placed the earliest, with no reference to how frequently it has been accessed. This algorithm takes advantage of temporal locality.

~~~~
```

Queue<CacheBlock> fifoQueue = new LinkedList<CacheBlock>();


...

case FIFO:
default:
      CacheBlock targetBlock;
      if (!fifoQueue.isEmpty()) { // if the queue is not empty, the target
block to replace is the next block to be removed
            targetBlock = fifoQueue.remove();
            for (int blockNumber = first; blockNumber <= last; blockNumber++) {
                  CacheBlock block = blocks[blockNumber];
                  if (block.valid && block.tag==targetBlock.tag) {
                        replaceBlock = blockNumber;
                        writeLog("***FIFO replace block"+replaceBlock);
                        break;
                  }
            }

      } else {
            writeLog("FIFO Queue is empty");
      }
```
~~~~

Overview

After implementing and testing our FIFO and MRU implementations in MARS, we wanted to experiment with different cache design parameters to see how the replacement policy affected hit rate for different programs. To do this, we devised a set of tests using the tutorial programs developed for MARS (http://courses.missouristate.edu/KenVollmar/mars/tutorial.htm).

Method

To understand how different caching algorithms perform in different situations, we needed to be able to vary both the structure of the cache and the programs we are running using caching.

Programs

We used the three tutorial assembly programs provided with MARS because they display a wide range of behaviors.

row-major.asm traverses a 16x16 array of data by rows sequentially. This means that we will see recently used blocks being used again soon if the block size is greater than one because of the high spatial locality.

col-major.asm traverses a 16x16 array of data by columns sequentially. Recently used blocks will not be used until an entire column is traversed, which means it has very low spatial locality.

Fibonacci.asm is what we would expect a more typical program to be like; because of its recursive structure it exhibits significant temporal and spatial locality.

Parameters

We experimented only for fully associative caching because we wanted to test how much a replacement policy could improve the performance of a cache given the most freedom to replace whatever blocks it needed to. It is also easier to understand and test replacement in the context of a fully associative cache as opposed to an N-way associative cache of with high N, and irrelevant in the case of direct mapping or low N. For the purpose of running the Data Cache Simulation Tool, we held the set size at the default of 8 blocks.

For each program, we tried each caching algorithm with three configurations: the default value block size of 8, a block size of 2 (Small), and a block size of 32 (Large). We wanted to vary the size of the cache (number of blocks * block size), and we chose block size so we could see how loading multiple words at a time would affect the performance of the algorithms. We chose a range of block sizes to see how compulsory misses reduced with increasing block size as well as seeing how capacity misses differed with different algorithms, which required small block size.

Results

Program	Algorithm	Cache Structure	Percentage	Accesses	Hits	Misses
Row-Major	Random	Default(8,8,4)	75	256	192	64
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	50	256	128	128
	LRU	Default(8,8,4)	75	256	192	64
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	50	256	128	128
	MRU	Default(8,8,4)	75	256	192	64
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	50	256	128	128
	FIFO	Default(8,8,4)	75	256	192	64
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	50	256	128	128
Col-Major	Random	Default(8,8,4)	14	256	35	35
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	9	256	23	233
	LRU	Default(8,8,4)	0	256	0	256
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	0	256	0	256
	MRU	Default(8,8,4)	97	256	248	8
		Large(8,8,32)	9	256	24	232
		Small(8,8,2)	3	256	8	248
	FIFO	Default(8,8,4)	0	256	0	256
		Large(8,8,32)	97	256	248	8
		Small(8,8,2)	0	256	0	256
Fib	Random	Default(8,8,4)	94	139	131	8
		Large(8,8,32)	99	139	137	2
		Small(8,8,2)	76	139	106	33
	LRU	Default(8,8,4)	94	139	131	8
		Large(8,8,32)	99	139	137	2
		Small(8,8,2)	81	139	113	26
	MRU	Default(8,8,4)	94	139	131	8
		Large(8,8,32)	99	139	137	2
		Small(8,8,2)	60	139	83	56
	FIFO	Default(8,8,4)	94	139	131	8
		Large(8,8,32)	99	139	137	2
		Small(8,8,2)	80	139	111	28

Analysis

Row-major

To understand why some algorithms performed better in row-major and others performed better in column-major, we needed to understand how the programs worked.

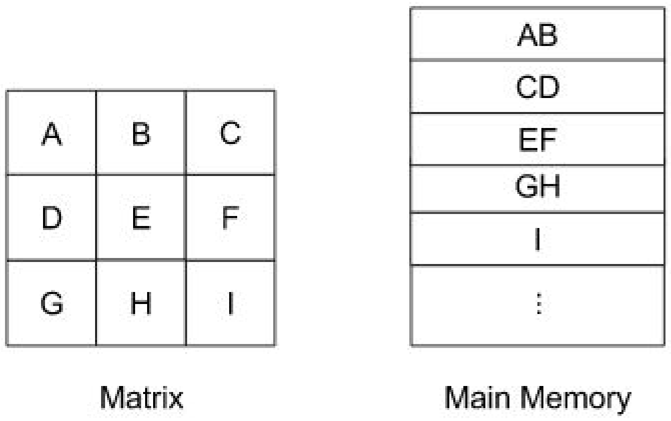

Figure 1: Matrix represents the matrices in row-major.asm and col-major.asm. Main Memory shows how the information would be stored if the block size = 2 words

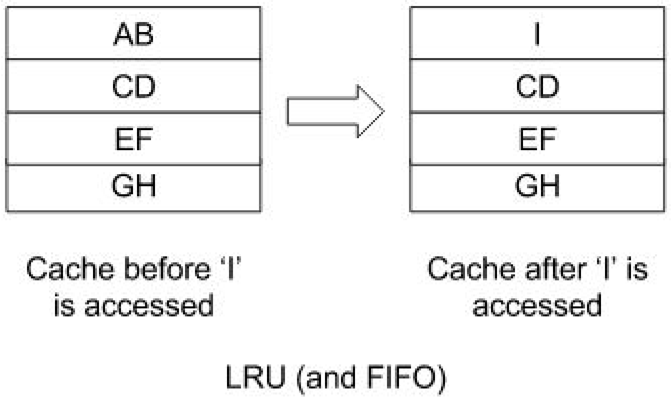

Figure 2: When 'I' is being requested from the memory and the cache is full, both LRU and FIFO choose to replace 'AB' with 'I'

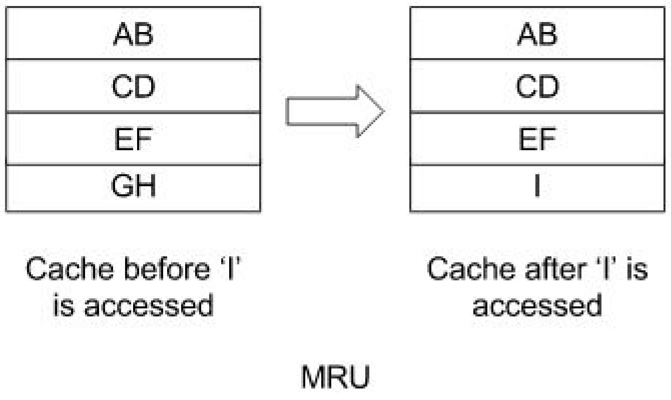

Figure 3: When 'I' is being requested from the memory and the cache is full, MRU chooses to replace 'GH' with 'I'

Because the least recently used (and also first in) block in row-major is not used for a long time, it makes sense to replace it. MRU is also not a problem in this scheme because even though a different block is replaced, the number of blocks needed to be replaced is the same. Regardless of which block replaced, performance for row-major is the same.

Column-major

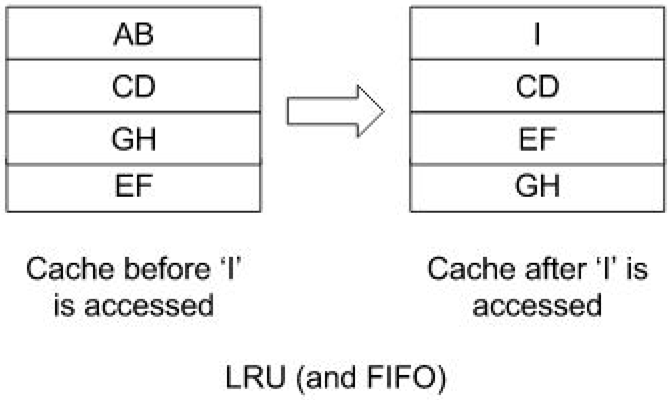

Figure 4: When 'I' is being requested from the memory and the cache is full, both LRU and FIFO choose to replace 'AB' with 'I'. Notice how 'GH' is requested before 'EF'

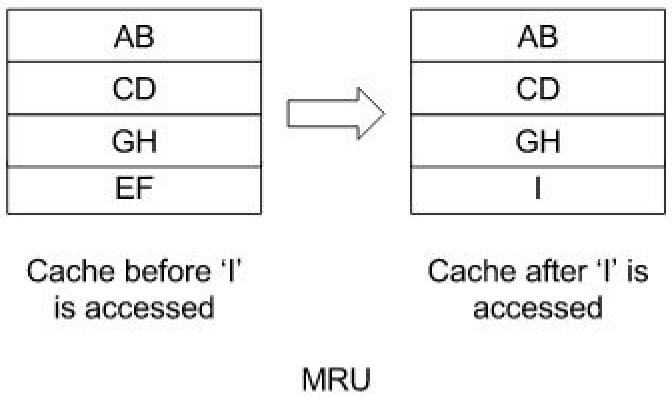

Figure 5: When ‘I’ is being requested from the memory and the cache is full, MRU chooses to replace ‘EF’ with ‘I’

The difference in the structure of the program for column major generates very different results from row major. Here, we see MRU performing the best and LRU and FIFO performing significantly worse than Random. This can be explained because in the case of sequential data access by column, the less recently something is used, the more likely it will be to be used soon in the future.

Fibonacci

For the Fibonacci tests, we noticed that the behavior was the same across algorithms for the Default (8) and Large (32) block size tests. This makes sense because our cache is large enough to hold all the necessary information to run the program, so the only misses were compulsory misses. We also saw that compulsory misses decreased with increased block size, which we expected.

For the Small (2) block size, the worst performing algorithm was MRU with a 60% hit rate, 21% below LRU. This indicates that the program has high temporal locality, which is consistent with our understanding of the implementation of Fibonacci in the program. Random performed slightly better than MRU with about a 75% hit rate. FIFO performed better still with an 80% hit rate, and LRU performed the best with a hit rate of 81%. This slight difference in FIFO and LRU is notable; for row and column major, FIFO was indistinguishable from LRU because the least recently used block was always the block that was first in as a consequence of the programs’ sequential access of data.

Conclusion

From looking at how each of these algorithms performed for various programs, we can draw some conclusions about how they work and make some recommendations on their usage. First, for short programs relative to cache size, replacement policies do not have much of an impact, and Random often does quite well. This means that only when the miss penalty is very high does it become important to consider heavily optimizing the replacement policy, especially if block size can be made larger easily. Most programs exhibit temporal and spatial locality in their structure, so algorithms that make use of that such as LRU and FIFO are far more useful generally than something like MRU, which assumes a lack of locality. MRU would only be recommended in the case of only running programs like column major. LRU is generally better for most programs than FIFO because it keeps track of not only when a block first enters a cache, but also when blocks are reused. If a block is used frequently in a short period of time, FIFO does not account for that. It could replace a block that was used just prior if it had entered the cache long enough ago, whereas LRU would keep it in the cache. This is useful for programs with lots of looping behavior. It is also interesting to note that it is hard to measure locality, but the difference in performance of algorithms like LRU and MRU could be used as a benchmark for measuring the locality of the program.