CS 211: Computer Architecture, Spring 2019

Programming Assignment 4: Cache Simulator

Due: May 6, 2019 at 11:55pm.

Overview

The goal of this assignment is to help you understand caches better by writing a cache simulator in C. The programs have to run on ilab machines. We are providing real program memory traces as input to your cache simulator. The format and structure of the memory traces are described below.

We will not give you improperly formatted files. You can assume all your input files will be in proper format as described.

Memory Access Traces

The input to the cache simulator is a memory access trace, which we have generated by executing real programs. The trace contains memory addresses accessed during program execution. Your cache simulator will have to use these addresses to determine if the access is a hit or a miss, and the actions to perform in each case. The memory trace file consists of multiple lines, each of which corresponds to a memory access performed by the program. Each line consists of multiple columns, which are space separated. The first column reports the PC (program counter) when this particular memory access occurred, followed by a colon. Second column lists whether the memory access is a read (R) or a write (W) operation. And the last column reports the actual 48-bit memory address that has been accessed by the program. In this assignment, you only need to consider the second and the third columns (i.e. you dont really need to know the PCs). The last line of the trace file will be the string `#eof`. We have provided you three input trace files (some of them are larger in size).

Here is a sample trace file:

```
0x804ae19: R 0x9cb3d40
0x804ae19: W 0x9cb3d40
0x804ae1c: R 0x9cb3d44
0x804ae1c: W 0x9cb3d44
0x804ae10: R 0xbf8ef498
#eof
```

Cache Simulator

You will implement a cache simulator to evaluate different congurations of caches. It should be able to run with different traces files. The followings are the requirements for your cache simulator:

- Simulate only one level cache, i.e., an L1 cache.
- The cache size, associativity, the replacement policy, and the block size are input parameters. Cache size and block size are specfied in bytes.
- Replacement algorithm: Least Recently Used (LRU). When a block needs to be replaced, the cache evicts the block that was accessed least recently. It does not take into account whether the block is frequently accessed.

- You have to simulate a write through cache.

Cache Simulator Interface

You have to name your cache simulator "first". Your program should support the following usage interface:

```
./first <cache size> <associativity> <cache policy> <block size> <trace file>
```

where:

- `<cache size>` is the total size of the cache in bytes. This number should be a power of 2.
- `<associativity>` is one of:
  - `direct` – simulate a direct mapped cache.
  - `assoc` – simulate a fully associative cache.
  - `assoc:n` – simulate an n-way associative cache. n will be a power of 2.
- `<cache policy>` Here the only valid cache policy is `lru`.
- `<block size>` is a power of 2 integer that specifies the size of the cache block in bytes.
- `<trace file>` is the name of the trace file.

Your program should check if all the inputs are in valid format, if not print "error" and exit.

Cache Prefetcher

Prefetching is a common technique to increase the spatial locality of the caches beyond the cache line. The idea of prefetching is to bring the data into the cache before it is needed (accessed). In a normal cache, you bring a block of data into the cache whenever you experience a cache miss. Now, we want you to explore a different type of cache that prefetches, not only bringing in the block corresponding to the access but also prefetches one adjacent block, which will result in one extra memory read.

For example, if a memory address 0x40 misses in the cache and the block size is 4 bytes, then the prefetcher would bring the block corresponding to 0x40 + 4 into the cache. The prefetcher is activated only on misses and not on a cache hit. If the prefetched block is already in the cache, it does not issue a memory read. With respect to cache replacement policies, if the prefetched block hits in the cache, the line replacement policy status should not be updated. Otherwise, it is treated similar to a block that missed the cache.

Cache Replacement Policy

The goal of the cache replacement policy is to decide which block has to be evicted in case there is no space in the set for an incoming cache block. It is always preferable – to achieve the best performance – to replace the block that will be referenced furthest in the future. There are different ways one can implement cache replacement policy. Here we use the least recently used (LRU) replacement policy.

LRU

Using this algorithm, you always evict the block accessed least recently in the set without any regard to how often or how many times it was accessed before. So let us say that your cache is empty initially and that each set has two ways. Now suppose that you access blocks A, B, A, C. To make room for C, you would evict B since it was accessed less recently than A.

Sample Run

Your program should print out the number of memory reads (per cache block), memory writes (per cache block), cache hits, and cache misses for the normal cache and the cache with prefetcher. You should follow the exact format shown below (pay attention to case sensitivity of the letters).

```
$ ./first 32 assoc:2 lru 4 trace1.txt
no-prefetch
Memory reads: 336
Memory writes: 334
Cache hits: 664
Cache misses: 336
with-prefetch
Memory reads: 336
Memory writes: 334
Cache hits: 832
Cache misses: 168
```

In this example, we are simulating a 2-way set associate cache of size 32 bytes. Each cache block is 4 bytes. The trace file name is trace1.txt. As you can see, the simulator should simulate both catch types (with and without the prefetcher) in a single run and display the results for both. Note: Some of the trace files are quite large. So it might take a few minutes for the autograder to grade all the testcases.

Simulation Details

-
  - When your program starts, there is nothing in the cache. So, all cache lines are empty (invalid).
  - you can assume that the memory size is 2^{48} bytes. Therefore, memory addresses are 48 bit (zero extend the addresses in the trace file if they're less than 48-bit in length).
  - the number of bits in the tag, cache address, and byte address are determined by the cache size and the block size.
- For a write-through cache, there is the question of what should happen in case of a write miss. In this assignment, the assumption is that the block is first read from memory (one memory read), and then followed by a memory write.
- You do not need to simulate the memory itself in this assignment, since the traces don't contain any information on data values transferred between the memory and the caches.
- You have to compile your program with the following flags: `-Wall -Werror -fsanitize=address`

Submission

You have to submit the assignment using Sakai. Put all files (source code + Makefile + report.txt) into a directory named first, which itself is a sub-directory under pa4. Then, create a tar file. Your submission should be only this tar file (named pa4.tar).

To create this file, put everything that you are submitting into a directory named pa4. Then, cd into the directory containing pa4 (that is, pa4's parent directory) and run the following command:

```
$ tar cvf pa4.tar pa4
```

To check that you have correctly created the tar file, you should copy it (pa4.tar) into an empty directory and run the following command:

```
$ tar xvf pa4.tar
```

You should have this folder structure:

```
pa4
└─first
   ├─Makefile
   ├─first.c
   ├─first.h
   └─report.txt
```

- Source code: all source files necessary for building your program, e.g., first.c and first.h.
- Makefile: There should be at least two rules in your Makefile:
  - first – build the executables (first)
  - clean – prepare for rebuilding from scratch
- report.txt: In a text file, you should briefly describe the main data structures used in your program. You should also report your observation on how the prefetcher changed the cache hits and number of memory reads, and why.

Grading Guidelines

This is a large class so that necessarily the most significant part of your grade will be based on programmatic checking of your program. That is, we will build the binary using the Makefile and source code that you submitted, and then test the binary for correct functionality against a set of inputs. Thus:

- You should make sure that we can build your program by just running make.
- You should test your code as thoroughly as you can. For example, programs should not crash with memory errors.
- Your program should produce the output following the example format shown in previous sections. Any variation in the output format can result in up to 100% penalty. Be especially careful to not add extra whitespace or newlines. That means you will probably not get any credit if you forgot to comment out some debugging message.

Be careful to follow all instructions. If something doesn't seem right, ask on piazza.