CSF 2019 Programming Assignment #6

Overview

For this assignment, you will be writing a cache simulator. The input to your simulator will be an abbreviated memory trace file. Each line in the trace file is composed of the memory access address and a flag indicating if the memory operation was a load (L) or a store (S).

As an example, consider the following three lines:

```
1fffff50 S
1fffff58 L
1fffff88 L
```

We first have a store to address 0x1fffff50, followed by a load from address 0x1fffff58 and another load from address 0x1fffff88.

The addresses are in hexadecimal and may be up to 16 characters long (64 bit addresses). There is no whitespace at the beginning of the line. Although in principle, the two fields are separated by an arbitrary amount of whitespace (spaces and tabs), for this project you may assume that the lines are no longer than 127 characters.

Cache types

For each trace file given to your simulator, you must simulate several caching techniques.

Direct mapped [DIR]

For this model, implement a cache with 8192 lines, with each line holding one memory address. Use the lower 13 bits of the memory access address as the index into the cache. Update the cache on both loads and stores. Always replace the cache entry with the current address on a miss.

Fully associative, least recently used [ASS]

For this model, implement a cache with 8192 lines, with each line holding one memory address. Keep track of how recently each entry was used, and if you run out of space, replace the least recently used with the new entry. (Note that you can keep track of how old an entry is by implementing a stack or by storing a time count of when the access last occurred.) For this model, update the cache on both loads and stores.

Set associative, least recently used [SET]

For this model, implement a cache with 2048 sets of 4 lines, with each line holding one memory address. Use the lower 11 bits of the memory access address as the index to select the set. Keep track of how recently each entry in each set was used, and if you run out of space in a set, replace the least recently used entry in the set with the new entry.

Blocked set associative, least recently used [BLK]

For this model, implement a cache with 256 sets of 4 lines, with each line holding 8 memory addresses. Use the lower 3 bits of the memory access address as the offset within the block, and the next 8 bits of the memory access address as the index to select the set. Keep track of how recently each block in each set was used, and if you run out of space in a set, replace the least recently used block in the set with the new block.

Write-through no write-allocate [NWA]

For this model, implement the blocked set associative [BLK] model, where the stores do not cause cache updates. That is, on a store, only update the cache hit and miss counters.

Prefetch [PRF]

For this model, implement the write-through no write-allocate [NWA] model with prefetch. Implement a prefetch algorithm that always fetches two consecutive blocks from memory. That is, on a cache miss, if block n is needed, fetch blocks n and $n + 1$.

Requirements

1. Your code must compile with the command `make` and produce the executable called `cache`. Among other implications, this means that your source code must include a makefile. (Note that `make` must return 0, so do not include a compiled version of your code.)
2. Your code may not generate errors or warnings when compiled.
3. Your code must be able to accept the trace file directly on `stdin`. Your code must produce its output on `stdout`.
4. Your code may not crash. If you encounter an error, your code must exit cleanly with a status of 1 and an explanatory message. (For this assignment, you do not need to check for any errors, but you can optionally return an error if there are too many arguments, or if an input line is longer than 127 characters.)
5. If you want partial credit or CA assistance, your code must be readable. **CA are allowed to refuse to review messy and/or unreadable code!**

Output format

For each caching technique, you must keep track of the number of cache hits and the number of cache misses.

The format is as follows:

```
DIR:                469498              45190
ASS:                478873              35815
SET:                476984              37704
BLK:                490951              23737
NWA:                468405              46283
PRF:                474529              40159
```

The first column lists the predictor, the second lists the number of cache hits and the third lists the number of cache misses.

The counts are 20 characters wide (to handle 64 bit counts), plus a single space separating the fields. (See coding tips at the end of this document.)

Examples

The supplied example input files (and corresponding output files) are:

- `ex1_in/out`: Shows the benefit of caching, with sixteen consecutive loads from the same address. All models will miss once at first and always hit thereafter.
- `ex2_in/out`: Shows the problem with direct mapped caches when alternating between two addresses that map to the same cache line.
- `ex3_in/out`: Shows the benefit of fully-associative caches when a large number of addresses map have the same lower bits.
- `ex4_in/out`: Shows the ability of set-associative caches to handle modest numbers of addresses with the same lower bits.
- `ex5_in/out`: Shows the benefit of blocking.
- `ex6_in/out`: Shows the benefit of having writes misses bypass the cache.
- `ex7_in/out`: Shows the benefit of prefetching.
- `ex8_in/out`: Input is a short trace file extracted from a gcc simulation. See Prof. Steven Swanson, UC San Diego.

Coding tips

The input format has been chosen so that you can use:

```
scanf("%llx %c",&address,&flag);
```

to read the input lines. The output format has been chosen so that you can use:

```
printf("XXX: %20llu %20llu\n",hits,misses);
```

to print the output lines.