Multi-level cache simulation project

UVM CS 222, Spring 2015
Due February 26, 2015

You are asked to write a multi-level cache simulator to perform an untimed simulation of a sequence of instruction read, data read, and data write accesses and to determine the number of accesses, number of misses, and miss rates of the various caches.

The value of simulation to a computer architect is in its ability to quickly and cheaply explore a variety of configurations without large effort, time, and cost required to buid actual hardware. Thus, your simulator should be parameterized to allow varying at least:

- Size (bytes)
- Associativity
- Block size
- Replacement strategy (implement at least FIFO; you may also want to implement random, LRU, and pseudo-LRU)
- The cache configuration, including:
  - The number of cache levels and the parameters of each
  - Split I & D caches or combined caches at the first level(s)

You may also want to explore different variations on write allocate vs. no write allocate, and write back vs. write through., but you should implement at least one of these two combinations:

- no write allocate with write through
- write allocate with write back

Write the simulator (in any language of your choosing, but I strongly suggest an OO language) and turn in the code. As always, your code should be well commented so I can understand it (and so you can too, should you go back and look at it months or years later!).

<u>The cache model:</u>

Using an OO language or style, write an interface (e.g., base class) to which you can send access requests. During construction or initialization of the cache object, pass the configuration parameters for the particular cache, and include a reference or pointer to a downstream instance of the interface (e.g., the next level cache) to which miss accesses will be passed. Building it in this way makes it easy to move from a single cache simulation to a multi-level one. I found it convenient to have each cache model object collect its own statistics.

<u>The driver:</u>

I will provide trace files with lines in the form:

```
<access_type> <hex_address>
```

An example:

```
I 48d23d
R 12ff200
I 48d240
I 48d248
W 7fffe7fefc8
```

I will also provide some configuration files I used in my version of the program, which you may use if you wish, or you may provide other means to input the cache parmateres and configuration. My files are of the form:

```
        <cache_type> <cache_level> <parameter> <value>
```

It also may also include comments beginning with "# ". An example, which includes all the parameters my model accepts:

```
        i 1 size 4096
        i 1 block 16
        i 1 ways 2
        i 1 replace oldest
        i 1 writeback yes
        i 1 writealloc yes
        d 1 size 4096
        d 1 block 16
        d 1 ways 2
        d 1 replace oldest
        d 1 writeback yes
        d 1 writealloc yes
        c 2 size 131072
        c 2 block 64
        c 2 ways 4
        c 2 replace oldest
        c 2 writeback yes
        c 2 writealloc yes
```

Values for <cache_type> are 'i' (instruction), 'd' (data', or 'c' (combined). Values for replace are "oldest", LRU", "random", and "pseudo-LRU". I required (and you may also) that block size, and size / ways are both powers of two (this allows separation of the tag, index, and block offset fields by simple bit masking of the incoming address). If you are implementing only one replace strategy and/or write strategy combination, you can ignore those parameters. My program requires that parameters are provided for all contiguous levels from 1 to the maximum in the file, that any particular level have i & d caches or a c cache but not both, and that there is some first level above which all caches are combined, and below which all are split (it is OK to have all levels split or all levels combined).

You should write a driver program which reads configuration information (in the above form or your own), constructs and links together the specified cache models, reads a trace file and passes the accesses to the cache(s) one by one, and prints summary data for each cache, including (at least) the number of read and write accesses, and the local miss rates.

I strongly suggest you write your simulator in stages, getting one stage working before adding features in the next stage. Here are my suggestions:

Stage 1:

Write a simple implemention that supports read only queries and one replacement strategy, but can vary size, block size, and associativity. Start by testing a single combined cache, ignoring (or treating as reads) all write accesses. Create a trivial memory model implementation of the interface that accepts all accesses and never misses (never passes them on), or pass a NULL for the downstream interface and recognize that misses cannot be passed on.

You will probably want to add some verbose mode to print lots of details of each access to help you in debugging.

I've provided two very short trace files that I used in debugging my model, one with only reads, and one with reads and writes, and some small single cache configration that let me trace by hand all operations I expected to see, and compare that with the verbose output provided by my model (...read... files were usd at this stage, others were used when I added write modeling):

    read.trace
    1level-read-fifo.config
    1level-read-lru.config
    1level-read-plru.config
    1level-read-random.config
    read-write.trace
    1level-nwa-nwb-fifo.config
    1level-nwa-wb-fifo.config
    1level-wa-nwb-fifo.config
    1level-wa-wb-fifo.config

Stage 2:

Test the multi-level operation by creating two caches (an L1 and an L2) and verify that L1 misses are properly passed to L2.

Test the ability to model split I & D caches by creating two L1 caches (I & D) and havng both of them pass on misses to a common L2.

Stage 3:

Starting again with a single cache simulatio, implement a write strategy. Remember the operations that occur for a read and write hit and miss for each combination of write strategies for L1, backed by an L2:

Read access (write back):

Hit:	Read from L1
Miss:	If dirty, write back from L1 to L2
	Read from L2 to L1

Read access (write through):

Hit:	Read from L1
Miss:	Read from L2 to L1

Write access (write back / write allocate):

Hit:	Write to L1
Miss:	If dirty, write back from L1 to L2
	Read (line fill) from L2 to L1
	Write to L1

Write access (write through / no write allocate):

Hit:	Write to L1
	Write to L2
Miss:	Write to L2

Write access (write back / no write allocate):

Hit:	Write to L1
Miss:	Write to L2

Write access (write through / write allocate):

Hit:	Write to L1
	Write to L2
Miss:	Read new cache block (line fill) from L2 to L1
	Write to L1
	Write to L2

<u>Stage 4:</u>

Put it all together by building performing a multi-level simulation including write behavior.

I've provided a few testcases with the results I got for each from my simulator with my base configuration various configurations. Note that to the extent that there is any random component in the simulation (random replacement, or psuedo-LRU using random replacement among the not-most-recently-used alternatives), you may not get exactly the same results.

- base.config
- gcc-addrs-10K-a.trace
- gcc-addrs-10K-b.trace
- base-10K-a.output
- base-10K-b.output

Exercise your model using the base configuration above and the following configurations (each of which varies some parameter from the base configuration used above) and trace files I provide (without my results). If you did not implement certain replacement strategies or write options, you can skip the configurations for those features. All the "gcc..." testcases are adapted from http://www.cis.upenn.edu/~milom/cis501-Fall12/traces/trace-format.html. Note that while the longest one I provide is about 10M, this is ***very*** small for real cache simulations. The "random..." testcase is a set of randomly generated access (74% instruction, 20% data read, 4% data write) within a 500 M address space.

- gcc-addrs-10K-c.trace
- gcc-addrs-10K-d.trace
- gcc-addrs-10K-e.trace
- gcc-addrs-10K-f.trace
- gcc-addrs-10M.trace
- rand-500Mspace-10K.trace
- base-2kL1.config
- base-8kL1.config
- base-16kL1.config
- base-32kL1.config
- base-64kL2.config
- base-512kL2.config
- base-lru.config
- base-plru.config
- base-random.config
- base-32B-block.config

base-128B-block.config
base-nwb-nwa.config

Stage 5 (optional):

Explore different replacement strategies and different write strategies.

Try these configurations, which are as close I as I could come to actual configurations in at least some versions of commercial processors:

i7.config
opteron.config
z10.config

Include anything interesting in your write up.

You should hand in:

- Your code
- A write up containing:
  - A brief description of the organization of your code, and any particular problems you encountered or discoveries you made while writing it.
  - Results for the various configurations and testcases. Include at least reads, writes, misses, and local miss rates for all caches in ach configuration.
  - A discussion of the comparison of results from real trace data (gcc...) and the random trace data.
  - A brief discussion of the effects you saw while varying parameters, and your explanation for why you believe these occurred.