Assignment #1: Simulation of a Hardware Cache

CS222 Spring 2022
40 points
Part I: due Saturday, Feb. 19th (20 points)
Part II: due Wednesday, Mar. 2nd (20 points)

1 Simulating a Hardware Memory Cache

You'll write a program to simulate the behavior of a hardware memory cache. You can use the language of your choice. You may work either by yourself or with a partner, except students taking the course for graduate credit must work individually.

1.1 Memory hierarchy

Model memory as an array of bytes. Each memory access will read or write one word (four bytes). Check that each address is aligned on a four-byte boundary, and assert if it isn't. Also check that each memory access is in range, and assert if it isn't.

In Python, use a `bytearray`; in C use a `char[]`.

You'll model a little-endian system, with 32-bit words: so for example if `memory[56] = 45` and `memory[57] = 12` and `memory[58] = 3` and `memory[59] = 7`, then the word referenced by address 56 would be $45 + 256 * (12 + 256 * (3 + 256 * 7)) = 117640237$.

When a range of values is loaded into the cache, the number of bytes loaded will thus be a multiple of four.

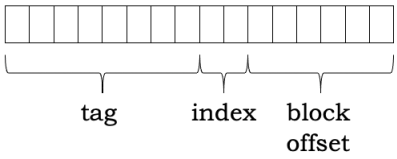
1.2 Cache structure

Recall how the index, tag, and block offset are computed from an address. For example suppose we have the following:

- a 64K memory (and 16-bit addresses)
- a 1K cache
- 64-byte cache blocks
- four-way associativity

Then:

- there are $2^{10}/2^6 = 16$ cache blocks
- and $16/4 = 4$ cache sets
- so we need two bits for set selection (the index)
- and six bits for byte selection in a block
- which means that the tag is $16 - 2 - 6 = 8$ bits in length



1.3 Parameters

Your simulator should let you specify these parameters:

- the size of a memory address, in bits (and then set the size of the memory to 2 to the power #bits)
- the size of the cache, in bytes
- this size of a cache block, in bytes
- the associativity of the cache
- whether the cache is a write-back or write-through cache

Do not hardcode any of these values —they should be configurable. Use a variable or a `#define` to represent each of them.

But, assume a write-allocate cache: on a write miss, the block containing the memory address in question is first brought into the cache.

1.4 Data structures

Here's how I approached this: by considering the fundamental component of a cache to be a set, consisting of one or more cache blocks. Each set then has k blocks, where k is the associativity. So, for example, a two-way associative cache of size 64K with 64-byte blocks will have 1024 blocks and 512 sets. (And a direct-mapped cache has $k = 1$.)

So, you can model your cache as a group of sets (for example, as an array of sets). You might instead choose to model the cache as a group of cache blocks, with some efficient way to mark set membership for each block.

Each cache block will need a tag and two additional attributes: dirty/clean and valid/invalid.

The cache data structure itself can be a global variable.

The memory is just an array of bytes. It can be a global variable.

In Python, create a class for cache, cache set, and cache block. In C, these can be structs. (Yes, you may use C++ or Java if you want.)

Use the least-recently-used (LRU) algorithm to control block replacement in a set: if you need to replace one of the blocks in a set, then pick the block that was least recently used. The key data structure you'll need in order to implement LRU is a tag queue. The tag queue can be an array of integers.

Each set will have a tag queue. Initialize the tag queue for each set to invalid values, such as -1. (Zero is a valid tag, so you must not initialize the tag-queue entries to zeros.)

Here's an example of how the queue will work. Suppose we have a four-way associative cache: then the tag queue for each set will have four positions. Now suppose the queue for a particular set is currently [4, 8, 12, 16], and that the most recently accessed tag is always kept in the last position.

- then after an access having tag=4, the queue will be [8, 12, 16, 4]
- and then after an access having tag=12, the queue will be [8, 16, 4, 12]

- and then after an access having tag=4, the queue will be [8, 16, 12, 4]
- and then after an access having tag=6, the queue will be [16, 12, 4, 6]
- and then after another access having tag=4, the queue will be [16, 12, 6, 4]

In this way, the least-recently accessed tag is always in the first position. When you need to replace one of the blocks in this set, you'll replace the block having the tag that is in the first position.

At a high level, your cache consists of a group of arrays representing the cache blocks. Each of these arrays will have auxiliary information with it (a tag, the dirty/clean bit and the valid/invalid bit, and some way to show which set this array belongs to). A read from memory thus consists of copying a range of values from your memory array to one of the arrays representing a cache block. A write to memory consists of copying a value to one of these arrays. The setting for write-through vs. write-back will determine when and if you also copy data from one of the cache-block arrays back to your memory array.

All reads or writes will be for a single word (four bytes). A write-through cache will write a single word to the memory (in addition to writing the word to the cache).

1.5 Algorithm

Here's the high-level algorithm for an access to memory location A

```
compute the block offset b, index i, and tag t
check each tag in set i
if t is found at position j in set i and the valid flag at position j is set
  // this is a cache hit
  // depending on whether this is read or write, do one of the following
  either read the specified word or write the specified word
  // on a write: for a write-through cache, write the word also to memory
  // on a write: for a write-back cache, mark the block as dirty
  set the tag for this block
  set the valid flag for this block
  update the tag queue for this set
else
  // cache miss
  // try to find an unused block in set i
  if the valid flag for any block is set i is false
    // use that block
    update the tag queue
    set the valid flag for this block to true
    if this is a read
      read a cache block from memory into that block
    else
      write to that cache block
      if a write-through cache then write to memory also
      set the dirty flag for this block
      set the tag for this block
  else
    // must evict a cache block
    find the least-recently used block, by checking the tag queue for this set
    update the tag queue
    set the tag for the target block
    if this is a write-back cache
```

```
  if the target block is dirty
    write back the block
if this is a read
   read the correct block from memory into this block
else
  // this is a write-allocate cache, so do the following
  read the correct block from memory into this block
  write the value to the block
  if this is a write-through cache
    write the value to memory also
```

1.6 Functions

Create two functions: one to read a word from memory, and one to write a word to memory.

For example: in C, my functions would look like this:

```
Word readWord(unsigned int address);
void writeWord(unsigned int address, Word word);
```

`Word` is a typedef to an `int`.

In Python, I might do this:

```
read_word(address)
write_word(address, word)
```

where `read_word()` returns the value read from the specified address and `write_word()` writes the provided word to the specified address.

Again, check each address for four-bit alignment and for range $0 \leq address < memSize$, where $memSize$ is the number of bytes in the memory.

Almost all of the processing of these two functions is identical, so it makes sense to write a single underlying function that each of these can call.

1.7 Output

In order to observe the behavior of the simulated cache, print output describing what happens in response to a read or write. For example, with a 64 K memory (16-bit addresses), a 1K cache, 64-byte blocks, and associativity = 1 (a direct-mapped cache), then in response to a read from address 56132, your functions should print out a string in this form (details will depend on whether this is a hit or miss, and whether an eviction is necessary):

```
read miss + replace [addr=56132 index=13 tag=54: word=56132 (56128 - 56191)]
```

If there is a read or write miss with a replacement necessary, then print out which tag, in which block index was evicted (the block index is the index of a block in its set):

```
[evict tag 4, in blockIndex 0]
```

And after each read or write, print the tag queue for the set that was accessed, in this format:

```
[ 12 20 32 54 ]
```

Check: 56132 = 110110 1101 000100, so the block offset is 000100 = 4; four bits are needed for the index (1024 / 64 = 16), giving the index 1101; and the tag is 110110 = 54.

Here's another example of the information you should print:

```
read miss + replace [addr=17536 index=2 tag=68: word=17536 (17536 - 17599)]
[evict tag 32, in blockIndex 1]
[write back (8320 - 8383)]
[ 36 44 64 68 ]
```

Print the eviction information only for a read or write miss + replace, and print the write-back info only for a write-back cache (and only if the evicted cache block is dirty).

Here's an example of the information to print for a write:

```
write miss + replace [addr=8320 index=2 tag=32: word=7 (8320 - 8383)]
[evict tag 28, in blockIndex 1]
[write back (7296 - 7359)]
[ 16 12 20 32 ]
```

And again, print the eviction information only for a miss + replace, and print the write-back info only for a write-back cache (and only if the evicted cache block is dirty).

1.8 Other notes

Initialize memory so that `memory[i] = i/4` (integer division) for each four-byte aligned value i with $0 \leq i < memSize$, where i is a four-byte integer. So in this way, `memory[i]` holds the lowest-order byte of $i/4$, `memory[i+1]` holds the second-lowest-order byte, etc. This way, if I read the four-byte value memory address a, the value will be a.

Initialize the tag queue for each block to have -1 in each position (not zero, since zero is a valid tag).

Initialize the tag for each block to -1, for the same reason.

Set the valid flag for each block in the cache to `false` initially.

2 Part One

Implement a direct-mapped cache that supports only read hits. To test this, "prefill" the cache with specially chosen values in specially chosen locations. For example, suppose I have a direct-mapped cache with this structure:

- 64 K memory, with 16-bit address
- 1024-byte cache
- 64-byte cache blocks

This means the cache contains 1024/64 = 16 cache blocks. Six bits are required for the block offset, and four bits for the index. This leaves $16 - 4 - 6 = 6$ bits for the tag.

Consider the address 46916 = 101101 1101 000100. The block offset is 4, the index is 13, and the tag is 45. So if I store a value at the four bytes starting at `cache.sets[13].blocks[0].data[4]`, then when I read from address 46916, I should get that same value. (In my program, each set has one or more blocks; in a direct-mapped cache, each set has a single block.)

Another example: the address 13388 corresponds to 001101 0001 001100, so the block offset is 12, the index is 1, and the tag is 13.

Print the block offset, index, and tag for each address, and make sure that you are calculating these correctly.

3 Part Two

Handle read and write misses, associative caches, he tag queue, and write-through/write-back behavior.

4 Testing

In the class gitlab site, you'll find three files: `testAwb.out`, `testBwb.out`, `testCwb.out`; and `testAwt.out`, `testBwt.out`, `testCwt.out` (wb is write-back; wt is write-through). They show my output for a sequence of reads and writes with various cache configurations and various read and write sequences. Verify that you are getting the same results.

5 What to Submit

Submit your source code.

6 Graduate Students

Students taking the course for graduate credit, and undergraduates who want a bit of extra credit: add a command-line option that will represent the name of an ASCII file. This file will have on each line a nonnegative base-10 integer representing a memory location, followed by a space and either `R` or `W`, depending on whether it represents a read or a write to that memory location. If a filename is provided, then process each line of the file using your cache simulator. Keep track of reads, read misses, read hits, writes, write misses, and write hits. At the end, print statistics, like this:

```
# reads = 17600
# read misses = 2662 (15.12%)
# read hits = 14938 (84.88%)
# writes = 8800
# write misses = 142 (1.61%)
# write hits = 8658 (98.39%)
```

Then, implement cache blocking, as shown on pp. 107-109. Use $N = 20$, and pick block sizes of $B = 2, 4, 6, 8, 10$.

Put the three arrays in global memory, with fixed size $N \times N$, and put `r` also in global memory, so that it will be near the arrays. Create an address trace by writing out the address of each element when it's accessed, along with `R` or `W`, depending on whether the access is a read or a write. Do this also for `r`. In C or C++, you can use the `&` operator to print the actual virtual addresses of the array elements. In Python, you can use the function `id()` to get the virtual address of a variable.

Then, normalize the addresses: subtract the minimum address value from each address, which will move the range of the addresses from $[min\ a_i, max\ a_i]$ to $[0, max\ a_i - min\ a_i]$. This will make the values more manageable in your cache simulator. You'll probably still have to increase the range of possible memory

addresses in your cache simulator though, from 2^{16} to perhaps 2^{24} (it depends on the virtual memory of the machine where you run your array-blocking code). You can do the normalization either in the array-blocking program (by saving all of the addresses and processing them before you write them out) or else as a separate step before running the normalized addresses through your cache simulator.

For this exercise, it's not necessary to have an actual array representing the memory—here, we care only about the cache miss rate.

Pick some particular implementation of a cache (in terms of associativity, size, block size, and write-back/write-through). Plot your results.