Computer Organization

Cache Simulator

Authors

Gonçalo Azevedo 93075, Tomás Cruz 103425, Rodrigo Friães 104139

1. Directly-Mapped L1 Cache

For the first part of the assignment we developed a directly mapped L1 Cache. The cache specifications are, 16 words (4B in size) per cache block and a cache capacity of 256 blocks. This equates to 16 KB of cache capacity.

This cache can only load and store words in a similar fashion to the “lw” and “sw” MIPS instructions so, even though memory is considered byte addressable, in reality, our cache can only address words. Because of this, the offset bits that should be used to address singular bytes within a word are not used, so the two least significant bits of every address are useless.

The cache starts by splitting the address into it’s components:

I. *Tag*: 18 bits, A31 - A14
II. *Index*: 8 bits, A6 - A13
III. *Offset*: 6 bits, A0 - A5

We use the *Index* to identify the line of the desired block in Cache. The *Offset* is then used to address the desired word within the block. The cache uses the *address’s Index* to look into the cache line destined for the memory block requested. If the block is not present we have a **cache miss** and we will proceed to load the block from DRAM into the L1 cache. If the line destined for this new block is currently being used and also contains a dirty block in it, then, a write back is issued into the DRAM memory in order to preserve the newly written contents of the replaced block. If the block is present, we have a **cache hit** and we don’t need to access the DRAM memory.

Once we know the block is surely in L1 (by fetching it from DRAM or by “hitting it”) we will use the address’ O*ffset* to determine which word we are targeting. The 4 most significant bits in the *Offset* will give us the position of the word in the block. Issuing a *shift right* operation by 2 bits will give us the number indicating the word’s position which we will use to address the desired word bytes.

If we are reading from memory the word will simply be copied into the function argument to be returned to the caller, if instead we are writing to memory, the word passed as argument will be written into the desired word position within the block currently in the L1 cache and will only be written into DRAM if this block is replaced, which may never happen because our cache uses a **write-back** policy.

2. Directly-Mapped L2 Cache

For the second part of the assignment we implemented a directly-mapped L2 cache on top of the directly-mapped L1 cache, simulating a slightly more complex memory hierarchy.

The L2 cache has blocks of the same size as L1 but it has a capacity of 512 blocks, double the amount of L1.

Aside from the size, the L2 cache is exactly like the L1 cache described above, since it is directly mapped and also uses a write-back policy. It splits the address and maps each block onto its corresponding line. The L2 cache only handles entire blocks, so the address *Offset* bits are never used. The *Index* also requires an extra bit since there are double the amount of blocks of L1, this also means the *Tag* has minus one bit.

On a **cache miss**, the L2 cache will fetch the desired memory block from DRAM and load it into L2. On **cache hit** it will simply supply the entire block to the higher level of the memory hierarchy, the L1 Cache.

With L2 implemented the memory hierarchy now works the following way:
Once memory operations are performed we always look into the L1 Cache, if it the desired data is there, we continue our operations, if it isn't, then we go down the hierarchy levels, looking for it in L2.

If the desired block is in L2, the data is copied into the L1 Cache and, if any conflict happens when reading this new memory block into L1, then, the replaced L1 block is evicted to L2.

If the block is also not found in L2, we look into DRAM, and then load it into L2 and from L2 into L1, evicting any conflicting blocks from L1 into L2 or writing replaced dirty blocks from L2 to DRAM if necessary. Finally, we continue our operation.

This way, L2 is only populated when we either: (1) have to access DRAM, or (2) when a block is evicted from L1. This two can happen on the same memory address request or not.
An example:
If we have a block on L2 and another on L1 and they conflict on the same L1 line, issuing a memory access to the block in L2 will load it into L1 and force the replaced L1 block to be evicted to L2, regardless of it being dirty or not.

A more technical example, issuing a 0x4000 write and 0xc000 read followed by a 0x8000 read. This operations will result in:
Write 0x4000: 0x4000 is read into L2 and L1 and, finally, written in L1 making it dirty.
Read 0xc000: 0xc000 is read into L2, 0x4000 is replaced and it isn't written into DRAM since it isn't marked as dirty in L2. 0xc000 is also read into L1, conflicting with 0x4000 again. Now, 0x4000 is evicted into L2, replacing 0xc000 that was just read into it. Now we have 0xc000 in L1 and 0x4000 in L2.
Read 0x8000: This situation will cause havoc. It will be read 0x8000 into L2, which will conflict with 0x4000, requiring a DRAM access to write 0x4000 into memory since it's dirty, and then, conflict with 0xc000 in L1, evicting 0xc000 into L2 and loading 0x8000 into L1.

3. **2-Way Set Associative L2 Cache**

Finally we turned our L2 directly-mapped cache into a **two way set associative** L2 Cache. It is in everything similar to the previous L2, in the sense that it reads from DRAM and writes to L1, being populated only by L1 evicts or DRAM reads.

The only difference now is that we have sets of two blocks, instead of one. Because of this one *Index* bit is lost and it is added to the *Tag*. This new L2 cache uses a **LRU replacement policy** and the same **write-back policy** as before. To turn the previous L2 cache into a two way set associative we only need to add a timestamp information to each line, in order to identify the least recently updated line in each set. Now, whenever addressing a memory position, we can't directly call it a hit or miss, we have to look through all the blocks in the set (which happen to be only two in this case).

After we determine if it is a hit or a miss, it will proceed as usual, read from DRAM and/or write to L1. Now, whenever a block is evicted or read into L2, we will need to scan the set and see if there are free lines to add the block. In our implementation it will usually select the last free line i.e. the free line with the highest position in the set. If there aren't any free lines, then, we will need to replace an existing block with the new one, which will require us to identify the least recently updated block and replace it. After identifying the least recently update block, by comparing the timestamp values of each line, the replacement is in everything similar to the previous directly-mapped L2 Cache, i.e. replace the block and write back the replaced block into a lower level of the memory hierarchy if it is dirty.