INF-2200 Computer Architecture and Organization

Assignment №3: Cache simulator

02 november, 2017

1. Introduction.

In this task, we need to implement the processor memory-cache hierarchy, and using simulator, determine the best cache architecture for the benchmark from the 1st assignment.  
As a benchmark, I chose the program for finding prime numbers using the Sieve of Eratosthenes. The search range of this benchmark is a several millions of numbers, which is close to real values.

1. Design.

I decided to present the cache in the form of a structure that includes such important values as: cache size, associativity, block size, etc., as well as hits and misses of readings and writes. Plus, each cache contains an array of block structures. The block itself consists of a tag, age, data, a dirty bit and a valid bit. A block tag is needed to identify a specific block of all. Age indicates the relevance of each block. Dirty bit and valid bit help to determine whether the given cache block is empty or not.

So, at the very beginning, we initialize our memory cache hierarchy, allocate memory for every block of cache. The two upper layers are the instruction cache and the data cache, which are not connected to each other, and the next cache - level 2 cache is connected to each of these caches.

After initialization, the program calls one of three functions: fetch\_memory, read\_data, write\_data.  
But before using one of these functions, It first transforms the received address into tag, index and offset of this address.

The address offset always occupies Log2(block size) bits, the address index always occupies Log2( number of blocks in the memory cache / associativity) bits. Knowing how many bits occupy the offset and the index, we can find the tag by bitwise shifting to the right. We shift to the right our address to the total number of bits that occupy the offset and index. Then, by subtracting the index from our address and performing a bitwise shift to the right by the number of offset bits, we find the address index. And having combined the index with the tag and subtracting the amount from the address, we find the offset of the address.

When we write these tag, index and offset into global variables, it would be much easier to find necessary block in the cache.

**Fetch\_memory**:  
First we process the address. Once we find all the values, we begin the search in the instruction cache. If this cache already has this block, we increase the number of hits and reset the age of this block to 0. If it does not exist in the first cache, we increase the number of misses in this cache and begin the search in the second cache. If it is in the second cache, then we find the available free block in the first cache and place this block from the second cache there. If, during the transfer from the second cache, all blocks in the first one are full, then we use LRU\_search function, which finds the most unused block from the first cache and changes it with the second cache block. If there is no such block in either the first or the second cache, then we add a new block to the first level cache. In the end, we increase the instruction counter and perform the function of aging.  
  
**Read\_memory and write\_memory**: the design is the same as in fetch\_memory, but now we use another cache - the Data Cache instead Instruction Cache.

Some other functions:

**Find\_log**: find log2 of input number. Unfortunately, C library does not support this operation.

**Aging**: look through all blocks of each cache, and if current block is not empty(valid\_bit == 1), then increase the age of this block by one.

**LRU\_search**: find the oldest block in the set of some cache and return the tag of this block.

**Change\_block**: swaps the block with input tag of input cache (L1I or L1D) with input tag2 of L2 cache.

1. Implementation.

As I mentioned before, Cache has a structure form, which includes cache options plus blocks, which are also structures. This solution has replaced the creation of a huge number of global variables. Plus, each function would have to perform operations on all caches simultaneously, rather than on each one separately.

1. Discussion.

While I was testing my program, I found a strange bag. In memory\_init() the program allocate memory for each block and everything works OK. But, when I try to use aging() function, which update all used blocks or when I need to change dirty\_bit, tag, valid\_bit or etc. of some blocks from L1 cache, segmentation fault happens. I found that bag appears, when I work with blocks from 0 to 27 from L1 Instruction cache and blocks from 0 to 13 of L1 Data cache. I though that problem might be with not enough memory, however L2 Cache is much bigger and all blocks allocated correctly.

Among a large number of replacement policies, I chose LRU because it is one of the “honest” replacement method that allows you to find the most unnecessary blocks and not to change the frequently used blocks. This leads to a reduction in misses. However, this algorithm requires to look through all caches’ blocks, after each access to the memory, which increases the execution time.

1. Evaluation.
2. Conclusion.