University of Tromsø

INF-2200

ahe103

2. november 2016

1 Introduction

This report covers in broad and narrow strokes the simulation of a cache structure between a processor and memory. This cache structure consists of several layers of increasingly small and fast memory, reading addresses from an input trace file.

1.1 Requirements

The requirements set for this simulation consists of:

2 layers of cache memory(L1 Instruction, L1 Data and L2 Unified cache
Variable size of parameters (Cache size, Block size, n-Way associativity)
Write Back and Write Through policies
At least 1 replacement policy

1.2 Technical specifications

Operating System: 16.04 Ubuntu, Linux, 4.4.0-45 generic

Languages used: C, Python 2.7

Additional utilities: GNU Debugger, Valgrind

2 Cache Simulation Implementation and Design

The architecture deemed most logical for this simulation was arrays and structs, as memory is no problem to get a hold of. The layout needed would need to be a cache struct with an array of sets that contained an array of blocks as seen below.

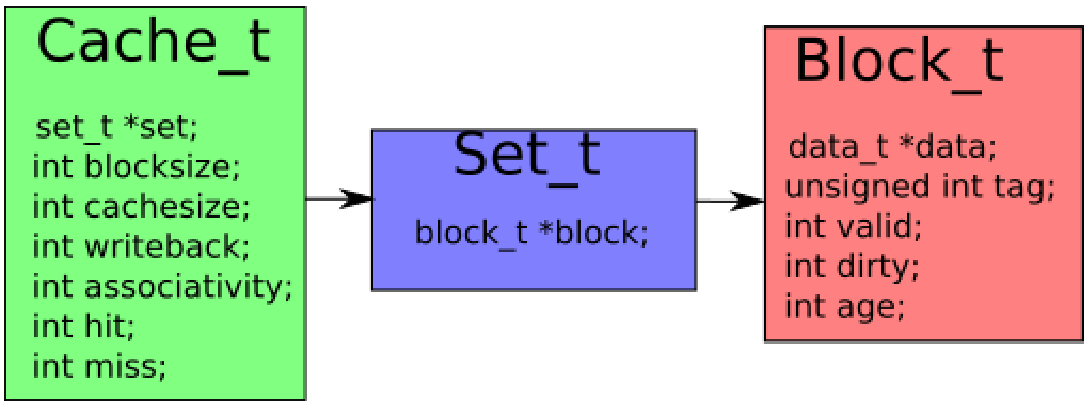

Figure 1: layout of the structs

Input variables has to be provided in order to generate the cache, set and blocks. The input variables needed are cache size, block size and the associativity for the different caches. Using these the cache structure can be set up like this.

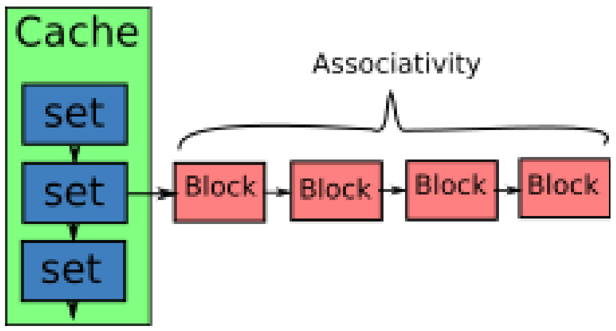

Figure 2: essence of cache layout

Any other variable should be initialized to 0 or NULL, as for what they're used for. The dirty value of a block determines if it's written at a higher level in the cache, meaning L1→L2 or L2→RAM if the dirty value is 1 then it's not written to a higher level and must be, before being overwritten by any new data as to not lose it.. The valid value determines if that is a genuine block in the cache or it is a fluke coincidence, without a valid bit it's treated as garbage to be overwritten. The age is there for keeping check on what block was the least used when one needs to be replaced, each block within each set must be incremented every time an instruction is passed. This is time consuming but will yield the most accurate age estimation of any other implementation. Only when the block is tampered with is the age set to 0 as to say it's not ready to be replaced.

In order to determine where a block should be placed some magic needs to be performed on the addres to determine what set and what block it should be inserted into, this may done by some form of bitmasking. The bitmask settled upon in this implementation utilizes the log2 function of the size of the cache to find the tag/set it should be inserted into, then the block id is determined by the the log2 function of the size of the blocks divided by the associativity for the cache.

3 Experiment methodology

In order to replicate the results in this report one would need a logfile produced by valgrind using the parameters:

```
valgrind --log-file=logfile --tool=lackey --trace-mem=yes [program-name]
```

Then using the python script to convert the logfile into a tracefile, using the generated trace.tr file the cache simulator is ready to be run. After the simulator is run it should be able to display misses, hits and other assorted stats to ones desire using different severity of the misses in L1 cache and L2 cache one can get a graph like this

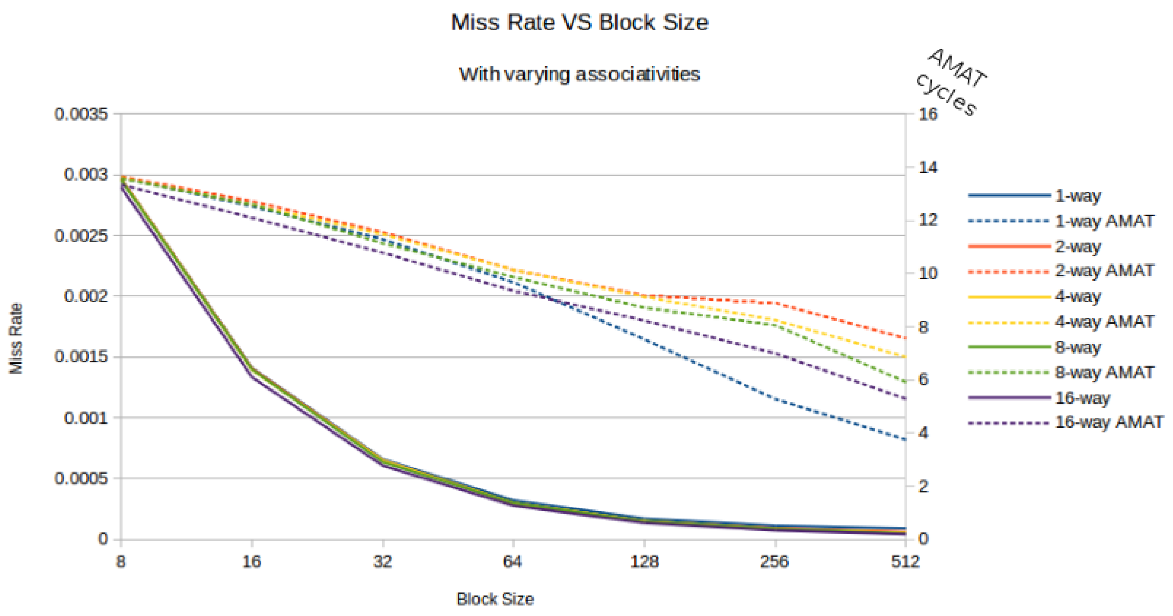

Figure 3: data with severity 10 for L1 misses and 30 for L2 misses, accompanied with AMAT (Average Memory Access Time)

To test the cache simulation for correctness a test trace was manually assembled in order to map out each hit and miss the cache should produce, this was done in this order. First 4 addresses were entered into the cache, since the cache had a cold start it must produce 4 misses. The next two addresses are reads which read to the first two addresses, so they should be in the L1 cache as they missed the first time around. Then comes another address that is supposed to miss the L1 cache and the L2 cache. The last address would have been in L1 cache if it weren’t for the previous miss that overwrote the L1 entry forcing a check of the L2 cache, which should produce a hit as it has more room. The control trace for this simulation was worked out in unison with others, from that control trace this simulation generates 2 hits in the L1 cache, 6 misses, 1 hit in the L2 cache, and 5 misses.

4 Experiment results

After conducting an experiment using a large set from an implementation of the Sieve of Eratosthenes the results of the(see Figure 3, or data.ods) experiment showed that what had most impact on the hit to miss ratio of the Sieve was the block size. Increasing the block size meant fewer misses which in turn brought down the AMAT of the simulation in turn. Due to memory constraints the resulting data could not gather more than block sizes of 512 bytes, but that should be plenty if not excessively expensive to create. The viability of this cache simulator is somewhat questionable, as there are no baseline for measurement against other cache implementations when it comes to AMAT to hit ratio other than lower is better.

With higher block sizes comes the need for bigger cache sizes and that will make the performance plateu at both cost, time and space.

5 Conclusion

Leaving with (what seems to be) a functioning cache simulator with (more or less) genuine hits and misses this report comes to an end. Cache structure now seems more acquaintance than before, meaning that it could be cobbled together quicker than this time. Lessons to take from this exercise equals to, not bask in the afterglow of a job well done too long. That might leave you in a similar crunch time as you left. An appreciation of parsers, which would have made this exercise way harder if that had to be implemented too.