Cache Simulator Report

Alok Hota, Mohammad Ahmadzadeh
https://github.com/auxiliary/CacheSimulator

October 26, 2014

1 Development

Our cache simulator takes a configuration file and a trace file from the command line. It then generates a memory hierarchy based on the configurations and runs the instructions read from the trace file.

1.1 Features

The following features have been implemented in the cache simulator:

- Fully configurable cache hierarchy using YAML
  - Architecture details, level 1 cache and main memory are required
  - Level 2 and level 3 are optional
- Supports write through and write back
- Optionally draws cache layout after simulation
- Writes simulation results to a log file and standard output

1.2 Assumptions

- When a new word is written to level 1 cache, a new block is allocated. If the CPU then reads another word from that block it will be reading an empty word. The simulator will count this as a read hit. This would actually result in a segfault but we consider this out of scope.

1.3 Functionality

The code begins with getting arguments from the command line using the argparse module and reads in cache configurations using the YAML format. The memory hierarchy is then created from the configurations. This is done using a bottom-up approach. We do this so that each level of cache can point to the previous level. The *build_hierarchy* function utilizes the *Cache* class for each level. After setting up logging features the code reads the trace file given by the user and a starts the simulation. Optionally it draws the final contents of each cache after simulation.

The constructor of the *Cache* class takes every configurable parameter needed for the cache and creates an empty cache object. The data in the cache is represented by a Python dictionary structure. The top level keys are set indexes, each of which contains another dictionary keyed by ways. Every way's value is a Block object, which contains its own metadata, such as a *dirty_bit* and *last_accessed* time for LRU.

The *Cache* class contains *read* and *write* methods that are called during simulation. The following figures depict the overall structure of this process. Note that when writing to next level, we are calling the next cache's write method recursively. The same goes for reading from the next level and getting back the response. In addition, the *read* method will call *write* if a block is being replaced.

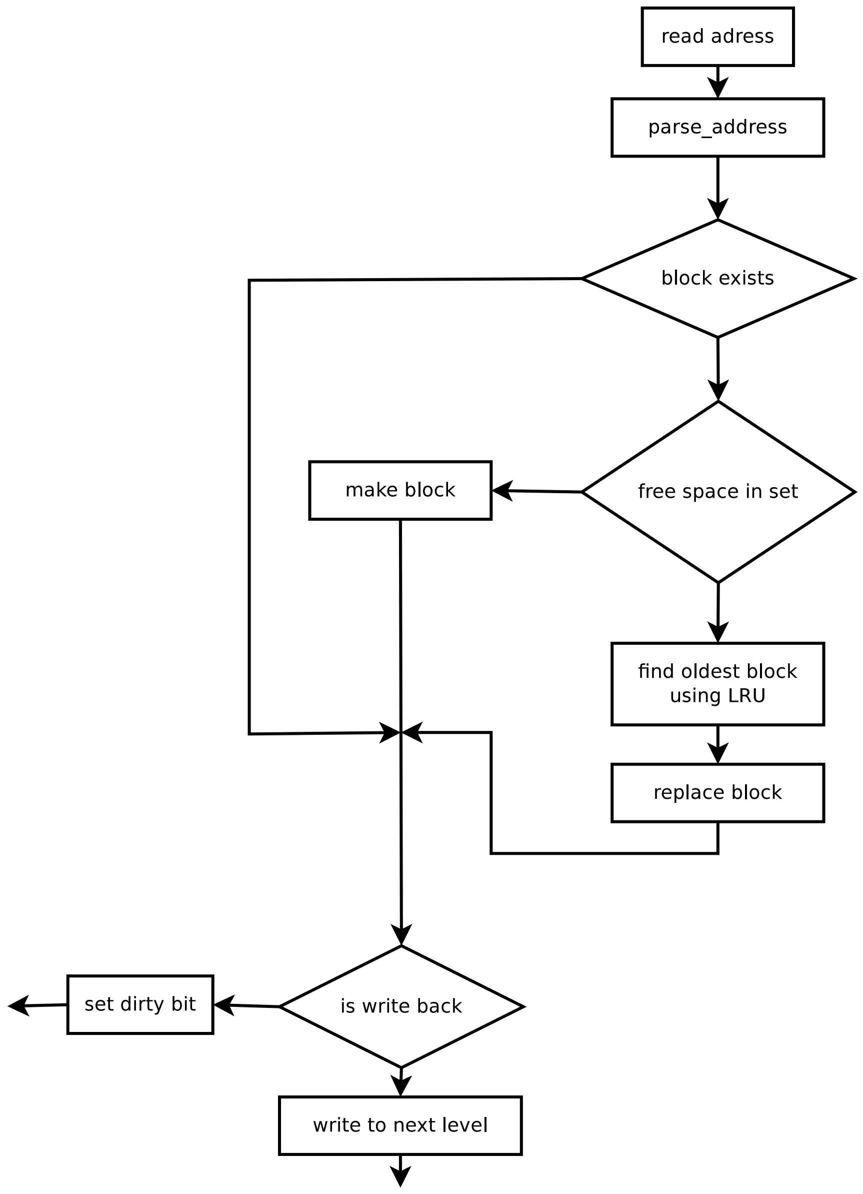

Figure 1: Overview of workflow for writing an address to cache

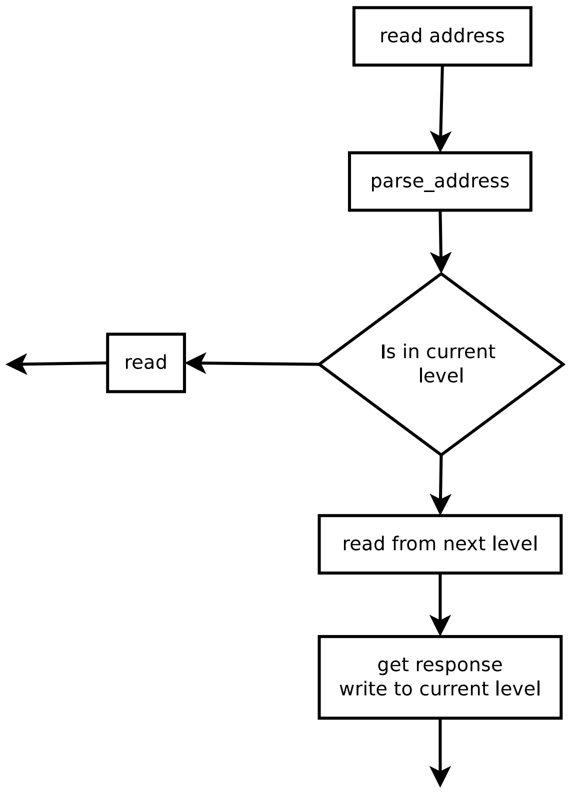

Figure 2: Overview of workflow for reading an address from cache

The simulation and calculation of AMAT rely on the *Response* objects. Response objects are created when an instruction is completed. They keep a record of the total time taken by the instruction and a trace of which levels of cache the instruction hit.

2 Testing

We tested the simulator with several different cache configurations and trace files as well as a stress test. The stress test is generated with a python script located in gen_test/gen_stress_test.py and provides a large randomly generated trace file. These random instructions very rarely result in a hit, which demonstrates the functionality of caches. If stress test is run with our *config_simple_multilevel* configuration, the AMAT for L1 is 1117 cycles as opposed to 1000 for memory alone. This shows that having poor hit rate and write allocate leads to an 11% increase in time taken. There are many other configurations and trace files included with which to test.

3 Conclusion

Writing this simulator greatly inforced the concepts of sets, ways, write back, write through and write allocate.