EE382N – 21: Assignment 1

Professor: Lizy K. John
TA: Jiajun Wang
Department of Electrical and Computer Engineering
University of Texas, Austin

Part 1 Due: 11:59PM September 7, 2016
Part 2 Due: 11:59PM September 21, 2016

1. Introduction and Goals

The goal of this assignment is to study the performance evaluation of multilevel caches in single core processors. Using a trace driven cache simulator, you will conduct a simple performance analysis, and eventually, understand performance implications of different cache organizations.

In this assignment, you are asked to implement a multi-level cache simulator. This assignment will be graded based on correctness. But in a later assignment, you will be judging the runtime efficiency of the single core cache simulator that you develop here. So it is always important to use efficient algorithms and data structures in your code.

1.1 Graded Items

The following items need to be completed by the deadline

A. Single core 1-level cache simulator
B. Single core 2-level cache simulator

2. Single Core 1 Level Cache Simulator (Part A)

2.1 Writing a single core one level cache simulator

In this section of the assignment, you are designing a single core 1 level cache simulator. The cache will be connected to the main memory to handle misses.

Table 1 shows the parameters your simulator must be able to accept. Any value in the range indicated must be handled by the simulator in order to get full credit. The cache will use a **true LRU** replacement policy, and it will be a **write-allocate, write-back cache**. The parameters are listed in Table 1.

Table 1: Single Core 1 Level Cache Parameters

L1 capacity (powers of 2 in KB from 1KB to 64KB)
L1 associativity (1 to 32)
L1 cacheline size (powers of 2 in bytes from 64B to 1KB)
L1 hit latency (read from parameter file eg: 3 cycles)
Main memory latency (read from parameter file eg: 100 cyles)

Table 1 will be read into the simulator from a single configuration file. All cache size and associativity numbers will be powers of two, and your simulator is only responsible for handling those in the range specified.

2.2 Input trace

The input trace will have full 64 bit addressing and will be in the following format:

<cycle>,<R/W>,<address>

<cycle>: cycle number in hexadecimal format
<R/W>: 1 if read, 0 if write
<address>: address in hexadecimal format

You will be given a sample trace file and it is your responsibility to generate your own trace files that test/validate your simulator. The <cycle> info is provided so that you can check whether the previous reference is finished or still outstanding.

2.3 Implementation Details

Cache can only deal with one request from core at a cycle. Input traces will be made in a way that only one request come in at the same cycle. At the beginning of simulation, you may assume all cache lines are invalid initially. A query of a cache for miss/hit can be assumed to happen on the exact same cycle that a request is received. The latencies in Table 1 represent the number of cycles it takes a request to perform tag lookup and access the data in case of a hit or miss.

You invalidate a cacheline when a conflicting access to the same cacheline is made, not after a fill. For the filling of cache lines into the caches on a miss, you can assume that these happen instantaneously after the correct number of latency cycles have elapsed. Metadata for the new cacheline (dirty bits, valid bit) will be updated when the cache has completed that request. LRU bits are updated on access, i.e. the same cycle as cache receives the request.

Assume that the cache is non-blocking (i.e. the cache does not freeze if there is a miss). Future accesses to the cache can happen while the miss is being serviced. The number of misses that can be outstanding is the number of MSHR entries. For the simplicity of this lab, you can assume there are unlimited number of MSHR entries.

The time at which the requests arrive at cache is captured in the input trace. This timing information has to be taken into account for initiating the request. Also, the input trace will be generated in a way that no multiple requests to the same cacheline will occur while the same cache line is currently being processed in the memory hierarchy.

2.4 Output format

At the end of the simulation, your code must be able to output the hit rates, latency, references, and the **Average Memory Access Time (AMAT).** This metric is calculated by the accumulated **sum of all memory request latencies divided by the total number of requests.** Total latency is the accumulated sum of all memory request latencies while

L1 references is the total number of references made to L1. Please note that latencies incurred by writebacks do not have to be considered in AMAT calculation for simplicity. At the end of the simulation, your code must be able to print the hit rates (with 2 decimal places), latency, references (as whole numbers) and the Average Memory Access Time (AMAT) with 2 decimal places (NOTE: you are not computing total execution time.).

```
% cachesim_l1…
L1 hit rate: 0.87
Total latency: 3000
L1 references: 150
AMAT: 20.00
%
```

Template files, sample input traces, and sample configuration file are compressed and are in the a1_A directory in *a1.zip* on the course website. The template file only has a skeleton where you have to implement the single core 1 level cache simulator. A sample configuration file is also provided with the lab template. You are free to change any value, but do not change the parameter name. A sample makefile is provided, but you are encouraged to modify it to suit your need. However, you are not allowed to change the output executable name, *cachesim_l1*. The command line argument is the following:

```
% <your cachesim directory>/a1_A/cachesim_l1 <config file> <trace-file >
```

Your simulator must be able to handle any parameter values listed in Table 1.

3. Single Core 2 Level Cache Simulator (Part B)

3.1 Writing a single core 2 level cache simulator

In this section of the assignment, you are designing a single core 2 level cache simulator. Table 2 shows the additional parameters your simulator must be able to accept.

Table 2: Single Core 2 Level Cache Parameters

L1 capacity (powers of 2 in KB from 1KB to 64KB)
L1 associativity (1 to 32)
L1 cacheline size (powers of 2 in bytes from 64B to 1KB)
L1 hit latency (read from parameter file eg: 3)
L2 hit latency (read from parameter file eg: 2-20)
Main memory latency (read from parameter file eg: 50-500)
L2 capacity (powers of 2 in MB from 1MB to 16MB)
L2 associativity (1 to 32)
L2 cacheline size (same as L1 cacheline size)

3.2 Implementation Details

Similar to L1, LRU update in L2 happens at the same cycle as L2 receives a request. If a cacheline is evicted, the old data becomes invalid immediately. The number of cycles it takes to complete a hit/miss request in the following three scenarios are:

L1 Hit	L1 hit latency (includes tag check and data access)
L1 Miss, but L2 Hit	L1 hit latency + L2 hit latency
L1 Miss, and L2 Miss	L1 hit latency + L2 hit latency + Memory latency

Please load the cache line into both the L1 and L2 cache on an L2 miss. Also, please note that on a L1 cache miss, you are not allowed to check the L2 cache on the same cycle as the L1 cache miss, but you must wait L1 hit latency cycles before the L1 cache sends the request to L2.

If you are evicting a clean cacheline from L1 and it's not in L2, there's no need to update L2. If you are evicting a dirty cacheline from L1, L2 should be updated. Please note that you must wait L1 hit latency cycles before the L2 cache receives the writeback request from the L1 cache. In case of a write back to L2 where the data is not resident, kick out the LRU cacheline that is already present in L2 (which happens immediately along with LRU update), write data back to L2.

L2 may receive cache access request from L1, write back request from L1, and data reply from memory at the same cycle. You can assume there are event queues where multiple requests arrive. The LRU update policy is that the request gets higher priority than write back. In this assignment, just assume L2 is able to start servicing those three different types of request in the same cycle. Please note that, the input trace will be generated in a way that no multiple requests to the same cacheline will occur while the same cache line is currently being processed in the memory hierarchy.

3.3 Input trace
The same input trace as Section 2 will be used.

3.4 Output format
At the end of the simulation, your code must be able to output the hit rates, latency, references, and the **Average Memory Access Time (AMAT).** This metric is calculated by the accumulated **sum of all memory request latencies divided by the total number of requests.** Total latency is the accumulated sum of all memory request latencies. Please note that latencies incurred by writebacks do not have to be considered in AMAT calculation for simplicity. L1 references is the total number of references made to L1 and L2 references is the total number of references made to L2 (not including L1 writebacks to L2). At the end of the simulation, your code must be able to print the hit rates (with 2 decimal places), latency, references (as whole numbers) and the Average Memory Access Time (AMAT) with 2 decimal places (NOTE: you are not computing total execution time).

```
% cachesim…
L1 hit rate: 0.87
L2 hit rate: 0.87
Total latency: 4000
L1 references: 300
L2 references: 40
```

```
AMAT: 23.25
%
```

Template files, sample input traces, and sample configuration file are compressed and are in the a1_B directory in *a1.zip* on the course website. The command line argument is the following:

```
% <your cachesim directory>/a1_B/cachesim_l2 <config file> <trace-file >
```

4. Assignment Guidelines

4.1 Coding guidelines

You can write your assignment in C/C++. The grading will be done with g++ on ECE LRC machines. If your code does not compile on these machines, you will lose 20% of the total grade automatically even if it is completely functional.

5. Submission Instructions and Grading Guidelines

5.1 Submission instructions

Your assignment must be compressed in tar.gz format. Please use the following command to tar your submission:

```
% tar –zcvf <your eid>-a1.tar.gz <your working directory>
```

There will be a submission link available on the course website, so please submit your tarball by deadline. Your submission directory must have the same directory structure as the template directory structure. Please remove all your temporary and input trace files. You are only required to submit your code and makefile. Make sure that the name of your working directory is named *<your eid>-a1*. Notice: your submission of partB should include both partA and partB codes (in a1_A and a1_B directory).

5.3 Grading guidelines

Your code will be compiled on LRC machines. LRC machine access information can be found on the ECE IT page (http://www.ece.utexas.edu/it/remote-linux). The grade will be based on correctness. Partial points will be given if appropriate comments are written in the code and late submissions will lose 20% of the total grade each 24 hours after the deadline.

5.4 Assignment questions

If you have any general questions, please post it under *a1* label on Piazza. The course teaching staff will not answer general questions sent to personal email addresses.