# Memory Simulation Project

Due Date: 5:00pm, Tuesday, December 13, 2011

The goal of this project is to implement a simulator that can simulate and evaluate a memory system with two levels of cache memory and a main memory. In this system, there is a level-1 (L1) instruction cache and an L1 data cache. The level-2 (L2) cache is a unified cache which handles requests from both L1 caches. Misses in the L2 cache are handled by the main memory.

To implement this simulator you may use any programming language you would like, however be aware that the input data is stored in *gzip* format. To be read, this data must be unzipped. A fairly efficient way to do this is to write your simulation program so it reads data from standard input. The Unix utility **zcat** can then be used to pipe the unzipped data into your simulator. So, to run a simulation you would type:

```
zcat <tracefile_name> | <simulator_name> <config_file>
```

The contents of the <config_file> are described below, as is the format of the trace file.

## 1 Memory Hierarchy Design

The memory system simulator is described by a set of parameters that specify the organization of the caches and the main memory. The simulator code should be written in terms of these parameters. At runtime, the parameters should be read from a file specified on the command line, or the default values should be used.

You may assume that the cache sizes, bus widths and associativity will always be powers of two. You can take advantage of this fact when computing masks and shift amounts. The size values are in bytes.

### 1.1 Cache Parameters

**L1 and L2 Cache Parameters**

| Parameter | L1 | | L2 | |
|---|---|---|---|---|
| | Name | Default | Name | Default |
| Block size | L1_block_size | 32 | L2_block_size | 64 |
| Cache size | L1_cache_size | 8192 | L2_cache_size | 65536 |
| Associativity | L1_assoc | 1 | L2_assoc | 1 |
| Hit Time | L1_hit_time | 1 | L2_hit_time | 4 |
| Miss Time | L1_miss_time | 1 | L2_miss_time | 6 |
| Transfer L1 to/from L2 | | | L2_transfer_time | 6 |
| Bus width L1 to L2 | | | L2_bus_width | 16 |

The **hit time** is the time to return an item that is a hit in the cache. The **miss time** is the time to determine that an item has missed and the time to make the request to the next cache level. With the given default parameters, the time to transfer a 32-byte value between L1 and L2 is $6 \times (32/16) = 12$ cycles.

## 1.2 Main Memory Parameters

The main memory will be modeled by a constant delay and bandwidth. Since the processor model is in-order, there can be no more than a single out-standing request to the memory system. Thus, there can be no contention for the bus or the interface logic.

| Parameter | Name | Default |
|---|---|---|
| Time to send the address to memory | mem_sendaddr | 10 |
| Time for the memory to be ready for start of transfer | mem_ready | 50 |
| Time to send/receive a single bus-width of data | mem_chunktime | 20 |
| Width of the bus interface to memory, in bytes | mem_chunksize | 16 |

Using these values, a single memory transaction for a 64 byte cache line is $10+50+(20\times64/16) = 140$ cycles. These latencies are for both reading and writing.

## 2 Organization

The caches are *write-allocate, write-back* caches. This means that all writes will immediately write to the L1 cache. There is no write buffer. If writing to a non-resident line, the system must first fill the line. Thus, the logic/code for writing is very similar to the logic/code for reading; in fact, the primary difference is one of accounting and marking cache lines as "dirty" if the program has written to a particular cache entry.

When an L1 block request is a miss, it must be fetched from the L2 cache.

When an L2 block request is a miss, it must be fetched from main memory.

When a dirty L1 block is evicted, it must be written back to the L2 cache.

When a dirty L2 block is evicted, it must be written back to the main memory.

A write request causes a line to be marked dirty only at the level that sees a write-type request. If a write request causes a miss in the L1 cache, when that line is brought in from the L2 cache, it should **not** be marked as dirty in the L2 cache. The dirty line in the L1 cache will eventually be written back to the L2 cache, as a write request, when the line is replaced by some other L1 request. During the write back to the L2 cache, the line is marked dirty in the L2 cache.

Each cache must maintain a true LRU (Least Recently Used) replacement policy for each set in a set-associative cache or in a fully associative cache.

## 3 Program Execution Trace

The input data for the simulator is an execution trace file in *gzip* format. The trace file is a sequence of records that contains information about each instruction that was executed. There is one line per record and one record per executed instruction. The format of a record is given below:

<Instruction type> <PC (in Hex)> <Execution information>

The *Instruction type* field contains an L, S, B or C for Load, Store, Branch or Computation. The *Execution information* field depends on the instruction type field and is interpreted as follows:

**Load/Store** The execution information is the memory address accessed (in hex).

**Branch** The information "0" for not taken or "1" for taken.

**Computation** The execution information is the latency or execution time of the instruction (a number 1-10).

We will assume that a taken or not taken branch takes the same amount of time to execute. If a branch instruction hits in the L1 instruction cache, its execution time is the L1_hit_time **plus** 1 cycle to execute the instruction. If it misses, the execution time is the total time amount of time it takes to service the miss plus 1 cycle.

For Load/Store instructions, the execution time is the L1 instruction cache access time (L1_hit_time if it's a hit, or whatever time it takes to service a miss) **plus** the L1 data cache access time. The L1 data cache access time will be L1_hit_time if it is a hit, or the amount of time it takes to service a miss.

The execution time for computational instructions is the L1 instruction cache access time (computed the same way as for branches and Load/Store instructions) plus the execution time from the trace.

With these assumptions, if the L1 hit time is 1 cycle and all instructions and data references hit in the cache (that is, a perfect memory system) and all 'C' type instructions take 1 cycle, the execution time is 2 times the number of instructions in the trace.

**Note:** After a cache miss has been serviced and the miss penalty has been added to the execution time, you must account for the time to insert the new block into the cache. This is done by adding the hit time of the cache level to the total execution time. This is essentially a 'replay' of the cache reference.

To read the trace data in C or C++, one could use the following code:

```
    while (scanf("%c %x %x\n",&op,&address,&exec_info) == 3) {
        Body of processing code..
        };
```

where the variables are defined as:

```
    char op;
    unsigned int address, exec_info;
```

## 4 Inputs

Several sets of input traces are provided. The first set of 6 traces, named I1 through I5 and I10, are tiny traces that you should use to test and debug your simulator. These traces are not in gzip format, since they are small, and you may want to look at them as you debug your code. They contain only between 9 and 1000 references.

The second set of 6 traces are the *production* traces. These traces are taken from the SPEC benchmarks and each contains between 181k and 2.9G references. The zipped trace files are between roughly 167MB and 3GB in size. All 6 traces total about 9.4 GB.

There are also a couple directories that contain the first 1 million and 5 million references from each of the production traces. These files are not too large and can be downloaded to help testing your code.

All these traces can be found on the eces-shell machine for which all students have accounts. There are some RedHat Linux machines in EE-287, from which students can log into the eces-shell machine. This room is at the back (northeast corner) of the Circuits Lab, EE-281. The traces are in the directory /scratch/arp/ecen4593. There is a "readme" file in the directory that describes the contents of the sub-directories. Of course, you can also remotely login into the machine using a secure shell.

Since the production traces are quite large, you should **not** copy these traces to your home directory on eces (in fact, you do not have a large enough disk quota to copy them all to your home directory on eces). If you choose to work from home, or elsewhere, you are welcome to copy these traces to the machine you are working on. Please do not perform such a large file transfer in the middle of the day, wait until the evening or night.

## 5 Output and Interface

Write your simulator program to get the trace file information from standard input. The program should also accept an optional configuration file from the command line. This configuration file should be read to set the cache and memory configuration parameters. If no configuration file is given, the default parameter values should be used. Use whatever format you find convenient for this parameter file.

The program should output appropriate statistics concerning hits, misses, miss rate, etc. When deciding on the format of this output file, remember you may want to do some post-processing of the output data to get it into a form that can be plotted or graphed.

A sample of the type of statistics you should produce is shown at the end of this document. Notice that in addition to simulation statistics, the memory system configuration parameters are also given in the output. Although not shown, I also printed valid, dirty and tag information (per cache block) for each cache because it can be useful for debugging.

When running the production traces, you will find that for some of the traces, the simulated execution time is over 4 billion cycles. You should define any timing statistic as an "unsigned long long" type of variable. When printing such values, use **"%Lu"** for the formatting conversion specifier. This conversion specifier indicates you want to print an unsigned long long (64-bit) integer.

## 6 Evaluation

### 6.1 Memory System Cost

To make the evaluation of your memory system more interesting and to provide a metric for comparison, we can assign a cost function to a memory system configuration. For this, assume the following:

- The L1 cache memory costs $100 for each 4KB, and an additional $100 for each doubling in associativity beyond direct-mapped (in other words, an 4KB direct mapped cache costs $100, 4KB 2-way is $200, 4KB 4-way is $300 and 8KB 1-way is $200). The cost for doubling the associativity is per 4KB, so the combined costs are multiplied. An 8KB 2-way L1 cache is $400.
- The L2 cache memory costs $50 per 64KB of direct mapped cache, and an additional $50 for each doubling in associativity.
- It costs $200 to decrease the main memory latency (mem_ready) by a factor of 2.
- It costs $100 to increase the bandwidth (mem_chunksize) by a factor of 2.
- In the base main memory system, the mem_ready latency of 100 costs $50 and the base 16-byte mem_chunksize bandwidth costs $25, for a total of $75.

Your program should compute the cost function based on the memory system configuration specified.

### 6.2 Memory System Configurations

Simulate the memory system configurations listed below. Unless otherwise noted, use the default parameters.

**Base:** 8KB Direct-mapped Icache, 8KB Direct-mapped Dcache, with a unified 64KB direct-mapped Level-2 cache.

**L1-2way:** 8KB two-way associative Icache, 8KB two-way associative Dcache, with a unified 64KB direct-mapped Level-2 cache.

**L2-2way:** 8KB Direct-mapped Icache, 8KB Direct-mapped Dcache, with a unified 64KB two-way associative Level-2 cache.

**All-2way:** 8KB two-way associative Icache, 8KB two-way associative Dcache, with a unified 64KB two-way associative Level-2 cache.

**2-4-way:** 8KB two-way associative Icache, 8KB two-way associative Dcache, with a unified 64KB four-way associative Level-2 cache.

**L2-Big:** 8KB two-way associative Icache, 8KB two-way associative Dcache, with a unified 128KB Direct-mapped Level-2 cache.

**All-FA:** Fully Associative, fully associative 8KB Icache, 8KB Dcache, and 64KB Level-2 cache.

### 6.3 What to turn in

Turn in printouts of your simulator code along with the output of simulations of the production traces with each of the seven memory system configurations; Base, L1-2way, L2-2way, All-2way, 2-4way, L2-Big, and All-FA. You should also plot graphs of the performance for these simulation results.

Using the gcc trace, you should then run simulations in which the bandwidth to main memory (mem_chunksize) is increased by powers of 2; that is, from 16 to 32 to 64. For these simulations, submit plots of simulation performance results versus cost. Write one paragraph describing the best memory system model (comparing cost and performance) and discuss the effects of the memory system bandwidth on overall system performance.

Just to be clear, you must submit a hardcopy of the material above. This is a project report, make it clear and organized. In addition to the hardcopy, please submit a CD (or DVD) with your simulator code and simulation results.

## 7 A few final words

You may work in two-person teams on this project. If you work alone, you must still run the simulations described above and turn in the same documentation.

There are several ways to approach writing this code. One way not to write the simulation code is try to a write a fully functional simulator from the very beginning. It is better to write the simulator incrementally and test its functionality before adding another feature. So, one could start by writing a simulation of a single cache, that assumes a fixed delay on a miss, get that working and then modify the code to add more levels in the memory hierarchy.

Whatever strategy you use, the one feature you must have coded from the very beginning is the ability to configure the cache size, block size and associative at run time. If you do not code these features to be parameters early-on, it will be difficult to change them later.

You might find it useful to sketch a flowchart or state transition diagram for the steps in handling a memory request. You could code this directly, but you may find yourself re-writing code because you forgot or miss-handled a case.

Even though several test traces are provided, you will probably want to create your own trace to check the functionality of your simulation.

If you find yourself getting frustrated, please come to office hours and ask questions. For example, when running multiple simulations, it is useful write a shell script. If you do not know how to write a shell script, come see me.

For debugging it can useful to put print statements in your code. Using *ifdef*'s in your code is a convenient way to include, or not include these print statements when compiling your code. Depending on how you structure your code, it could also be useful to have a simple *makefile* for compiling your code. Again, if you want to use these, but don't know how, please come and see me.

And finally, as you have heard before, do not wait until the last minute. You are not finished when you have a working simulator. You still must run the production traces and analyze the results. This will take longer than you may think.

```
---------------------------------------------------------------
gcc.L2-2way         Simulation Results
---------------------------------------------------------------

Memory system:
  Dcache size = 8192 : ways = 1 : block size = 32
  Icache size = 8192 : ways = 1 : block size = 32
  L2-cache size = 65536 : ways = 2 : block size = 64
  Memory ready time = 50 : chunksize = 16 : chunktime = 20

Execute time = 10727121042 : Total refs = 2565131003
Inst refs = 1892121367 : Data refs = 673009636

Number of instructions:  [Percentage]
  Loads  (L) = 440382918    [23.3%] : Stores (S) = 232626718    [12.3%]
  Branch (B) = 333299155    [17.6%] : Comp.  (C) = 885812576    [46.8%]
  Total  (T) = 1892121367

Cycles for instructions:  [Percentage]
  Loads  (L) = 4663995140    [43.5%] : Stores (S) = 1610094668    [15.0%]
  Branch (B) = 1102745620    [10.3%] : Comp.  (C) = 3350285614    [31.2%]
  Total  (T) = 10727121042

Cycles per Instruction (CPI):
  Loads     (L) = 10.6 : Stores    (S) =  6.9
  Branch    (B) =  3.3 : Comp.     (C) =  3.8
  Overall (CPI) :  5.7

Cycles for processor w/perfect memory system 3784242734
Cycles for processor w/simulated memory system 10727121042
Ratio of simulated/perfect performance = 2.834681

Memory Level:  L1i
  Hit count = 1804702086 Miss Count = 87419281 Total Request = 1892121367
  Hit Rate = 95.4% Miss Rate =  4.6%
  Kickouts : 87419025 Dirty kickouts : 0 Transfers : 87419281

Memory Level:  L1d
  Hit count = 609775533 Miss Count = 63234103 Total Request = 673009636
  Hit Rate = 90.6% Miss Rate =  9.4%
  Kickouts : 63233847 Dirty kickouts : 22825701 Transfers : 63234103

Memory Level:  L2
  Hit count = 149954461 Miss Count = 23524624 Total Request = 173479085
  Hit Rate = 86.4% Miss Rate = 13.6%
  Kickouts : 23523600 Dirty kickouts : 4156889 Transfers : 23524624

L1 cache cost (Icache $200) + (Dcache $200) = $400
L2 cache cost = $100
Memory cost = $275
Total cost = $775
```