Memory Simulation Project

Due Date: 5:00pm, Friday, April 29, 2016
This is a **hard** deadline.
No projects will be accepted after the due date and time.

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. An 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 and bus width values are in bytes.

1.1 Cache Parameters

Default 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	32768
Associativity	L1_assoc	1	L2_assoc	1
Hit Time	L1_hit_time	1	L2_hit_time	8
Miss Time	L1_miss_time	1	L2_miss_time	10
Transfer L1 to/from L2			L2_transfer_time	10
Bus width L1 to L2			L2_bus_width	16

The **hit time** is the time to return an item that is a hit in a 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 $10 \times (32/16) = 20$ cycles.

1.2 Main Memory Parameters

The main memory is 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 bus 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	15
Width of the bus interface to memory, in bytes	mem_chunksize	8

Using these values, a single memory transaction for a 64 byte cache line is $10 + 50 + (15 \times 64/8) = 180$ cycles. These latencies are for both reading and writing.

2 Cache Organization

The caches are *write-allocate, write-back* caches with an 8-entry **victim cache**. If the line associated with a request is found in a cache, a cache hit occurs. 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. The logic/code for a write request is very similar to the logic/code for a read request; 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.

If a request results in a miss, the cache's victim cache is checked first. If the requested line is found in the victim cache, that line is copied from the victim cache and the line being evicted from the cache is copied to the victim cache. The victim cache is maintained like an 8-deep LRU stack. The victim cache is checked on a miss and, if we have a hit at an entry, that entry is brought to the top of the stack, where the contents are swapped with the line being evicted.

If the request which caused the miss is not found in the victim cache, then that line must be brought in from the next level in the hierarchy. However, before that is done, room must be made in the victim cache for the line that is being evicted from the cache. So, the entry at the bottom of the victim cache is checked to see if it is dirty. If it is dirty, it is written back to the next level of the hierarchy (this the write-back part of the cache). That entry is then brought to the top of the victim cache and over-written with the line being evicted from the cache. It doesn't matter if the request that caused the miss is a read or a write, the system must first get the line from the next level in the hierarchy (this the write-allocate part of the cache).

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, and that line has to be 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 (a dirty kickout from the victim cache). During the write back to the L2 cache, the line is marked dirty in the L2 cache.

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.

3 Program Trace Format

The input data for the simulator is an execution trace file compressed with *gzip*. The trace file is a sequence of records that contains information about each reference that was made. The format of a record is given below:

<Reference type> <Address (in Hex)> <Number of bytes>

– The *Reference type* field contains an I, R, or W for Instruction, Data Read, or Data Write. This field is in column one of the line.
– The *Address* field is a memory address in hexadecimal and can be 64 bits long, 16 hex digits. This is the address associated with the instruction or data reference.
– The *Number of bytes* is the number of bytes that were referenced by this request.

The program traces being used come from an Intel architecture, so instruction and data references do not have to be aligned. That is, they can fall on any byte boundary. Furthermore, since the instruction set uses a variable length encoding and can reference many data types, the *Number of bytes* field is needed. Recall, that if we were using a MIPS architecture all instructions would fall on word boundaries and all instruction are 32-bits. It would also have been safe to assume all data references were to words.

Since this is not the case, an instruction or data reference may generate multiple references to the first-level caches. For your simulator assume that there is a 4-byte bus between the processor and the L1 caches. All references to the L1 caches look like they are aligned on 4-byte boundaries and the L1 cache would return a 4-byte value. An unaligned 4-byte reference would look like 2 4-byte references to an L1 cache. An 8-byte unaligned reference may require 3 references to an L1 cache.

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

```
while (scanf(”%c %Lx %d\n”,&op,&address,&bytesize) == 3) {
    Body of processing code..
     };
```

where the variables are defined as:

```
char op;
unsigned long long int address;
unsigned int bytesize;
```

When you write your simulator, you will have to be very aware of when to use a *long int* versus a *long long int*, and when to make the int *signed* versus *unsigned*.

4 Program Traces

Several sets of input traces are provided. The first set of 6 traces, named tr1 through tr6, are tiny traces that you should initially use when testing and debugging 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 10 and 1000 references.

There is a second set of 6 traces that are the *production* traces. These traces are taken from the SPEC benchmarks and contain between roughly 10 billion and 17 billion references. These zipped trace files are between roughly 4.4 GB and about 19 GB in size. The 6 traces total about 62 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.

Five of the production traces are sampled from traces of the complete execution of the benchmarks. The samples were 100,000 references spaced equally throughout the total traces. The original traces contain between 10's and 100's of billions of references. The libquantum.gz trace is the complete trace of its execution on the training input.

All these traces can be found on the **eces-shell** machine for which all students have accounts. The traces are in the directory /scratch/arp/ecen4593-sp16. There is a "readme" file in the directory that describes the contents of the sub-directories. You can login remotely onto the machine using a secure shell (ssh).

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 the middle of the 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 had an output routine that printed valid, dirty and tag information (per cache block) for each cache because this information can be useful for debugging.

When running the production traces, you will find that 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 **"%llu"** or **"%Lu"** for the formatting conversion specifier. This conversion specifier indicates you want to print an unsigned long long (64-bit) integer.

6 Memory System Evaluations

Your simulator will simulate program execution time for a particular cache configuration. To help make an evaluation of various cache configurations, it is useful to make some assumptions about the execution time for an ideal processor. For an ideal processor-cache system you should assume that an instruction takes 1 cycle to execute. This is in addition to the time is takes to fetch the instruction. So the best-case execution time would be 1 cycle for each instruction plus 1 cycle to fetch the instruction and 1 cycle for each data reference.

However, since the memory references are not aligned and may involve multiple cache accesses, it is also worthwhile to compute an execution time where mis-aligned and multiple memory accesses are made. An example of these statistics are shown in the sample output at the end of this document.

For your simulations, the execution time is one cycle of for each instruction, plus the simulated time to access the memory hierarachy for an instruction fetch and any data references.

6.1 Memory System Cost

To make the evaluation of your memory system more interesting and to provide another 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 16KB 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 50 costs $50 and the base 8-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

For your project report, you must simulate the memory system configurations listed below. Unless otherwise noted, use the default parameters.

Default: 8KB direct-mapped Icache, 8KB direct-mapped Dcache, with a unified 32KB direct-mapped Level-2 cache.

L1-2way: 8KB two-way set associative Icache, 8KB two-way set associative Dcache, with a unified 32KB direct-mapped Level-2 cache.

All-2way: 8KB two-way set associative Icache, 8KB two-way set associative Dcache, with a unified 32KB two-way set associative Level-2 cache.

All-4way: 8KB four-way set associative Icache, 8KB four-way set associative Dcache, with a unified 32KB four-way set associative Level-2 cache.

L1-8way: 8KB eight-way set associative Icache, 8KB eight-way set associative Dcache, with a unified 32KB direct-mapped Level-2 cache.

L1-small: 4KB direct-mapped Icache, 4KB direct-mapped Dcache, with a unified 32KB direct-mapped Level-2 cache.

L1-small-4way: 4KB 4-way set associative Icache, 4KB 4-way set associative Dcache, with a unified 32KB direct-mapped Level-2 cache.

All-small: 4KB direct-mapped Icache, 4KB direct-mapped Dcache, with a unified 16KB direct-mapped Level-2 cache.

All-FA: Fully Associative 8KB Icache, 8KB Dcache, and 32KB 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 nine memory system configurations: Default, L1-2way, All-2way, All-4way, L1-8way, L1-small, L1-small-4way, All-small and All-FA.

Your simulation printouts must contain, at least, the information shown on the sample printout page at the end of this document. (I want to see the actual counts associated with hits, misses, etc., not just a percentage.) You should plot graphs of the performance for these simulation results. You need to determine how and what is the best way to plot data from your simulations. Also, write several pages commenting on these simulation results. You are welcome to run more simulations, for other cache configurations, to help confirm or explain trends in the data.

Using only the sjeng trace and the default configuration, you should also run simulations in which the bandwidth to main memory (mem_chunksize) is increased by powers of 2; that is, from 8 to 16 to 32 to 64. For these simulations, submit plots of simulated performance results versus cost. Write at least one paragraph describing the best main memory system model (comparing cost and performance) and discuss the effects of the main memory system bandwidth on overall system performance.

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

7 A few final words

You must work in two-person teams (no single or three person teams) on this project.

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.

Whatever strategy you use, the one feature you should 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 write code directly, without a flowchart, but you may find yourself re-writing code because you forgot or miss-handled a case.

Even though several test traces are provided, you may 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. Another approach is to use "-D" variables on the gcc (or g++) command line when compiling to conditionally compile code sequences. 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 techniques, 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. (Running all the simulations can take as much as 36 hours of CPU time.)

```
-------------------------------------------------------------------------
      sjeng.L1-2way       Simulation Results
-------------------------------------------------------------------------

  Memory system:
    Dcache size = 8192 : ways = 2 : block size = 32
    Icache size = 8192 : ways = 2 : block size = 32
    L2-cache size = 32768 : ways = 1 : block size = 64
    Memory ready time = 50 : chunksize = 8 : chunktime = 15

  Execute time =  30608145911;  Total refs = 5000000000
  Inst refs = 3685034778;  Data refs = 1314965222

  Number of reference types:  [Percentage]
    Reads  =   940933865    [18.8%]
    Writes =   374031357    [ 7.5%]
    Inst.  =  3685034778    [73.7%]
    Total  =  5000000000

  Total cycles for activities:  [Percentage]
    Reads  =  6362234282    [20.8%]
    Writes =  5668329310    [18.5%]
    Inst.  = 18577582319    [60.7%]
    Total  = 30608145911

  CPI =  8.3
  Ideal: Exec. Time = 8685034778; CPI =  2.4
  Ideal mis-aligned: Exec. Time = 11645500025; CPI =  3.2

  Memory Level:  L1i
    Hit Count = 6162531865  Miss Count = 108939899
    Total Requests = 6271471764
    Hit Rate = 98.3%   Miss Rate =  1.7%
    Kickouts = 103082777; Dirty kickouts = 0; Transfers = 103083041
    VC Hit count = 5856858

  Memory Level:  L1d
    Hit Count = 1622984580  Miss Count = 66008903
    Total Requests = 1688993483
    Hit Rate = 96.1%   Miss Rate =  3.9%
    Kickouts = 57344834; Dirty kickouts = 30336644; Transfers = 57345098
    VC Hit count = 8663805

  Memory Level:  L2
    Hit Count = 132055819  Miss Count = 58708964
    Total Requests = 190764783
    Hit Rate = 69.2%   Miss Rate = 30.8%
    Kickouts = 56097023; Dirty kickouts = 15342421; Transfers = 56097543
    VC Hit count = 2611421

  L1 cache cost (Icache $400) + (Dcache $400) = $800
  L2 cache cost = $100;  Memory cost = $75  Total cost = $975
```