NC STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ECE 463/521: SPRING 2016: PROJECT 1: CACHE HIERARCHY DESIGN (V7)

Due date: Sunday, Oct 2nd, 11:59 PM

1. Ground rules

- This is an *individual* project.
- Collaboration by sharing code is strictly prohibited and is considered cheating.
- A TA *will* scan code from current *and past* semesters through automated tools available to us that can detect cheating; code that is flagged by these tools will be dealt with severely and will receive appropriate action in accordance with university policy.
- Use the tag Project1 in the Discussion Forum for questions (or write to TA/instructor).
- Don't submit any *code* in the Discussion Forum (check with TA or instructor if in doubt)
- You *must* use C/C++ programming languages. Exceptions (JAVA/Python) must be approved by TA or instructor.
- Compatibility with EOS Linux environment is *required*.

2. Project Summary

In this project, you will implement a cache hierarchy simulator. Your cache simulator must estimate the cache miss rates for different cache configurations when running different workloads. Different applications have different memory behavior. Accordingly, you will be provided with memory access traces derived from varied applications. The rest of this document describes precisely what you will implement (Specification), how you will test your simulator (Validation runs), how will you communicate your findings (Report) and what all you will submit (Report, simulator source code etc.).

3. Specification

Model a cache with parameterized geometry, replacement policy and inclusion policy. For simplicity, assume that the cache access is atomic; no new cache accesses will occur until the current access is served. Also assume that write policy in effect is the Write-Back Write Allocate (WBWA) policy. *ECE521 students are required to implement multi-level cache hierarchy. ECE463 are required to implement only one level cache (only L1 Cache).*

3.1.1. Replacement Policy

Model a generic cache hierarchy with the following replacement policies supported. For all of these policies assume that when invalid lines are present they are selected as replacement candidates (for consistency in results just stick to picking the left most invalid line, i.e. the lowest numbered way with the invalid line). If invalid lines are not present use the replacement policy in effect.

- LRU: In the LRU replacement policy the least recently used block is evicted in order to make room for an access that misses the cache. A block that is accessed on a hit or installed on a miss becomes the most recently used. The LRU ranking of all the blocks is adjusted.
- FIFO: In the FIFO replacement policy the block which is the oldest to have been inserted into the set from amongst all the available blocks is the one that is evicted.
- pseudoLRU: LRU is expensive to implement in terms of storage overhead, so approximated LRU policies such as Pseudo-LRU are more commonly used. Pseudo-LRU implements something like a

binary tree to keep track of the of the temporal locality of blocks. When an access matches a cache line, we set all the bits leading to that cache line to the other direction. If we assume 0 means left and 1 means right, then an access to line_0 will set bit_1 to 1 and bit_0 to 1. However, if the access was to line_1, bit_1 will be set 0 and bit_0 will be set to 1. In both cases, bit_2 will remain unchanged. When trying to find a block to be evicted, we simply follow the directions indicated by the bits. For example, if bit_0 value is 1, then we go to the right sub-tree and check bit_2 value, if it is 1, then we evict line_3, otherwise we evict line_2. In your implementation, assume that all bits are initially set (has value of 1), and the order of the tree leafs start from line_0 for the left most leaf and line (N-1) for the right most leaf. Note that Pseudo-LRU works only with power of 2 associativity value. Make your implementation generic for any power of 2 associativity value.

- Optimal: Replace the block that will be needed farthest in the future. Note that this is the most difficult replacement policy and it is impossible to implement in a real system. This will need preprocessing the trace to determine reuse distance for each memory reference (i.e. how many accesses later we will need this cache block). You can then run the actual cache simulation on the output of the preprocessing stage. *This part is optional to both ECE463 and ECE521 students. Correctly implementing this replacement policy can get 10 extra points out of 100. See Grading Section for details.*

3.1.2. Inclusion Property

Model caches to support various inclusion properties. *ECE463 students have to only implement non-inclusive caches. ECE521 students should additionally implement exclusive and inclusive caches.*

- Inclusive L2 caches: These caches maintain the invariant that any block that exists in the L1 cache be also present in the L2 cache. This leads to back-invalidations. Back-invalidations are needed when a block is evicted from L2 Cache and there is still a copy of that block in the L1 Cache. The block at L1 will be invalidated to ensure that the L1 never has any block that the L2 does not have. To keep it simple it is OK to send a back invalidation to the L1 upon every L2 eviction (without having to explicitly track at the L2 cache whether or not a block is in the L1 cache).
- Exclusive L2 caches: These caches maintain the invariant that a block with never reside in both the L1 and the L2. Whenever an access misses in L1 cache, the L2 cache is looked up. If it is a miss in the L2 the block is brought into the L1 from main memory without installing it in the L2. If the access were a hit in the L2 the block is brought into the L1 from the L2 and is invalidated in the L2. Note that the only way the L2 gets populated in this case is when the L1 evicts a block (whether clean or dirty), effectively making the L2 a victim cache. We will still use the term writeback only for the blocks that are dirty when evicted. However, the total writes to L2 will be the sum of writebacks and clean evictions from the L1.

4. Simulator Input

The simulator reads in a trace file in the following format:

```
r|w <hex address>
r|w <hex address>
...
```

The first argument is the operation. The character “r” means it is a read operation and the character “w” indicates a write operation. The second argument is the accessed address in hexadecimal format. Note that we assume the memory is byte addressable, so to obtain the block address you need to properly mask the provided address. The addresses can be up to 64 bits long.

Here is a sample trace:

```
r fff432
```

```
w ec422
r ec422
```

5. Simulator Output

Your simulator should output the following metrics at the end of a run. See validation runs for the exact format.

1. Memory hierarchy configuration and trace filename.
2. The following measurements:

a. Number of L1 reads
b. Number of L1 read misses
c. Number of L1 writes
d. Number of L1 write misses
e. L1 miss rate
   = (L1 read misses + L1 write misses)/(L1 reads + L1 writes)
f. Number of writebacks from L1
   dirty evictions from the L1 (with an Exclusive L2, clean L1 evictions are also written to L2 but don't count those here)
g. Number of L2 reads
   = L1 read misses + L1 write misses
h. Number of L2 read misses
i. Number of L2 writes
   = writebacks from L1, in case of an Inclusive or Non-inclusive L2
   = writebacks from L1 + clean L1 evictions, in case of an Exclusive L2
j. Number of L2 write misses
k. L2 miss rate
   = (L2 read misses + L2 write misses)/(L2 reads + L2 writes)
l. Number of writebacks from L2 to memory
m. Total memory traffic (or the number of blocks transferred to/from memory)
   Assuming the presence of an L2, this should match L2 read misses + L2 write misses + writebacks from L2 to memory, in case of a Non-inclusive or Exclusive L2 cache. In case of an Inclusive L2, if the blocks evicted from the L1 as a result of the back invalidation happen to be dirty, those should also be taken into account as writes to the memory (since they hold more recent data than what the memory contains).

6. Validation

6.1. Validation requirements

Sample simulation outputs will be provided on the website. Each validation output provided includes:

- The cache configuration to use
- All measurements described in Section 5

You must run your simulator and debug it until it matches these provided *validation outputs*. Your simulator must print outputs to the console (i.e., to the screen). Your output *must* match both numerically and in terms of formatting because the TA will literally "diff" your output with the correct output. Therefore, redirect the console output of your simulator to a temporary file. This can be achieved by placing

```
> <your_output_file>
```

after the simulator command. And then make sure you test whether or not your outputs match the expected output, by running this unix command:

`diff –iw <your_output_file> <posted_validation_file>`

The –iw flag tells diff to ignore case (uppercase vs. lowercase) and whitespace. Therefore, just make sure there is some whitespace (such as a tab) where you see whitespace in the validation output. If the above command returns without printing anything to the screen, your validation was successful. Do this for each validation output provided.

6.2. Compiling and Running the Simulator

Part of what you need to hand in is source code. The TA will compile and run your simulator. As such, you must meet the following strict requirements. Failure to meet these requirements will result in point deductions (see Section 10).

1. You *must* be able to compile and run your simulator on Eos Linux machines. This is required so that the TA can compile and run your simulator. If you are logging into an Eos machine remotely and do not know whether or not it is Linux (as opposed to SunOS), use the "uname" command to determine the operating system.
2. Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named *sim_cache*. The TA should be able to type only *make* (in the directory containing your sources and Makefile) and the simulator should compile successfully. The TA should be able to type only *make clean* to automatically remove object files and the simulator executable. An example Makefile will be posted on the web page, which you can copy and modify for your needs.
3. Your simulator must accept exactly 8 command-line arguments in the following order:
   sim_cache <BLOCKSIZE> <L1_SIZE> <L1_ASSOC> <L2_SIZE> <L2_ASSOC> <REPL_POLICY> <INCLUSION> <TRACE_FILE>
   - BLOCKSIZE: Positive int. Block size in bytes (assumed to be same for all caches)
   - L1_SIZE: Positive int. L1 cache size in bytes.
   - L1_ASSOC: Positive int. L1 set-associativity (1 is direct-mapped).
   - L2_SIZE: Positive int. L2 cache size in bytes; 0 signifies that there is no L2 cache.
   - L2_ASSOC: Positive int. L2 set-associativity (1 is direct-mapped).
   - REPL_POLICY: Positive int. 0 for LRU, 1 for FIFO, 2 for pseudoLRU, 3 for optimal.
   - INCLUSION: Positive int. 0 for non-inclusive, 1 for inclusive and 2 for exclusive.
   - TRACE_FILE: Character string. Full name of trace file including any extensions.

7. Experiments and Report

You will be provided with synthetic traces that are generated synthetically from several spec 2006 benchmark traces. We use the STM cloning framework to generate the synthetic traces. Plot the required figures for each of the following traces: MCF, GCC and LBM. You will have to use a combination of the simulator output as well as area and latency reports from a cache simulator called CACTI in order to generate a set of plots and discuss insights from your findings. CACTI results will be summarized and provided to you as a spreadsheet on the project webpage; you will not be required to run CACTI yourself.

The metrics of interest will generally be the miss rates of L1 and, if applicable, the L2 cache, the Average Access Times (AATs) for a given cache hierarchy and the die area of the cache hierarchy. The simulator output is set up to directly provide the miss rates.

The AATs have to computed by additionally using the hit and miss *latencies*, which come from the CACTI spreadsheet. "Access time (ns)" in the spreadsheet provides the hit times at each cache size. The miss penalty for the last level of cache is computed as follows.

MissPenalty = MemoryLatency + Blocksize/MemoryBandwidth – 1

where:

MemoryLatency = 25ns

MemoryBandwidth = 16GB/s

The following formulae may be employed to calculate the AAT.

For a memory hierarchy without an L2 :

$AAT_{noL2} = HitTime_{L1} + (MissRate_{L1} * MissPenalty)$

For a memory hierarchy with an L2 :

$AAT_{L2} = HitTime_{L1} + (MissRate_{L1} * HitTime_{L2}) + (MissRate_{L1} * MissRate_{L2} * MissPenalty)$

Area estimates also must be made by studying the CACTI spreadsheet. Cache area is the product of cache height and cache width, which are provided in the CACTI spreadsheet.

7.1. Exploring the L1 Cache Design

Assume that there is no L2 cache. Assume that LRU replacement policy is used for the L1 cache.

Plot 1

Fix the L1 block size at 64B and the L1 associativity at 4. Vary the cache size between 8KB, 16KB, 32KB and 64KB. Plot the L1 cache miss rates. The plot should have 4 data points per benchmark.

Plot 2

Fix the L1 block size at 64B and the L1 cache size at 32KB. Vary the associativity between 1, 2, 4 and 8. Plot the L1 cache miss rates. The plot should have 4 data points per benchmark.

Further analysis and discussion based on Plots 1 and 2

Plot the Average Access Time (AAT) for the configurations from Plots 1 and 2 (you can plot the AATs on a secondary y-axis within plots 1 and 2 or you may choose separate plots called 1a and 2a). Discuss how the cache size and associativity affect the AAT. Does a lower L1 miss rate always result in a lower AAT? Explain your answer. Try to find the best design that achieves a low AAT and, at the same time, doesn't need a large die area (note that this requires you to evaluate the area for each configuration from the CACTI spreadsheet).

7.2. Exploring the Replacement Policy

Assume that there is no L2 Cache.

Plot 3

Fix the L1 block size at 64B and the L1 associativity at 4. Vary cache size between 8KB, 16KB, 32KB and 64KB. For each configuration vary the replacement policy between LRU, FIFO and pseudoLRU. Plot the L1 Cache miss rates for each replacement policy. Note that for each benchmark this plot will have 3 data series (one per replacement policy) and each data series will have 4 data points (one per cache size). That is, 12 data points per benchmark.

Further analysis and discussion based on Plot 3

Discuss any interesting trends you see – make recommendations between the various replacement policies based on their complexity and the resulting miss rate.

7.3. Exploring the L2 Cache Design (This is only applicable to ECE521 students)

Fix block size at 64B, L1 cache size at 16KB, L1 associativity at 4 and the replacement policy to be LRU. Assume that the L2 Cache is always non-inclusive.

Plot 4

Fix L2 associativity at 8. Vary the L2 cache size between 128KB, 256KB, 512KB and 1MB. Plot the L2 cache miss rates. The plot should have 4 data points per benchmark.

Plot 5

Fix the L2 cache size at 128KB. Vary L2 associativity between 1, 2, 4 and 8. Plot the L2 cache miss rates. The plot should have 4 data points per benchmark.

Further analysis and discussion based on Plots 4 and 5

Plot the Average Access Time (AAT) for the configurations from Plots 4 and 5. Plot, discuss and make recommendations similar to how you did for Plots 1 and 2.

7.4. Exploring the Inclusion Property choices (This is only applicable to ECE521 students)

Fix the block size at 64B, the L1 associativity at 4 and the L2 associativity at 8. Vary L1 cache size between 8KB, 16KB, 32KB and 64KB. For each resulting configuration, vary the L2 cache size between 128KB, 256KB, 512KB and 1MB. There are a total of 16 configurations.

Plot 6

Plot the number of cache misses resulting in a memory read for each configuration for each inclusion policy (inclusive, exclusive and non-inclusive). Note that for each benchmark this plot with have 3 data series (one per inclusion policy) and each data series will have 16 points (one per configuration).

Plot 7

Assume that the AAT for Exclusive caches can be calculated similarly to the AAT calculation for non-inclusive caches. Even if blocks brought in from memory are directly installed in the L1 cache (no insertion time at L2 cache), we still encounter the hit latency of L2 cache as a result of checking the L2 cache before reading from memory. Plot the AAT when using non-inclusive, inclusive and exclusive L2 caches for each configuration. The number of data points will be similar to Plot 6.

Further analysis and discussion based on Plots 6 and 7

Discuss interesting trends and observations. When do you think an exclusive cache can be better than a non-inclusive cache? When could a non-inclusive cache significantly outperform and inclusive Cache? For this part, no area studies or design recommendations are required.

8. What to Submit via Wolfware

You must hand in a single zip file called **proj1.zip** that is no more than 1MB in size. Please respect the size limit on behalf of fellow students as the Wolfware submission space is limited. Notify the TA beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient. Below is an example showing how to create proj1.zip from an Eos Linux machine. Suppose you have a bunch of **source code files** (*.cc, *.h), the **Makefile**, and your project report (**report.pdf**). Run "zip proj1 *.cc *.h Makefile report.pdf"

proj1.zip must only contain the following (any deviation may result in point deductions):

- Project report: This must be a single PDF document named report.pdf (or a single MS Word document called report.doc). The report must include a cover page. A sample cover page will be provided and will contain the project title, an honor pledge, and space for writing your full name as electronic signature of the honor pledge. See Section 7 for the content required in the report.
- Source code: Commented simulator source code; use any number of .cc/.h files, .c/.h files, etc.
- Makefile: A sample Makefile will be posted on the project webpage. See section 6.2 for what is expected of the makefile.

Note: Zip only the *files* listed above, *not* the directory containing these files. Do not use tar or tar.gz or anything else. It has to be a zip archive.

9. Grading

The following is the high level break-up of the points awarded

- 30 points for substantial programming effort;
- 50 points for correct validation
- 20 points for experiments and report.

Validation will be scored as follows:

Item		Points for ECE463	Points for ECE521
L1 with LRU	Validation #0	9	4
	Validation #1	9	4
	Mystery A	8	5

L1 with FIFO, pseudoLRU	Validation #2	8	4
	Validation #3	8	4
	Mystery B	8	5
L1 with Optimal	Mystery E	10	10
L1, L2 work	Validation #4	NA	4
	Validation #5	NA	4
	Mystery D	NA	4
L1, L2 inclusion property	Validation #6	NA	4
	Validation #7	NA	4
	Mystery D	NA	4

Experiments and reports will be scored as follows:

Item		Points for ECE463	Points for ECE521
Experiments and Report	Plot 1 + Discussion	6	3
	Plot 2 + Discussion	6	3
	Plot 3 + Discussion	8	4
	Plot 4 + Discussion	NA	2
	Plot 5 + Discussion	NA	2
	Plot 6 + Discussion	NA	3
	Plot 7 + Discussion	NA	3

10. Deductions

- -10 point for each hour late, according to Wolfware timestamp. TIP: Submit whatever you have completed by the deadline just to be safe. If, after the deadline, you want to continue working on the project to get more features working and/or finish experiments, you may submit a second version of your project late. The TA will grade both the on-time and late versions (with late penalty applied to the late version), and use the one scoring more as the official submission. That said, please try and complete everything by the deadline.
- Up to -20 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties.
- Cheating: Source code that is flagged by tools available to us will be dealt with according to University Policy. This includes a 0 for the project and referral to the Office of Student Conduct.

11. Other Recommendations

Please follow the proposed simulator design style and speed guidelines to make it easier for the TA and the instructor to help you when you have bugs in implementation.

11.1. Keep your design modular

Building a simulator in a modular fashion is a good programming practice. Modular design means that you implement different functions for different tasks rather than implementing everything in one long function. It also means you group or organize related data and methods into *structures* or *classes* and capture the behavior of similarly-behaving components with little code duplication.

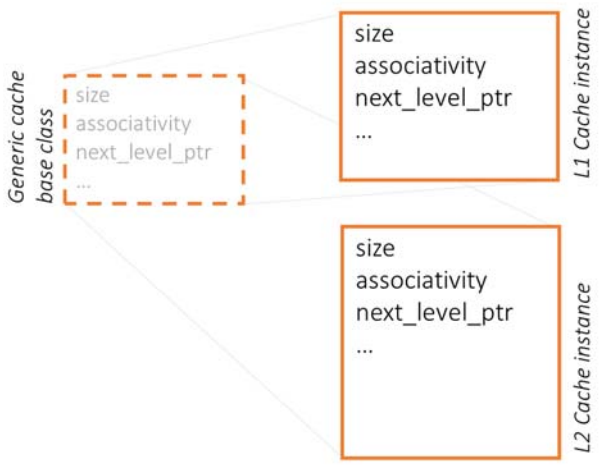

Figure 1: Sample Cache Hierarchy High Level Design

For example, different levels of the cache hierarchy may have different geometries and control policies, but at a high level they all are caches and to that extent may have the same high level methods and structural components. As a more specific example, notice that the L1 cache should have a function to locate the cache block to be evicted using a specific replacement policy. But every other cache level also needs to have a replacement policy, albeit a different policy. Further, whenever there is a hit at a cache level, the replacement policy related meta-data is modified, and whenever there is a miss at a cache level, the access is forwarded to the next level of cache.
It is strongly recommended that you observe these patterns of similarity (while being aware of the differences) and modularize the data structures that represents a generic cache. Use that as base class and allocate different instances (or derived class instances) of that class for each of the cache levels. Figure shows an example of such design.

11.2. Keep your simulator parametric

Always assume that parameters are changeable. As an example, you should not assume a specific cache size or block size. Such parameters should be parsed from the input of the simulator and then used to instantiate an instance of the generic class.

11.3. Keep backups

It is good practice to frequently make backups of all your project files, including source code, your report, etc. You can backup files to another hard drive (Eos account vs. home PC vs. laptop vs. USB memory stick etc.). Just keep consistent copies in multiple places.

11.4. Make your simulator fast

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important for a couple of reasons. First, the TA needs to test every student's simulator. Therefore, we are placing the constraint that your simulator must finish a single run in 2 minutes or less. If your simulator takes longer than 2 minutes to finish a single run, please see the TA. Second, you will be running many experiments: many cache configurations and multiple traces. Therefore, you will benefit from implementing a simulator that is reasonably fast. One simple thing you can do to make your simulator run faster is to compile it with a high optimization level. The example Makefile posted on the web page includes the –O3 optimization flag. Note that when you are debugging your simulator in a debugger (such as gdb), it is recommended that you compile without –O3 and, instead, compile with –g. Optimization includes register allocation and register-allocated variables are often not displayed properly in debuggers, which is why you want to disable optimization when using a

debugger. The –g flag tells the compiler to include symbols (variable names, etc.) in the compiled binary. The debugger needs this information to recognize variable names, function names, line numbers in the source code, etc. When you are done with debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

11.5. Use the VCL

In addition to using the grendel cluster, and other Eos linux machines, NCSU's Virtual Computing Lab. (VCL) allows you to reserve virtual linux machines. Go to http://vcl.ncsu.edu/

1. select "Reservation > New Reservation",
2. you'll see a popup asking for environment and duration,
3. for the environment, select "Linux Lab Machine (Realm RHEnterprise 6)"
4. you may choose to reserve a shell for up to 4 hours beginning now or later (with the possibility to extend reservation before it expires),
5. click "Create Reservation" button,
6. wait until the screen updates with a reservation and click "Connect!",
7. you will be provided an internet address that you can then ssh into using putty or other ssh clients (i.e. the same way that you remotely and securely login to other linux machines).