NC State University

Department of Electrical and Computer Engineering

ECE 463/521: Fall 2015 (Rotenberg)

Project #1: Cache Design, Memory Hierarchy Design (Version 1.0)

Due: Friday, Oct. 2, 5:00 PM

1. Groundrules

This is the first project for the course, so let me begin by discussing some groundrules:

1. All students must work alone. The project scope is reduced (but still substantial) for ECE 463 students, as detailed in this specification.
2. Sharing of code between students is considered <u>cheating</u> and will receive appropriate action in accordance with University policy. <u>The TAs will scan source code (from current and past semesters) through various tools available to us for detecting cheating. Source code that is flagged by these tools will be dealt with severely: 0 on the project and referral to the Office of Student Conduct for sanctions.</u>
3. A Wolfware message board will be provided for posting questions, discussing and debating issues, and making clarifications. It is an essential aspect of the project and communal learning. Students must not abuse the message board, whether inadvertently or intentionally. Above all, never post actual code on the message board (unless permitted by the TAs/instructor). When in doubt about posting something of a potentially sensitive nature, email the TAs and instructor to clear the doubt.
4. It is recommended that you do all your work in the C, C++ or Java languages. Exceptions must be approved by the instructor.
5. <u>Use of the Eos Linux environment is *required*. This is the platform where the TAs will compile and test your simulator.</u> (WARNING: If you develop your simulator on another platform, get it working on that platform, and then try to port it over to Eos Linux at the last minute, you may encounter major problems. Porting is not as quick and easy as you think. What's worse, malicious bugs can be hidden until you port the code to a different platform, which is an unpleasant surprise close to the deadline.)

2. Project Description

In this project, you will implement a flexible cache and memory hierarchy simulator and use it to compare the performance, area, and energy of different memory hierarchy configurations, using a subset of the SPEC-2000 benchmark suite.

3. Specification of Memory Hierarchy

Design a generic cache module that can be used at any level in a memory hierarchy. For example, this cache module can be "instantiated" as an L1 cache, an L2 cache, an L3 cache, and so on. Since it can be used at any level of the memory hierarchy, it will be referred to generically as CACHE throughout this specification.

3.1. Configurable parameters

CACHE should be configurable in terms of supporting any cache size, associativity, and block size, specified at the beginning of simulation:

- SIZE: Total bytes of data storage.
- ASSOC: The associativity of the cache (ASSOC = 1 is a direct-mapped cache).
- BLOCKSIZE: The number of bytes in a block.

There are a few constraints on the above parameters: 1) BLOCKSIZE is a power of two and 2) the number of *sets* is a power of two. *Note that ASSOC (and, therefore, SIZE) need not be a power of two.* As you know, the number of sets is determined by the following equation:

$$\# sets = \frac{SIZE}{ASSOC \times BLOCKSIZE}$$

3.2. Replacement policy

CACHE should use the LRU (least-recently-used) replacement policy.

Note regarding simulator output: When printing out the final contents of CACHE, you must print the blocks within a set based on their recency of access, i.e., print out the MRU block first, the next most recently used block second, and so forth, and the LRU block last. This is required so that your output is consistent with the output of the TAs' simulator.

3.3. Write policy

CACHE should use the WBWA (write-back + write-allocate) write policy.

- Write-allocate: A write that misses in CACHE will cause a block to be allocated in CACHE. Therefore, both write misses and read misses cause blocks to be allocated in CACHE.
- Write-back: A write updates the corresponding block in CACHE, making the block dirty. It does not update the next level in the memory hierarchy (next level of cache or memory). If a dirty block is evicted from CACHE, a writeback (*i.e.*, a write of the entire block) will be sent to the next level in the memory hierarchy.

3.4. Allocating a block: Sending requests to next level in the memory hierarchy

Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy, as shown in Fig. 1.

CACHE receives a read or write request from whatever is above it in the memory hierarchy (either the CPU or another cache). The only situation where CACHE must interact with the next level below it (either another CACHE or main memory) is when the read or write request misses in CACHE. When the read or write request misses in CACHE, CACHE must "allocate" the requested block so that the read or write can be performed.

Thus, let us think in terms of allocating a requested block X in CACHE. The allocation of requested block X is actually a two-step process. *The two steps must be performed in the following order*.

1. *Make space for the requested block X*. If there is at least one invalid block in the set, then there is already space for the requested block X and no further action is required (go to step 2). On the other hand, if all blocks in the set are valid, then a victim block V must be singled out for eviction, according to the replacement policy (Section 3.2). If this victim block V is dirty, then a write of the victim block V must be issued to the next level of the memory hierarchy.
2. *Bring in the requested block X*. Issue a read of the requested block X to the next level of the memory hierarchy and put the requested block X in the appropriate place in the set (as per step 1).

To summarize, when allocating a block, CACHE issues a write request (*only* if there is a victim block and it is dirty) followed by a read request, both to the next level of the memory hierarchy. Note that each of these two requests could themselves miss in the next level of the memory hierarchy (if the next level is another CACHE), causing a cascade of requests in subsequent levels. *Fortunately, you only need to correctly implement the two steps for an allocation locally within CACHE. If an allocation is correctly implemented locally (steps 1 and 2, above), the memory hierarchy as a whole will automatically handle cascaded requests globally.*

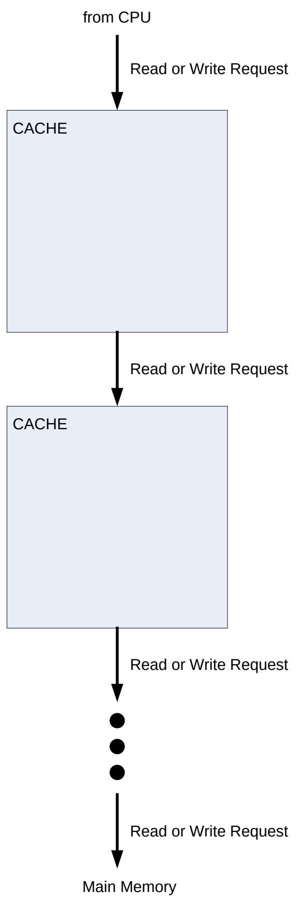

Fig. 1: Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy.

3.5. Updating state

After servicing a read or write request, whether the corresponding block was in the cache already (hit) or had just been allocated (miss), remember to update other state. This state includes LRU counters affiliated with the set as well as the valid and dirty bits affiliated with the requested block.

4. ECE 521 Students: Augment CACHE with a Victim Cache

Students enrolled in ECE 521 must additionally augment CACHE with a Victim Cache (VC).

In this project, consider the VC to be an extension implemented within CACHE. This preserves the clean abstraction of one or more instances of CACHE interacting in an overall memory hierarchy (see Fig. 1), where each CACHE may have a VC within it.

4.1. Victim cache can be enabled or disabled

Your simulator should be able to specify, for each CACHE, whether or not its VC is enabled.

4.2. Victim cache parameters

The VC is fully-associative and uses the LRU replacement policy. The number of blocks in the VC should be configurable. A CACHE and its VC have the same BLOCKSIZE.

4.3. Operation: Interaction between CACHE and its own VC

This section describes how CACHE and its VC interact, when its VC is enabled.

The only time CACHE and its VC *may* interact is when a read or write request misses in CACHE. In this case, the requested block X was not found in CACHE. If the corresponding set has at least one invalid block (*the set is not full*), then this set has never transferred a victim block to VC, therefore, VC cannot possibly have the requested block X and need not be searched. In this special case, CACHE need not interact with VC and instead goes directly to the next level. (If you want, as a sanity check, you can "assert" VC does not have requested block X.) Suppose, however, that the set does not have any invalid blocks (*the set is full*). In this more common case, a victim block V is singled out for eviction from CACHE, based on its replacement policy. CACHE issues a "swap request [X, V]" to its VC. Here is how the swap request works.

If VC has block X, then block X and block V are swapped:

- o block V moves from CACHE to VC (it is placed where block X was), and
- o block X moves from VC to CACHE (it is placed where block V was).

Note the following:

- o When swapping the blocks, everything is carried along with the blocks, including dirty bits.
- o After swapping the blocks, block X is considered to be the most-recently-used block in its set in CACHE and block V is considered to be the most-recently-used block in VC.
- o After swapping the blocks, the dirty bit of block X in CACHE may need to be updated.

On the other hand, if VC does not have block X, then three steps must be performed in the following order:

1. VC must make space for block V from CACHE. If VC is not full (it has at least one invalid block), then there is already space. If VC is full (only valid blocks), then VC

singles out a secondary victim block V2 for eviction based on its replacement policy. If secondary victim block V2 is dirty, then VC must issue a write of secondary victim block V2 to the next level in the memory hierarchy with respect to CACHE.

2. Block V (including its dirty bit) is put in VC, in the appropriate place as determined in step 1 above (*i.e.*, it replaces either an invalid block or secondary victim block V2).
3. CACHE issues a read of requested block X to the next level in the memory hierarchy.

5. Memory Hierarchies to be Explored in this Project

While Fig. 1 illustrates an arbitrary memory hierarchy, you will only study the memory hierarchy configurations shown in Fig. 2a (ECE 463) and Fig. 2b (ECE 521). Also, these are the only configurations the TAs will test.

For this project, all CACHEs in the memory hierarchy will have the same BLOCKSIZE.

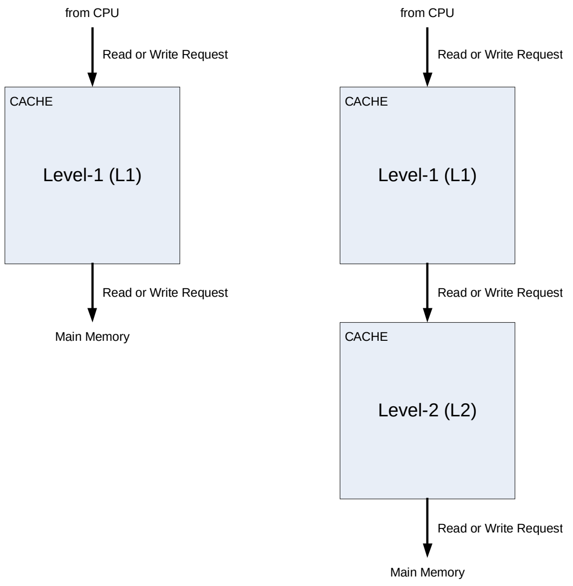

Fig. 2a: ECE463: Two configurations to be studied.

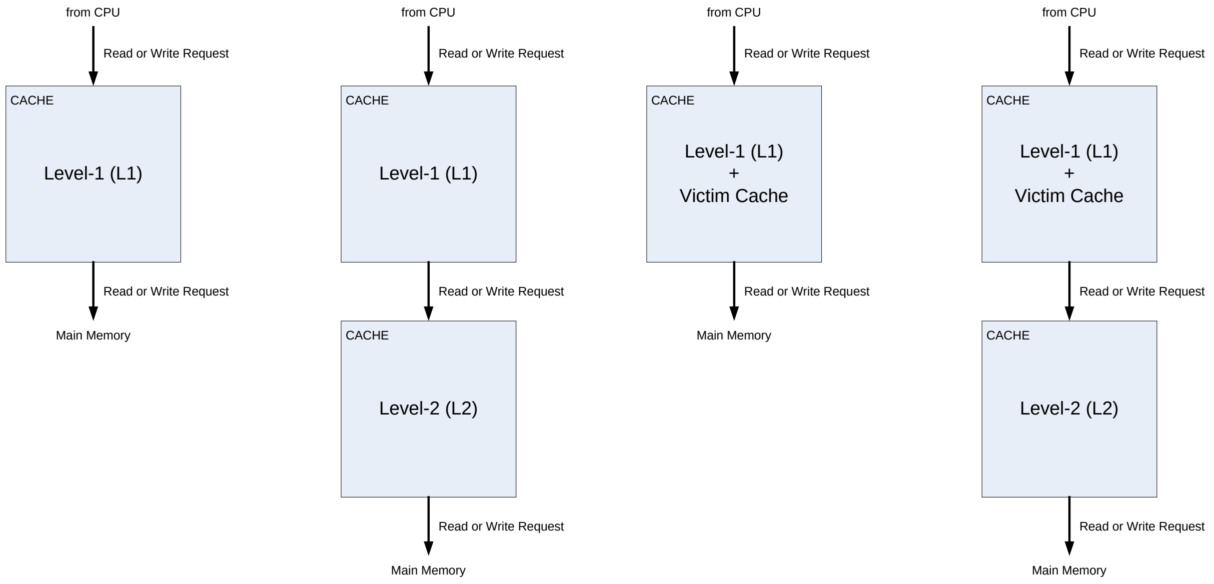

Fig. 2b: ECE521: Four configurations to be studied.

6. Inputs to Simulator

The simulator reads a trace file in the following format:

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

"r" (read) indicates a load and "w" (write) indicates a store from the processor.

Example:

```
r ffe04540
r ffe04544
w 0eff2340
r ffe04548
...
```

Traces are in the directory /afs/eos.ncsu.edu/courses/ece/ece521/lec/001/www/projects/proj1/traces. (It will be possible to download the traces directly from the web.)

NOTE:
All addresses are 32 bits. When expressed in hexadecimal format (hex), an address is 8 hex digits as shown in the example trace above. In the actual trace files, you may notice some addresses are comprised of fewer than 8 hex digits: this is because there are leading 0's which are not explicitly shown. For example, an address "`ffff`" is really "`0000ffff`", because all addresses are 32 bits, *i.e.*, 8 nibbles.

7. Outputs from Simulator

Your simulator should output the following: (see posted validation runs for exact format)
1. Memory hierarchy configuration and trace filename.
2. The final contents of all caches.
3. 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. number of swap requests from L1 to its VC (*swap requests=0 if no VC or VC disabled*)
   Note: As described earlier, a "swap request [X, V]" is when the L1 searches its VC for X.
f. swap request rate = $\boxed{\text{SRR}}$ = (swap requests)/(L1 reads + L1 writes)
g. number of swaps between L1 and its VC (*swaps=0 if no VC or VC is disabled*)
   Note: A "swap" is when X is found in VC, leading to swapping of X and V.
h. combined L1+VC miss rate = $\boxed{MR_{L1+VC}}$ =
   (L1 read misses + L1 write misses – swaps)/(L1 reads + L1 writes)
i. number of writebacks from L1 or its VC (if enabled), to next level
j. number of L2 reads (*should match: L1 read misses + L1 write misses – swaps*)
k. number of L2 read misses
l. number of L2 writes (*should match i: number of writebacks from L1 or its VC*)
m. number of L2 write misses
n. L2 miss rate (*from standpoint of stalling the CPU*) = $\boxed{MR_{L2}}$ = (L2 read misses)/(L2 reads)
o. number of writebacks from L2 to memory
p. total memory traffic = number of blocks transferred to/from memory
   (*with L2, should match k+m+o: L2 read misses + L2 write misses + writebacks from L2*)
   (*without L2, should match: L1 read misses + L1 write misses – swaps + writebacks from L1 or VC*)

8. Validation and Other Requirements

8.1. Validation requirements

Sample simulation outputs will be provided on the website. These are called "validation runs". You must run your simulator and debug it until it matches the validation runs.

Each validation run includes:
1. Memory hierarchy configuration and trace filename.
2. The final contents of all caches.
3. All measurements described in Section 7.

Your simulator must print outputs to the console (i.e., to the screen). (Also see Section 8.2 about this requirement.)

Your output must match both <u>numerically</u> and in terms of <u>formatting</u>, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following these two steps for each validation run:
1) Redirect the console output of your simulator to a temporary file. This can be achieved by placing `> your_output_file` after the simulator command.
2) Test whether or not your outputs match properly, by running this unix command:
```
diff –iw <your_output_file> <posted_output_file>
```
The –iw flags tell "diff" to treat upper-case and lower-case as equivalent and to ignore the amount of whitespace between words. Therefore, you do not need to worry about the exact number of spaces or tabs as long as there is some whitespace where the validation runs have whitespace.

8.2. Requirements for compiling and running simulator

You will hand in source code and the TAs 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 "Grading").

1. You must be able to compile and run your simulator on Eos Linux machines. This is required so that the TAs 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 TAs should be able to type only "make" and the simulator will successfully compile. The TAs should be able to type only "make clean" to automatically remove object (.o) 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 7 command-line arguments in the following order:

```
sim_cache  <BLOCKSIZE>
           <L1_SIZE>  <L1_ASSOC>  <VC_NUM_BLOCKS>
           <L2_SIZE>  <L2_ASSOC>
           <trace_file>
```

   - `BLOCKSIZE`: Positive integer. Block size in bytes. (Same block size for all caches in the memory hierarchy.)
   - `L1_SIZE`: Positive integer. L1 cache size in bytes.
   - `L1_ASSOC`: Positive integer. L1 set-associativity (1 is direct-mapped).
   - `VC_NUM_BLOCKS`: Positive integer. Number of blocks in the Victim Cache. `VC_NUM_BLOCKS` = 0 signifies that there is no Victim Cache.
   - `L2_SIZE`: Positive integer. L2 cache size in bytes. `L2_SIZE` = 0 signifies that there is no L2 cache.
   - `L2_ASSOC`: Positive integer. L2 set-associativity (1 is direct-mapped).
   - `trace_file`: Character string. Full name of trace file including any extensions.

   Example: 8KB 4-way set-associative L1 cache with 32B block size, 7-block victim cache affiliated with L1, 256KB 8-way set-associative L2 cache with 32B block size, gcc trace:

```
sim_cache  32  8192  4  7  262144  8  gcc_trace.txt
```

4. Your simulator must print outputs to the console (i.e., to the screen). This way, when a TA runs your simulator, he/she can simply redirect the output of your simulator to a filename of his/her choosing for validating the results.

8.3. Keeping 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 … keep consistent copies in multiple places) or removable media (Flash drive, etc.).

8.4. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important for a couple of reasons.

First, the TAs need 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 TAs as they may be able to help you speed up your simulator.

Second, you will be running many experiments: many memory hierarchy 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 with –g. Optimization includes register allocation. Often, register-allocated variables are 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 debugging, recompile with –O3 and without –g, to get the most efficient simulator again.

8.5. Test your simulator on Eos linux machines

You must test your simulator on Eos linux machines such as: *remote.eos.ncsu.edu* and *grendel.ece.ncsu.edu*.

9. Experiments and Report

Benchmarks

Use the GCC benchmark for all experiments.

Calculating AAT and Area

Table 1 gives names and descriptions of parameters and how to get these parameters.

Table 1. Parameters, descriptions, and how you obtain these parameters.

Parameter	*Description*	*How to get parameter*
SRR	Swap request rate.	From your simulator. See Section 7.
MR_{L1+VC}	Combined L1+VC miss rate.	
MR_{L2}	L2 miss rate (from standpoint of stalling the CPU).	
HT_{L1}	Hit time of L1.	"Access time (ns)" from CACTI tool. A spreadsheet of CACTI results will be made available on the Project-1 website. Some VC configurations may be too small for CACTI to analyze. For these, use HT_{VC} = CT. (CT is described in this table.)
HT_{VC}	Hit time of VC.	
HT_{L2}	Hit time of L2.	
Miss_Penalty	Time to fetch one block from main memory.	From Project-1 website.
CT	Cycle time of processor.	From Project-1 website.
A_{L1}	Die area of L1.	"Cache height x width (mm)" from CACTI tool. A spreadsheet of CACTI results will be made available on the Project-1 website.
A_{VC}	Die area of VC.	
A_{L2}	Die area of L2.	

For memory hierarchy *without* L2 cache:

$$\boxed{\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot \text{HT}_{\text{L1}} + (\text{swap requests}) \cdot \text{HT}_{\text{VC}} + (\text{L1 read misses} + \text{L1 write misses - swaps}) \cdot \text{Miss\_Penalty}}$$

$$\boxed{\text{Average access time (AAT)} = \frac{\text{Total access time}}{(\text{L1 reads} + \text{L1 writes})}}$$

$$\begin{aligned}\text{AAT} &= \text{HT}_{\text{L1}} + \left(\frac{\text{swap requests}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{HT}_{\text{VC}} + \left(\frac{\text{L1 read misses} + \text{L1 write misses - swaps}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{Miss\_Penalty}\\ &= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot \text{Miss\_Penalty}\end{aligned}$$

For memory hierarchy *with* L2 cache:

$$\boxed{\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot \text{HT}_{\text{L1}} + (\text{swap requests}) \cdot \text{HT}_{\text{VC}} + (\text{L1 read misses} + \text{L1 write misses - swaps}) \cdot \text{HT}_{\text{L2}} + (\text{L2 read misses}) \cdot \text{Miss\_Penalty}}$$

$$\boxed{\text{Average access time (AAT)} = \frac{\text{Total access time}}{(\text{L1 reads} + \text{L1 writes})}}$$

$$\begin{aligned}\text{AAT} &= \text{HT}_{\text{L1}} + \left(\frac{\text{swap requests}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{HT}_{\text{VC}} + \left(\frac{\text{L1 read misses} + \text{L1 write misses - swaps}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{HT}_{\text{L2}} + \left(\frac{\text{L2 read misses}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{Miss\_Penalty}\\
&= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot \text{HT}_{\text{L2}} + \left(\frac{\text{L2 read misses}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{Miss\_Penalty}\\
&= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot \left(\text{HT}_{\text{L2}} + \left(\frac{1}{\text{MR}_{\text{L1+VC}}}\right) \cdot \left(\frac{\text{L2 read misses}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{Miss\_Penalty}\right)\\
&= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot \left(\text{HT}_{\text{L2}} + \left(\frac{\text{L1 reads} + \text{L1 writes}}{\text{L1 read misses} + \text{L1 write misses - swaps}}\right) \cdot \left(\frac{\text{L2 read misses}}{\text{L1 reads} + \text{L1 writes}}\right) \cdot \text{Miss\_Penalty}\right)\\
&= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot \left(\text{HT}_{\text{L2}} + \left(\frac{\text{L2 read misses}}{\text{L1 read misses} + \text{L1 write misses - swaps}}\right) \cdot \text{Miss\_Penalty}\right)\\
&= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot \left(\text{HT}_{\text{L2}} + \left(\frac{\text{L2 read misses}}{\text{L2 reads}}\right) \cdot \text{Miss\_Penalty}\right)\\
&= \text{HT}_{\text{L1}} + \text{SRR} \cdot \text{HT}_{\text{VC}} + \text{MR}_{\text{L1+VC}} \cdot (\text{HT}_{\text{L2}} + \text{MR}_{\text{L2}} \cdot \text{Miss\_Penalty})\end{aligned}$$

The total area of the caches:

$$\text{Area} = \text{A}_{\text{L1}} + \text{A}_{\text{VC}} + \text{A}_{\text{L2}}$$

If a particular cache does not exist in the memory hierarchy configuration, then its area is 0.

9.1. L1 cache exploration: SIZE and ASSOC

<u>GRAPH #1</u> (***total number of simulations: 55***)

For this experiment:

- L1 cache: SIZE is varied, ASSOC is varied, BLOCKSIZE = 32.
- Victim Cache: None.
- L2 cache: None.

Plot L1 miss rate on the y-axis versus $\log_2$(SIZE) on the x-axis, for eleven different cache sizes: SIZE = 1KB, 2KB, … , 1MB, in powers-of-two. (That is, $\log_2$(SIZE) = 10, 11, …, 20.) The graph should contain five separate curves (*i.e.*, lines connecting points), one for each of the following associativities: direct-mapped, 2-way set-associative, 4-way set-associative, 8-way set-associative, and fully-associative. All points for direct-mapped caches should be connected with a line, all points for 2-way set-associative caches should be connected with a line, etc.

Discussion to include in your report:

1. Discuss trends in the graph. For a given associativity, how does increasing cache size affect miss rate? For a given cache size, what is the effect of increasing associativity?
2. Estimate the *compulsory miss rate* from the graph.
3. For each associativity, estimate the *conflict miss rate* from the graph.

<u>GRAPH #2</u> (***no additional simulations with respect to GRAPH #1***)

Same as GRAPH #1, but the y-axis should be AAT instead of L1 miss rate.

Discussion to include in your report:

1. For a memory hierarchy with only an L1 cache and BLOCKSIZE = 32, which configuration yields the best (*i.e.*, lowest) AAT?

<u>GRAPH #3</u> (***total number of simulations: 45***)

Same as GRAPH #2, except make the following changes:

- Add the following L2 cache to the memory hierarchy: 512KB, 8-way set-associative, same block size as L1 cache.
- Vary the L1 cache size only between 1KB and 256KB (since L2 cache is 512KB).

Discussion to include in your report:

1. With the L2 cache added to the system, which L1 cache configurations result in AATs close to the best AAT observed in GRAPH #2 (*e.g.*, within 5%)?
2. With the L2 cache added to the system, which L1 cache configuration yields the best (*i.e.*, lowest) AAT? How much lower is this optimal AAT compared to the optimal AAT in GRAPH #2?

3. Compare the *total area* required for the optimal-AAT configurations with L2 cache (GRAPH #3) versus without L2 cache (GRAPH #2).

9.2. L1 cache exploration: SIZE and BLOCKSIZE

GRAPH #4 (***total number of simulations: 24***)

For this experiment:

- L1 cache: SIZE is varied, BLOCKSIZE is varied, ASSOC = 4.
- Victim Cache: None.
- L2 cache: None.

Plot L1 miss rate on the y-axis versus $\log_2$(BLOCKSIZE) on the x-axis, for four different block sizes: BLOCKSIZE = 16, 32, 64, and 128. (That is, $\log_2$(BLOCKSIZE) = 4, 5, 6, and 7.) The graph should contain six separate curves (*i.e.*, lines connecting points), one for each of the following L1 cache sizes: SIZE = 1KB, 2KB, …, 32KB, in powers-of-two. All points for SIZE = 1KB should be connected with a line, all points for SIZE = 2KB should be connected with a line, etc.

Discussion to include in your report:

1. Discuss trends in the graph. Do smaller caches prefer smaller or larger block sizes? Do larger caches prefer smaller or larger block sizes? Why? As block size is increased from 16 to 128, is the tradeoff between *exploiting more spatial locality* versus *increasing cache pollution* evident in the graph, and does the balance between these two factors shift with different cache sizes?

9.3. L1 + L2 co-exploration

GRAPH #5 (***total number of simulations: 44***)

For this experiment:

- L1 cache: SIZE is varied, BLOCKSIZE = 32, ASSOC = 4.
- Victim Cache: None.
- L2 cache: SIZE is varied, BLOCKSIZE = 32, ASSOC = 8.

Plot AAT on the y-axis versus $\log_2$(L1 SIZE) on the x-axis, for nine different L1 cache sizes: L1 SIZE = 1KB, 2KB, … , 256KB, in powers-of-two. (That is, $\log_2$(L1 SIZE) = 10, 11, …, 18.) The graph should contain six separate curves (*i.e.*, lines connecting points), one for each of the following L2 cache sizes: 32KB, 64KB, 128KB, 256KB, 512KB, 1MB. All points for the 32KB L2 cache should be connected with a line, all points for the 64KB L2 cache should be connected with a line, etc. *For a given curve / L2 cache size, only plot points for which the L1 cache is smaller than the L2 cache. For example, the curve for L2 SIZE = 32KB should have only 5 points total, corresponding to L1 SIZE = 1KB through 16KB. In contrast, the curve for L2 SIZE = 512KB (and 1MB) will have 9 points total, corresponding to L1 SIZE = 1KB through 256KB.*

There should be a total of 44 points (5+6+7+8+9+9, for the six curves / L2 cache sizes, respectively).

Discussion to include in your report:

1. Which memory hierarchy configuration yields the best (*i.e.*, lowest) AAT?
2. Which memory hierarchy configuration has the smallest total area, that yields an AAT within 5% of the best AAT?

9.4. Victim cache study (ECE 521 students only)

<u>GRAPH #6</u> **(*total number of simulations: 42*)**

For this experiment:

- L1 cache: SIZE is varied, ASSOC is varied, BLOCKSIZE = 32.
- Victim Cache: # entries is varied.
- L2 cache: SIZE = 64KB, ASSOC = 8, BLOCKSIZE = 32.

Plot AAT on the y-axis versus $\log_2$(L1 SIZE) on the x-axis, for six different L1 cache sizes: L1 SIZE = 1KB, 2KB, … , 32KB, in powers-of-two. (That is, $\log_2$(L1 SIZE) = 10, 11, …, 15.) The graph should contain seven separate curves (*i.e.*, lines connecting points), one for each of the following scenarios:

- Direct-mapped L1 cache with no Victim Cache.
- Direct-mapped L1 cache with 2-entry Victim Cache.
- Direct-mapped L1 cache with 4-entry Victim Cache.
- Direct-mapped L1 cache with 8-entry Victim Cache.
- Direct-mapped L1 cache with 16-entry Victim Cache.
- 2-way set-associative L1 cache with no Victim Cache.
- 4-way set-associative L1 cache with no Victim Cache.

Discussion to include in your report:

1. Discuss trends in the graph. Does adding a Victim Cache to a direct-mapped L1 cache yield performance comparable to a 2-way set-associative L1 cache of the same size? …for which L1 cache sizes? …for how many Victim Cache entries?
2. Which memory hierarchy configuration yields the best (*i.e.*, lowest) AAT?
3. Which memory hierarchy configuration has the smallest total area, that yields an AAT within 5% of the best AAT?

10. What to Submit via Wolfware

You must hand in a single zip file called **project1.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 and exceeding your quota will cause problems come submission time. (Notify the TAs beforehand if you have special space requirements. However, a zip file of 1MB should be sufficient.)

Below is an example showing how to create **project1.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).

```
zip  project1  *.cc  *.h  Makefile  report.pdf
```

project1.zip must contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

1. **Project report**. This must be a single PDF document named **report.pdf**. The report must include the following:
   - A cover page with the project title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page will be posted on the project website.
   - See Section 9 for the required content of the report.
2. **Source code**. You must include the commented source code for the simulator program itself. You may use any number of .cc/.h files, .c/.h files, etc.
3. **Makefile**. See Section 8.2, item #2, for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

11. Grading

Table 2 shows the breakdown of points for the project:

[30 points] **Substantial programming effort.**

[50 points] **A working simulator**, as determined by matching validation runs.

[20 points] **Experiments and report.** If your simulator works for L1 only, you can get credit for experiments with L1. If your simulator works for L1 and L1+L2, you can get credit for L1 and L1+L2 experiments. And so on.

Table 2. Breakdown of points.

[30 points] Substantial programming effort.

Item	Points (ECE 463)	Points (ECE 521)
Substantial simulator turned in	30 points	30 points

[50 points] A working simulator: match validation runs.

Item		Points (ECE 463)	Points (ECE 521)
L1 works	validation run #1	9 points	4 points
	validation run #2	9 points	4 points
	mystery run A	8 points	5 points
L1, L2 works	validation run #3	8 points	4 points
	validation run #4	8 points	4 points
	mystery run B	8 points	5 points
L1+VC works	validation run #5	*not applicable*	4 points
	validation run #6		4 points
	mystery run C		4 points
L1+VC, L2 works	validation run #7		4 points
	validation run #8		4 points
	mystery run D		4 points

[20 points] Experiments and report.

Item		Points (ECE 463)	Points (ECE 521)
Experiments and Report	GRAPH #1 + disc.	5 points	4 points
	GRAPH #2 + disc.	3 points	3 points
	GRAPH #3 + disc.	4 points	4 points
	GRAPH #4 + disc.	4 points	3 points
	GRAPH #5 + disc.	4 points	3 points
	GRAPH #6 + disc.	*not applicable*	3 points

Various deductions (out of 100 points):

-1 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. If you choose to do this, the TA will grade both the on-time version and the late version (*late penalty applied to the late version*), and take the maximum score of the two. Hopefully you will complete everything by the deadline, however.

Up to -10 points for not complying with specific procedures. Follow all procedures very carefully to avoid penalties. Complete the **SUBMISSION CHECKLIST** that is posted on the project website, to make sure you have met all requirements.

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.