NC State University

Department of Electrical and Computer Engineering

ECE 463/563: Fall 2022 (Rotenberg)

Project #1: Cache Design, Memory Hierarchy Design (Version 3.0)

Due: Fri., Oct. 14, 11:59 PM

1. Preliminary Information

1.1. Academic integrity

- Academic integrity:
  - **Source code:** Each student must design and write their source code alone. They must do this (design and write their source code) without the assistance of any other person in ECE 463/563 or not in ECE 463/563. They must do this (design and write their source code) without searching the web for past semesters' projects and without searching the web for source code with similar goals (*e.g.*, modeling computer architecture components such as caches, predictors, pipelines, *etc*.), which is strictly forbidden. They must do this (design and write their source code) without looking at anyone else's source code, without obtaining electronic or printed copies of anyone else's source code, *etc*.
  - **Explicit debugging:** With respect to "explicit debugging" as part of the coding process (*e.g.*, using a debugger, inspecting code for bugs, inserting prints in the code, iteratively applying fixes, *etc*.), each student must explicitly debug their code without the assistance of any other person in ECE 463/563 or not in ECE 463/563.
  - **Report:** Each student must run their own experiments, collect and process their own data, and write their own report. Plagiarism (lifting text, graphs, illustrations, *etc*., from someone else, whether one adjusts the originals or not) is prohibited. Using someone else's data (in raw form and/or graph form) is prohibited. Fabricating data is prohibited.
  - **Sanctions:** The sanctions for violating the academic integrity policy are (1) a score of 0 on the project and (2) academic integrity probation for a first-time offense or suspension for a subsequent offense (the latter sanctions are administered by the Office of Student Conduct). Note that, when it comes to academic integrity violations, both the giver and receiver of aid are responsible and both are sanctioned. Please see the following RAIV form which has links to various policies and procedures and gives a sense of how academic integrity violations are confronted, documented, and sanctioned: RAIV form.
  - **Enforcement:** The TAs will scan source code (from current and past semesters) through tools available to us for detecting cheating. The outputs from these tools, combined with in-depth manual analysis of these outputs, will be the basis for investigating suspected academic integrity violations. TAs will identify suspected plagiarism and/or data fabrication and these cases will be investigated.
- Reasonable assistance: If a student has any doubts or questions, or if a student is stumped by a bug, the student is encouraged to seek assistance using both of the following channels.
  - Students may receive assistance from the TAs and instructor.

- Students are encouraged to post their doubts, questions, and obstacles, on the Moodle message board for this project. The instructor and TA will moderate the message board to ensure that reasonable assistance is provided to the student. Other students are encouraged to contribute answers so long as no source code is posted.

  * An example of reasonable assistance via the message board: Student A: "*I'm encountering the following problem: I'm getting fewer writebacks from L2 to main memory than the validation runs. Has anyone else encountered something like this?*" Student B: "*Yes, I encountered something similar, and you might want to take a look at how you are doing XYZ because the problem has to do with such-and-such.*"

  * Another example of a reasonable exchange: Student A: "*I'm unsure how to split the address into tag and index, and how to discard the block offset bits. I've successfully computed the # bits for each but I am stuck on how to manipulate the address in C/C++. Do you have any advice?*" Instructor/TA/Student B: "*We suggest you use an unsigned integer type for the address and use bitwise manipulation, such as ANDs (&), ORs (|), and left/right shifts (<<. >>) to extract values from the unsigned integer, the same as one would do in Verilog.*"

  * Another example of a reasonable exchange: Student A: "*I'm unsure how to dynamically allocate memory, such as dynamically allocating a 1D array of structs/classes or (more appropriately for a cache) a 2D array of structs/classes. Can you point me to some references on this?*" Instructor/TA/Student B: "*Sure, here is a web site or reference book that discusses dynamic memory allocation including 1D and 2D arrays.*"

- **The intent of the academic integrity policy is to ensure students code and explicitly debug their code by themselves. It is NOT our intent to stifle robust, interesting, and insightful discussion and Q&A that is helpful for students (and the instructor and TA) to learn together. We also would like to help students get past bugs by offering advice on where they may be doing things incorrectly, or where they are making incorrect assumptions, *etc*., from an academic and conceptual standpoint.**

1.2. Reduced project scope for ECE 463 students

The project scope is reduced but still substantial for ECE 463 students, as detailed in this specification.

1.3. Programming languages for this project

You must implement your project using the C, C++, or Java languages, for two reasons. First, these languages are preferred for computer architecture performance modeling. Second, our Gradescope autograder only supports compilation of these languages.

1.4. Responsibility for self-grading your project via Gradescope

You will submit, validate, and SELF-GRADE your project via Gradescope; the TAs will only manually grade the report. While you are developing your simulator, you are required to frequently check via Gradescope that your code compiles, runs, and gives expected outputs with respect to your current progress. This is necessary to resolve porting issues in a timely fashion (*i.e.*, well before the deadline), caused by different compiler versions in your programming environment and the Gradescope backend. This is also necessary to resolve non-compliance issues (*i.e.*, how you specify the simulator's command-line arguments, how you format the simulator's outputs, *etc.*) in a timely fashion (*i.e.*, well before 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 2006 benchmark suite, SPEC 2017 benchmark suite, and/or microbenchmarks.

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. ASSOC = # blocks in the cache = SIZE/BLOCKSIZE is a fully-associative 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.

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 Figure 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.*

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.

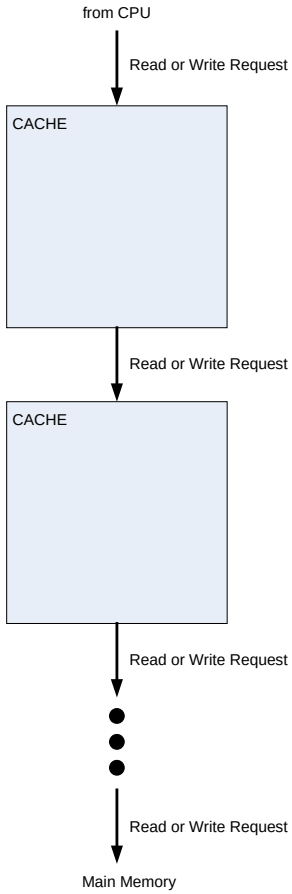

Figure 1. Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy.

4. ECE 563 Students: Augment CACHE with Stream-Buffer Prefetching

Students enrolled in ECE 563 must additionally augment CACHE with a prefetch unit. The prefetch unit implements Stream Buffers.

In this project, consider the prefetch unit 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 Figure 1), where each CACHE may have a prefetch unit within it.

4.1. CACHE should support a configurable prefetch unit

Your generic implementation of CACHE should support a configurable prefetch unit as follows. The prefetch unit has N Stream Buffers. Each Stream Buffer contains M memory blocks. Both N and M should be configurable. Setting N=0 disables the prefetch unit.

4.2. Operation of a single Stream Buffer

A Stream Buffer is a simple queue that is capable of holding M *consecutive* memory blocks.

A Stream Buffer has a single valid bit that indicates the validity of the buffer as a whole. If its valid bit is 0, it means the Stream Buffer is empty and doesn't contain a prefetch stream. If its valid bit is 1, it means the Stream Buffer is full and contains a prefetch stream (M consecutive memory blocks).

When CACHE receives a read or write request for block X, both CACHE and its Stream Buffer are checked for a hit. Note that all Stream Buffer entries, not just the first entry (as in the original Stream Buffer paper), are searched for block X. There are four possible scenarios:

1. **Scenario #1 (create a new prefetch stream): Requested block X misses in CACHE and misses in Stream Buffer:** Handle the miss in CACHE as usual (see Section 3.4). In addition to fetching the requested block X into CACHE, prefetch the next M consecutive memory blocks into the Stream Buffer. That is, prefetch memory blocks X+1, X+2, …, X+M into the Stream Buffer, thereby replacing the entire contents of the Stream Buffer. Note that prefetches are implemented by issuing read requests to the next level in the memory hierarchy.[1] Also note that replacing the contents of the Stream Buffer does not involve any writebacks from the Stream Buffer: this will be explained in Section 4.4.
2. **Scenario #2 (benefit from and continue a prefetch stream): Requested block X misses in CACHE and hits in the Stream Buffer:** Perform an allocation in CACHE as follows. First, make space in CACHE for the requested block X (as described in Section 3.4). Second, instead of fetching the requested block X from the next level in the memory hierarchy, copy the requested block X from the Stream Buffer into CACHE (since the

[1] For accurate performance accounting using the Average Access Time (AAT) expression, you will need to convey to the next level in the memory hierarchy that these read requests are prefetches. This will enable the next level in the memory hierarchy to distinguish between 1) its read misses that originated from normal read requests versus 2) its read misses that originated from prefetch read requests. Note that this is only needed for accurate performance accounting.

Stream Buffer contains the requested block X, in this scenario). Next, manage the Stream Buffer as illustrated in Figure 2. Notice in the "before" picture, the fourth entry of the Stream Buffer hit (it contained the requested block X). As shown in the "after" picture, all blocks before and including block X (X-3, X-2, X-1, X) are removed from the Stream Buffer, the blocks after block X (X+1, X+2) are "shifted up", and the newly freed entries are refilled by prefetching the next consecutive blocks (issue prefetches of blocks X+3, X+4, X+5, X+6). A non-shifting circular buffer implementation, based on a head pointer that points to the least block address in the prefetch stream, is more efficient in real hardware *and* in software simulators, and is illustrated in Figure 3.

3. **Scenario #3 (do nothing): Requested block X hits in CACHE and misses in the Stream Buffer:** In this case, nothing happens with respect to the Stream Buffer.
4. **Scenario #4 (continue prefetch stream to stay in sync with demand stream): Requested block X hits in CACHE and hits in the Stream Buffer:** Manage the Stream Buffer identically to Scenario #2. The only difference is that the requested block X hit in CACHE, so there is no transfer from Stream Buffer to CACHE.

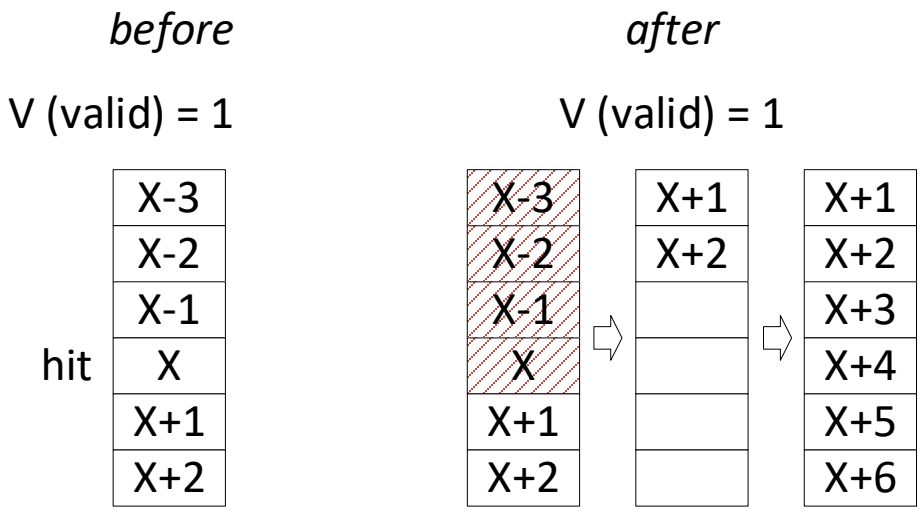

Figure 2. Managing the Stream Buffer when there is a hit in the Stream Buffer (scenarios #2 and #4).

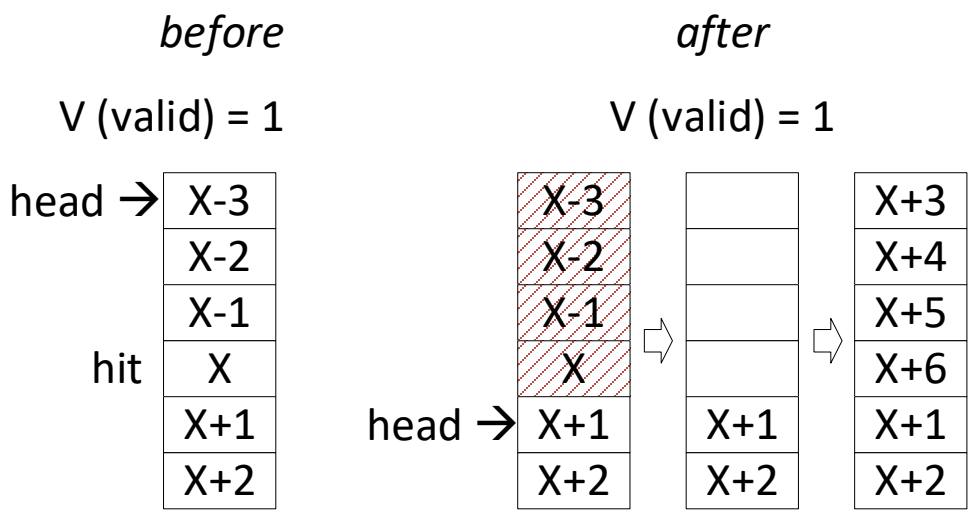

Figure 3. A non-shifting circular buffer implementation is more efficient in hardware *and* in software simulators.

4.3. Multiple Stream Buffers

The operation of a single Stream Buffer, described in the previous section, extends to multiple Stream Buffers. The main difference is that all Stream Buffers are checked for a hit.

For Scenario #1 (request misses in CACHE and misses in all Stream Buffers), one of the Stream Buffers must be chosen for the new prefetch stream: select the least-recently-used Stream Buffer, *i.e.*, apply the LRU policy to the Stream Buffers as a whole. When a new stream is prefetched into a particular Stream Buffer (Scenario #1), or a particular Stream Buffer supplies a requested block to CACHE (Scenario #2), or we are keeping a Stream Buffer in sync (Scenario #4), that Stream Buffer becomes the most-recently-used buffer.

Policy for multiple Stream Buffer hits:

It is possible for two or more Stream Buffers to have some blocks in common (redundancy). For example, suppose all Stream Buffers are initially invalid and CACHE is empty; the CPU requests block X which creates the prefetch stream X+1 to X+6 in a first Stream Buffer (assume M=6); and then the CPU requests block X-2 which creates the prefetch stream X-1 to X+4 in a second Stream Buffer; thus, after these initial two misses, the Stream Buffers have X+1 to X+4 in common. Other scenarios create redundancy as well, such as one continuing prefetch stream reaching the start of another prefetch stream.

Redundancy means that a given request may hit in multiple Stream Buffers. Managing multiple Stream Buffers as in Figure 2, for the same hit, results in redundant prefetches because the multiple Stream Buffers will all try to continue their overlapping streams. **A simple solution is to only consider the hit to the most-recently-used Stream Buffer among those that hit and ignore the other hits.** From a simulator standpoint, this could mean (for example) searching Stream Buffers for a hit in recency order, and stopping at the first hit. Only *that* Stream Buffer is managed as shown in Figure 2, *i.e.*, only *that* Stream Buffer continues its prefetch stream.

4.4. Assume Stream Buffers are updated by writebacks with no effect on recency (no explicit modeling required in your simulator)

A Stream Buffer never contains dirty blocks, that is, it never contains a block whose content differs from the same block in the next level of the memory hierarchy. The benefit of this design is that replacing the contents of the Stream Buffer will never require writebacks from the Stream Buffer.

In this section, we discuss a Stream Buffer complication that we will handle *conceptually*. The problem and solution are only discussed out of academic interest. **The solution does not require any explicit support in the simulator.**

Consider that a dirty copy of block Y may exist in CACHE while a clean copy of block Y exists in a Stream Buffer. Here is a simple example of how we can get into this situation (assume M=6 for the example):

- Write request to block Y misses in CACHE. Block Y is allocated in CACHE, write is performed, and block Y is dirty in CACHE. Prefetch stream Y+1 to Y+6 is created in a first Stream Buffer, although this is not germane for this example.
- Then, a request to block Y-2 misses in CACHE. Prefetch stream Y-1 to Y+4 is created in a second Stream Buffer. Thus, at this point, CACHE has a dirty copy of block Y and the second Stream Buffer has a clean copy of block Y.

Now, suppose CACHE evicts its dirty copy of block Y (*e.g.*, it is replaced by a missed block Z) before referencing it again (fyi: referencing it as a hit might wipe it from the Stream Buffer to keep the latter's prefetch stream in sync with demand references, as per scenario #4). Stale block Y still exists in the Stream Buffer which could lead to incorrect operation in the future, namely, when the CPU requests block Y again and hits on the stale copy in the Stream Buffer.

We will assume a solution that does NOT require any code in your simulator. When a dirty block Y is evicted from CACHE (*i.e.*, when there is a writeback), any Stream Buffers that contain block Y update their copy of block Y. In this way, a Stream Buffer's copy of block Y will remain clean and up to date with respect to the next level, since the writeback is performed not only in the next level but also in the Stream Buffer. In addition, let us also assume that this operation does NOT update recency among Stream Buffers. Therefore, the only effect is updating data, and your simulator does not model data.

5. Memory Hierarchies to be Explored in this Project

While Figure 1 illustrates an arbitrary memory hierarchy, you will only study the memory hierarchy configurations shown in Figure 4 (ECE 463) and Figure 5 (ECE 563). Also, these are the only configurations that Gradescope will test.

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

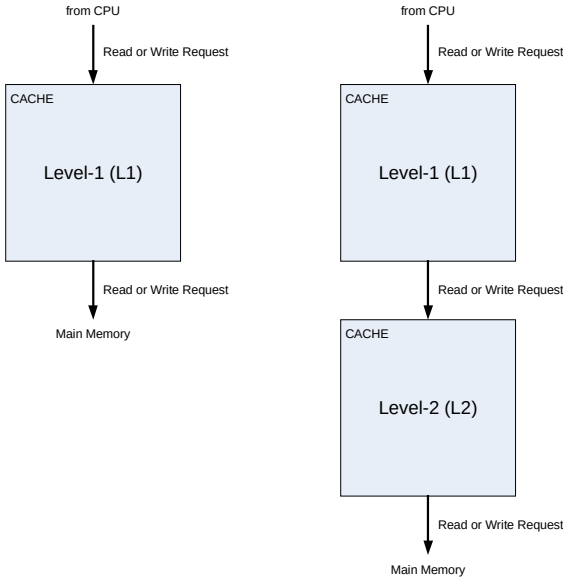

Figure 4. ECE 463: Two configurations to be studied.

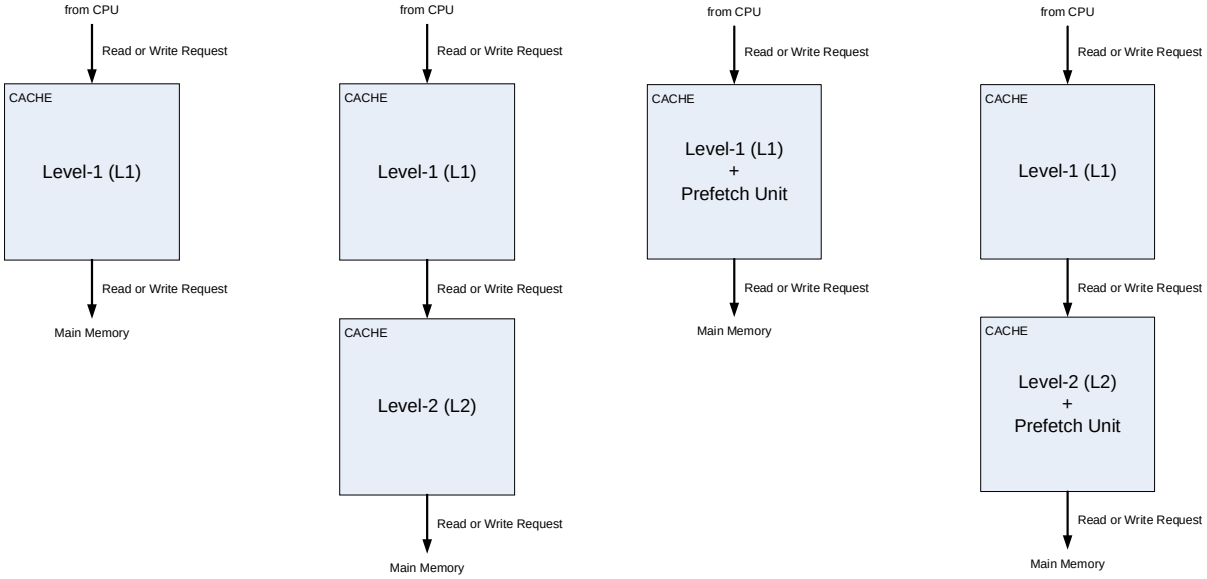

Figure 5. ECE 563: Four configurations to be studied.

6. Inputs to Simulator

6.1. Traces

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 posted on the Moodle website.

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.

6.2. Command-line arguments to the simulator

The simulator executable built by your Makefile must be named "sim" (the Makefile is discussed in Section 8).

Your simulator must accept exactly 8 command-line arguments in the following order:

```
sim     <BLOCKSIZE>
        <L1_SIZE>  <L1_ASSOC>
        <L2_SIZE>  <L2_ASSOC>
        <PREF_N>  <PREF_M>
        <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, L1_SIZE/BLOCKSIZE is fully-associative).

- *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, L2_SIZE/BLOCKSIZE is fully-associative).
- *PREF_N*: Positive integer. Number of Stream Buffers in the L1 prefetch unit (if there is no L2) or the L2 prefetch unit (if there is an L2). *PREF_N* = 0 disables the prefetch unit.
- *PREF_M*: Positive integer. Number of memory blocks in each Stream Buffer in the L1 prefetch unit (if there is no L2) or the L2 prefetch unit (if there is an L2).
- *trace_file*: Character string. Full name of trace file including any extensions.

Example: 8KB 4-way set-associative L1 cache with 32B block size, 256KB 8-way set-associative L2 cache with 32B block size, L2 prefetch unit has 3 stream buffers with 10 blocks each, gcc trace:

```
sim  32  8192  4  262144  8  3  10  gcc_trace.txt
```

Some additional points:

- You may assume that only valid arguments will be applied to your simulator by Gradescope. Thus, while it is good practice for programmers to check arguments (*e.g.*, check that the specified trace file exists and can be opened; *e.g.*, check that SIZE, ASSOC, and BLOCKSIZE lead to a feasible cache; *e.g.*, check that the correct number of arguments are passed in), you are NOT required to do so.
- Although ECE 463 students do not implement prefetching, they must still parse the prefetching-related arguments the same as ECE 563 students. ECE 463 students may assume that, for their projects, Gradescope will always specify "0 0" for arguments PREF_N and PREF_M.
- ECE 563 students may notice that, for this project, there is only one pair of prefetcher configuration arguments (PREF_N, PREF_M) and that these are applied only to the last level of cache in the memory hierarchy. This is consistent with Figure 5.

7. Outputs from Simulator

Your simulator should output the following:
(See Section 8 regarding the formatting of these outputs and validating your simulator.)

1. Memory hierarchy configuration and trace filename.
2. The final contents of all caches and their stream buffers (if applicable). For cache contents, blocks within a set must be printed out from most-recently-used to least-recently-used. Omit (do not print) invalid blocks within a set, if there are any. If a given set has no valid blocks, omit (do not print) that set. Stream buffers (if present) must be printed out from most-recently-used to least-recently-used, and only valid stream buffers should be printed out.
3. The following measurements: (*note that "miss" means neither the cache nor its stream buffers hit*)

a. number of L1 reads
b. number of L1 read misses, *excluding L1 read misses that hit in the stream buffers if L1 prefetch unit is enabled*
c. number of L1 writes
d. number of L1 write misses, *excluding L1 write misses that hit in the stream buffers if L1 prefetch unit is enabled*
e. L1 miss rate = $\boxed{MR_{L1}}$ = (L1 read misses + L1 write misses)/(L1 reads + L1 writes)
f. number of writebacks from L1 to next level
g. number of L1 prefetches (*prefetch requests from L1 to next level, if prefetch unit is enabled*)
h. number of L2 reads that did not originate from L1 prefetches (*should match b+d: L1 read misses + L1 write misses*)
i. number of L2 read misses that did not originate from L1 prefetches, *excluding such L2 read misses that hit in the stream buffers if L2 prefetch unit is enabled*
j. number of L2 reads that originated from L1 prefetches (*should match g: L1 prefetches*)
† SEE IMPORTANT NOTE BELOW.
k. number of L2 read misses that originated from L1 prefetches, *excluding such L2 read misses that hit in the stream buffers if L2 prefetch unit is enabled*
† SEE IMPORTANT NOTE BELOW.
l. number of L2 writes (*should match f: number of writebacks from L1*)
m. number of L2 write misses, *excluding L2 write misses that hit in the stream buffers if L2 prefetch unit is enabled*
n. L2 miss rate (*from standpoint of stalling the CPU*) = $\boxed{MR_{L2}}$ = (item i)/(item h)
o. number of writebacks from L2 to memory
p. number of L2 prefetches (*prefetch requests from L2 to next level, if prefetch unit is enabled*)
q. total memory traffic = number of blocks transferred to/from memory
(*with L2, should match i+k+m+o+p:*
all L2 read misses + L2 write misses + writebacks from L2 + L2 prefetches)
(*without L2, should match b+d+f+g:*
L1 read misses + L1 write misses + writebacks from L1 + L1 prefetches)

† For this project, as shown in Figure 5 for ECE 563 students, prefetching is only tested and explored in the last-level cache of the memory hierarchy. This means that measurements j and k,

above, should always be 0 because the L1 will not issue prefetch requests to the L2. Nonetheless, a well-done implementation of a generic CACHE will distinguish incoming *demand read requests* from incoming *prefetch read requests*, even though in *this* project the distinction will not be exercised.

Note for ECE 463 students: Just assume and print 0 for any prefetch-specific measurement. These are: g, j, k, p.

8. Submit, Validate, and Self-Grade with Gradescope

Sample simulation outputs are provided on the Moodle site. These are called "validation runs". **Refer to the validation runs to see how to format the outputs of your simulator.**

You must submit, validate, and self-grade[2] your project using Gradescope. Here is how Gradescope (1) receives your project (zip file), (2) compiles your simulator (Makefile), and (3) runs and checks your simulator (arguments, print-to-console requirement, and "diff -iw"):

1. **How Gradescope receives your project: zip file.** While you are developing your simulator, you may continuously submit new zip files to Gradescope containing the latest version of your project. The latest submission is the one that is considered for your grade. Gradescope will accept a zip file consisting of three things: your source code, a Makefile to compile your source code, and your project report. In the early stages of your project, before creating the report, your zip file will have only source code and a Makefile. Once the report is completed, your zip file will contain everything.

    1. Report (included in the zip file once available): The report must be a PDF file named "report.pdf" located at the top level of the zip file, because that is what Gradescope looks for when checking completeness of the submission. 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 is available on the Moodle site.
        - See Section 9 for the required content of the report.
    - Makefile: The Makefile must be at the top level of the zip file, because Gradescope runs "make" with the expectation that the Makefile is at the top level.
    - Source code: Whether your source code is at the top level of the zip file or in directories below the top level, your Makefile must be designed to compile your source code, accordingly.

2. **How Gradescope compiles your simulator: Makefile.** Along with your source code, you must provide a Makefile that automatically compiles the simulator. This Makefile must create a simulator named "sim". An example Makefile is posted on the Moodle site, which you can copy and modify for your needs.

3. **How Gradescope runs and checks your simulator: arguments, print-to-console requirement, "diff -iw", and timeout.**

- Your simulator executable (created by your Makefile) must be named "sim" and take command-line arguments in the manner specified in Section 6.2, because Gradescope assumes these things.
- Your simulator must print outputs to the console (*i.e.*, to the screen), because Gradescope assumes this.

[2] The mystery runs component of your grade will not be published until we release it. The report will be manually graded by the TAs.

- Your output must match the validation runs both numerically and in terms of formatting, because Gradescope runs "diff -iw" to compare your output with the correct output. 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. Note, however, that extra or missing blank lines are NOT ok: "diff -iw" does not ignore extra or missing blank lines.
- Gradescope's autograder has a timeout for compiling the simulator and running all tests. The default timeout is 10 minutes. This is usually ample time for this course. If the autograder times out for your project (very inefficient simulator or a bug that causes deadlock), you will see a grade of zero for that submission. Please see Section 12.2 regarding optimizing run time. Also seek advice from the instructor and TAs, as needed.

9. Experiments and Report

Benchmarks

Unless otherwise specified, use the *gcc* benchmark (`gcc_trace.txt`) for all experiments.

Calculating AAT, Area, and Energy

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*
MR_{L1}	L1 miss rate.	From your simulator. See Section 7.
MR_{L2}	L2 miss rate (from standpoint of stalling the CPU).	
HT_{L1}	Hit time of L1.	Refer to the spreadsheet of CACTI results available on the Project-1 website: sheet "CACTI results", column E "Access time (ns)".
HT_{L2}	Hit time of L2.	
Miss_Penalty	Time to fetch one block from main memory.	Refer to the spreadsheet of CACTI results available on the Project-1 website: sheet "Miss_Penalty", cell B1.
A_{L1}	Die area of L1.	Refer to the spreadsheet of CACTI results available on the Project-1 website: sheet "CACTI results", column G "Area (mm*mm)".
A_{L2}	Die area of L2.	
E_{L1}	Dynamic energy per access of L1.	"Energy Per Access (nJ)" from CACTI tool. A spreadsheet of CACTI results is available on the Project-1 website.
E_{L2}	Dynamic energy per access of L2.	
E_{MEM}	Dynamic energy per access of main memory.	Refer to the spreadsheet of CACTI results available on the Project-1 website: sheet "E_MEM".

* We will not be using energy in any of the experiments for the Fall 2022 Project 1, after all. Thus, they are grayed-out in Table 1.

For memory hierarchy *without* L2 cache:

$$\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot HT_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot \text{Miss\_Penalty}$$

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

$$\begin{aligned} AAT &= HT_{L1} + \left(\frac{\text{L1 read misses} + \text{L1 write misses}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \\ &= HT_{L1} + MR_{L1} \cdot \text{Miss\_Penalty} \end{aligned}$$

For memory hierarchy *with* L2 cache:

$$\text{Total access time} = (\text{L1 reads} + \text{L1 writes}) \cdot HT_{L1} + (\text{L1 read misses} + \text{L1 write misses}) \cdot HT_{L2} + (\text{L2 read misses not originating from L1 prefetches}) \cdot \text{Miss\_Penalty}$$

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

$$\begin{aligned} AAT &= HT_{L1} + \left(\frac{\text{L1 read misses} + \text{L1 write misses}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot HT_{L2} + \left(\frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \\ &= HT_{L1} + MR_{L1} \cdot HT_{L2} + \left(\frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \\ &= HT_{L1} + MR_{L1} \cdot \left(HT_{L2} + \left(\frac{1}{MR_{L1}} \right) \cdot \left(\frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot \left(HT_{L2} + \left(\frac{\text{L1 reads} + \text{L1 writes}}{\text{L1 read misses} + \text{L1 write misses}} \right) \cdot \left(\frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 reads} + \text{L1 writes}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot \left(HT_{L2} + \left(\frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L1 read misses} + \text{L1 write misses}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot \left(HT_{L2} + \left(\frac{\text{L2 read misses not originating from L1 prefetches}}{\text{L2 reads not originating from L1 prefetches}} \right) \cdot \text{Miss\_Penalty} \right) \\ &= HT_{L1} + MR_{L1} \cdot (HT_{L2} + MR_{L2} \cdot \text{Miss\_Penalty}) \end{aligned}$$

The total area of the caches:

$$\text{Area} = A_{L1} + A_{L2}$$

If a particular cache does not exist in the memory hierarchy configuration, then its area is 0. Note that it is difficult to estimate the area of the prefetch unit using CACTI due to the specialized structure of the Stream Buffers.

Dynamic energy estimates:
Each read or write request to a cache consumes that cache's access energy. Each read or write request that misses in the cache causes a "line fill" (allocation) into the cache, which also consumes that cache's access energy.[3] Each writeback of an evicted dirty block involves reading that block from the cache, which also consumes that cache's access energy.

For memory hierarchy *without* L2 cache:

Total dynamic energy =
(L1 reads + L1 writes + L1 read misses + L1 write misses + L1 writebacks) * E_{L1} +
(L1 read misses + L1 write misses + L1 writebacks + L1 prefetches) * E_{MEM}

For memory hierarchy *with* L2 cache:

Total dynamic energy =
(L1 reads + L1 writes + L1 read misses + L1 write misses + L1 writebacks) * E_{L1} +
(all L2 reads + L2 writes + all L2 read misses + L2 write misses + L2 writebacks) * E_{L2} +
(all L2 read misses + L2 write misses + L2 writebacks + L2 prefetches) * E_{MEM}

average dynamic energy per access = (total dynamic energy)/(L1 reads + L1 writes)

[3] Note: There is a noticeable underestimate of energy when a request misses in the cache but hits in its stream buffer. We don't count this as a miss (it doesn't get counted as an L1 read miss or L1 write miss) and we don't explicitly count this scenario. Yet, this scenario also involves a "line fill" (allocation) into the cache (transfer block from stream buffer to cache). This can be fixed by explicitly counting this scenario but we shall ignore it in this project.

9.1. L1 cache exploration: SIZE and ASSOC

GRAPH #1 **(*total number of simulations: 55*)**

For this experiment:

- Benchmark trace: gcc_trace.txt
- L1 cache: SIZE is varied, ASSOC is varied, BLOCKSIZE = 32.
- L2 cache: None.
- Prefetching: 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*.

Answer the following questions in your report:

1. For a given associativity, how does increasing cache size affect miss rate?
2. For a given cache size, how does increasing associativity affect miss rate?
3. Estimate the *compulsory miss rate* from the graph and briefly explain how you arrived at this estimate.

GRAPH #2 **(*no additional simulations with respect to GRAPH #1*)**

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

Answer the following question 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 and what is that AAT?

GRAPH #3 **(*total number of simulations: 20*)**

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

- Add the following L2 cache to the memory hierarchy: 16KB, 8-way set-associative, same block size as L1 cache.
- Vary the L1 cache size only between 1KB and 8KB (since L2 cache is 16KB).

Answer the following questions in your report:

1. With the L2 cache added to the system, which L1 cache configuration yields the best (*i.e.*, lowest) AAT and what is that AAT?
2. How does the lowest AAT with L2 cache (GRAPH #3) compare with the lowest AAT without L2 cache (GRAPH #2)?

3. Compare the *total area* required for the lowest-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:

- Benchmark trace: gcc_trace.txt
- L1 cache: SIZE is varied, BLOCKSIZE is varied, ASSOC = 4.
- L2 cache: None.
- Prefetching: 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*.

Answer the following questions in your report:

1. Do smaller caches prefer smaller or larger block sizes?
2. Do larger caches prefer smaller or larger block sizes?
3. As block size is increased from 16 to 128, is the tension between *exploiting more spatial locality* and *cache pollution* evident in the graph? Explain.

9.3. L1 + L2 co-exploration

GRAPH #5 **(*total number of simulations: 12*)**

For this experiment:

- Benchmark trace: gcc_trace.txt
- L1 cache: SIZE is varied, BLOCKSIZE = 32, ASSOC = 4.
- L2 cache: SIZE is varied, BLOCKSIZE = 32, ASSOC = 8.
- Prefetching: None.

Plot AAT on the y-axis versus $\log_2$(L1 SIZE) on the x-axis, for four different L1 cache sizes: L1 SIZE = 1KB, 2KB, 4KB, 8KB. (That is, $\log_2$(L1 SIZE) = 10, 11, 12, 13.) The graph should contain three separate curves (*i.e.*, lines connecting points), one for each of the following L2 cache sizes: 16KB, 32KB, 64KB. All points for the 16KB L2 cache should be connected with a line, all points for the 32KB L2 cache should be connected with a line, *etc*.

Answer the following question in your report:

1. Which memory hierarchy configuration in Graph #5 yields the best (*i.e.*, lowest) AAT and what is that AAT?

9.4. Stream buffers study (ECE 563 students only)

TABLE #1 (*total number of simulations: 5*)

For this experiment:

- Microbenchmark: stream_trace.txt
- L1 cache: SIZE = 1KB, ASSOC = 1, BLOCKSIZE = 16.
- L2 cache: None.
- PREF_N (number of stream buffers): 0 (pref. disabled), 1, 2, 3, 4
- PREF_M (number of blocks in each stream buffer): 4

The trace "stream_trace.txt" was generated from the loads and stores in the loop of interest of the following microbenchmark:

```
#define SIZE 1000

uint32_t a[SIZE];
uint32_t b[SIZE];
uint32_t c[SIZE];

int main(int argc, char *argv[]) {
…
   // LOOP OF INTEREST
   for (int i = 0; i < SIZE; i++)
      c[i] = a[i] + b[i];   // per iteration: 2 loads (a[i], b[i]) and 1 store (c[i] = …)
…
}
```

In your report, fill in the following table and answer the following questions:

PREF_N, PREF_M	L1 miss rate
0,0 (pref. disabled)	
1,4	
2,4	
3,4	
4,4	

1. For this streaming microbenchmark, with prefetching disabled, do L1 cache size and/or associativity affect the L1 miss rate (feel free to simulate L1 configurations besides the one used for the table)? Why or why not?
2. For this streaming microbenchmark, what is the L1 miss rate with prefetching disabled? Why is it that value, *i.e.*, what is causing it to be that value? Hint: each element of arrays a, b, and c, is 4 bytes (uint32_t).
3. For this streaming microbenchmark, with prefetching disabled, what would the L1 miss rate be if you doubled the block size from 16B to 32B? (hypothesize what it will be and then check your hypothesis with a simulation)
4. With prefetching enabled, what is the minimum number of stream buffers required to have any effect on L1 miss rate? What is the effect on L1 miss rate when this many

stream buffers are used: specifically, is it a modest effect or huge effect? Why are this many stream buffers required? Why is using fewer stream buffers futile? Why is using more stream buffers wasteful?

10. 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 563)
Substantial simulator turned in	30 points	30 points

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

Item		Points (ECE 463)	Points (ECE 563)
L1 works	validation run #1	9 points	6 points
	validation run #2	9 points	7 points
	mystery run A	8 points	7 points
L1, L2 works	validation run #3	8 points	6 points
	validation run #4	8 points	7 points
	mystery run B	8 points	7 points
L1+pref. works	validation run #5	*not applicable*	2 points
	validation run #6		2 points
	mystery run C		2 points
L1, L2+pref. works	validation run #7		2 points
	validation run #8		1 points
	mystery run D		1 points

[20 points] Experiments and report.

Item		Points (ECE 463)	Points (ECE 563)
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
	TABLE #1 + disc.	*not applicable*	3 points

Analysis:

463 max points with just L1 working and corresponding graphs+discussion (#1,2,4): 56 (sim.) + 12 (exp.) = 68
563 max points with just L1 working and corresponding graphs+discussion (#1,2,4): 50 (sim.) + 10 (exp.) = 60
563 max points with everything but pref. working, and corr. graphs+discussion (#1-5): 70 (sim.) + 17 (exp.) = 87
563 max points with everything but L2 working, and corr. graphs+discussion (#1,2,4,T1): 56 (sim.) + 13 (exp.) = 69

11. Penalties

Various deductions (out of 100 points):

-1 point for each day (24-hour period) late, according to the Gradescope timestamp. The late penalty is pro-rated on an hourly basis: -1/24 point for each hour late. We will use the "ceiling" function of the lateness time to get to the next higher hour, *e.g.*, ceiling(10 min. late) = 1 hour late, ceiling(1 hr, 10 min. late) = 2 hours late, and so forth. **For this first project, Gradescope will accept late submissions no more than two weeks after the deadline. The goal of this policy is to encourage forward progress for other work in the class.**

See Section 1.1 for penalties and sanctions for academic integrity violations.

12. Advice on backups and run time

12.1. 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 (your AFS or NFS locker in your NCSU account, home PC, laptop … keep consistent copies in multiple places) or removable media (flash drive, *etc*.).

12.2. Run time of simulator

Correctness of your simulator is of paramount importance. That said, making your simulator *efficient* is also important because 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 Moodle site 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.

As mentioned in Section 8, another reason for being wary of excessive run times is Gradescope's autograder timeout.