University of Central Florida

Department of Computer Science

CDA 5106: Spring 2021

Machine Problem 1: Cache Design, Memory Hierarchy Design

1. Ground Rules

1. All students must work alone.
2. Sharing of code between students is considered cheating and will receive appropriate action in accordance with University policy. The TAs will scan source code 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.
3. You must do all your work in the C/C++ or Java languages. Exceptions must be pre-approved by the instructor.

2. Machine Problem Description

In this machine problem, 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

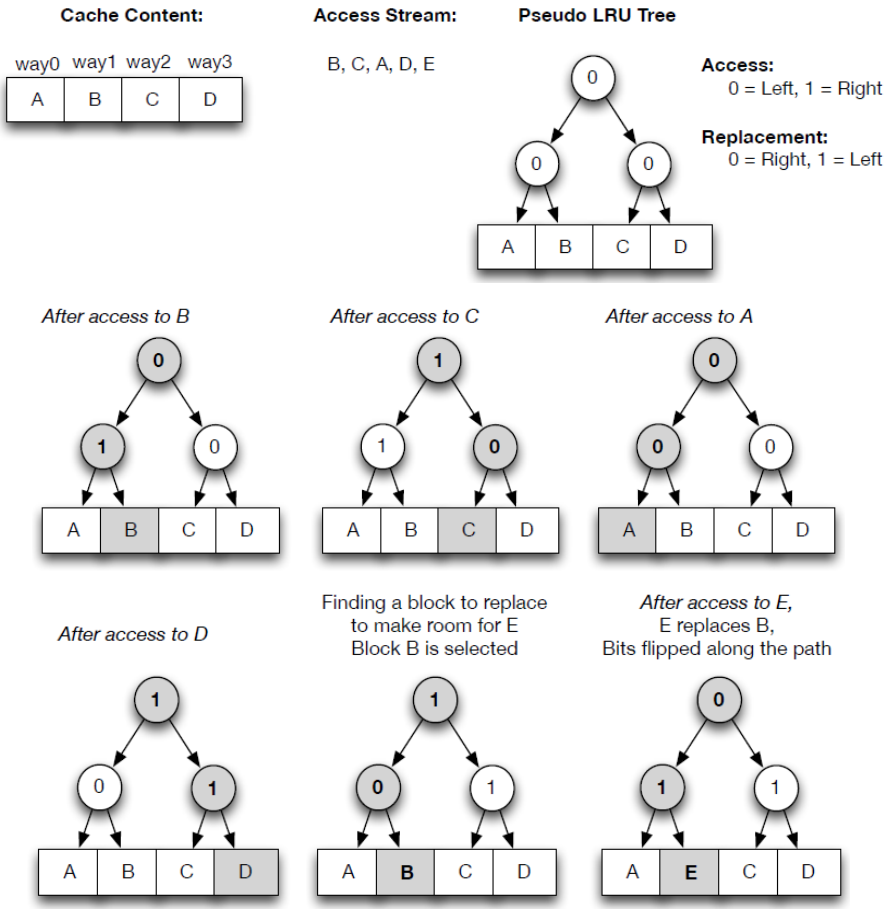
All students need to implement three replacement policies: LRU (least-recently-used), Pseudo-LRU policy and optimal policy. Replacement policy will be a configurable parameter for the CACHE simulator.

3.2.1 LRU policy

Replace the block that was least recently touched (updated on hits and misses).

3.2.2 Pseudo-LRU policy (Tree-PLRU)

Replace the block that was least recently touched by using a binary tree, for example, a 4-way set associative cache needs three bits to keep track of most recently used block. Following is an example with diagrams.



3.2.3 Optimal policy

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.

Note: If there is more than one block (in a set) that's not going to be reused again in the trace, replace the leftmost one that comes up from the search.

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.

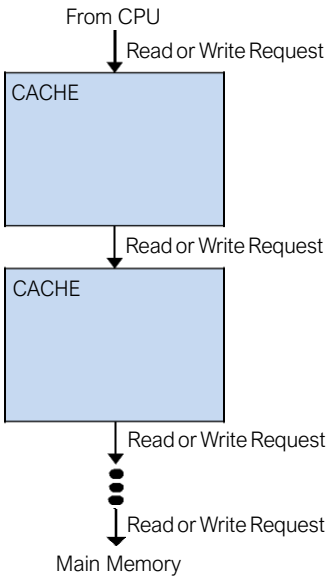


Fig. 1: Your simulator must be capable of modeling one or more instances of CACHE to form an overall memory hierarchy.

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/Pseudo-LRU/optimal counters affiliated with the set as well as the valid and dirty bits affiliated with the requested block.

3.6. Inclusion Property

Now, implement another inclusion property (inclusive property) for CACHE. Inclusion property will be a configurable parameter for the CACHE simulator.

3.6.1 Non-inclusive cache

Non-inclusive property is the default property used in this machine problem. It is simply what you'll get if you follow the directions listed above. There is no enforcement of either the cache inclusion nor the cache exclusion property. A cache block in an inner cache may or may not be in an outer cache.

3.6.2 Inclusive cache

According to the inclusive property, an outer cache should be a superset of all inner caches it surrounds. i.e. any reference in L1 cache must also hit in the L2 cache. For homogeneous caches, such as the ones we shall be testing, the only difference between inclusive and non-inclusive cache is on L2 eviction (happens when read or write request misses at the L2 cache and the requested block needs to be allocated). When a victim block in the L2 cache needs to be evicted, the L2 cache must invalidate the corresponding block in L1 as well (assuming it exists there). If the L1 block that needs to be invalidated is dirty, a write of the block will be issued to the main memory directly.

4. Memory Hierarchies to be Explored in this Machine Problem

While Fig. 1 illustrates an arbitrary memory hierarchy, you will only need to study the memory hierarchy configurations shown in Fig. 2a and Fig. 2b. Also, these are the only configurations the TAs will test.

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

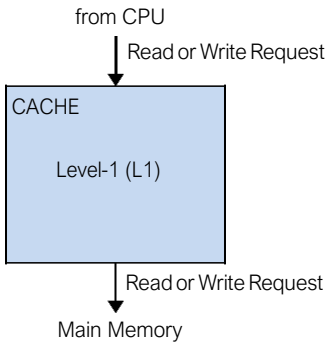


Fig. 2a: Configurations to be studied.

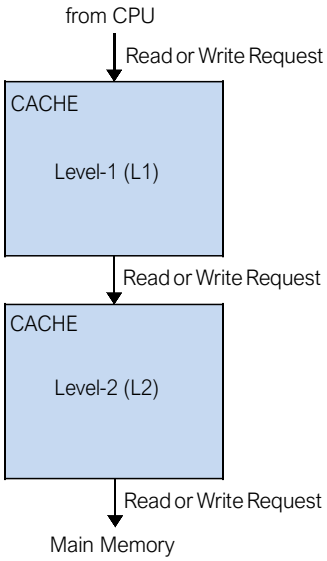


Fig. 2b: Configurations to be studied.

5. 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

...

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. 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. L1 miss rate = MR_{L1} = (L1 read misses + L1 write misses)/(L1 reads + L1 writes)
   f. number of writebacks from L1 to next level
   g. number of L2 reads (*should match b+d: L1 read misses + L1 write misses*)
   h. number of L2 read misses if there is a L2 cache
   i. number of L2 writes (*should match f: number of writebacks from L1 to L2*)
   j. number of L2 write misses
   k. L2 miss rate (from standpoint of stalling the CPU) = MR_{L2} = (item h)/(item g)
   l. number of writebacks from L2 to memory
   m. total memory traffic = number of blocks transferred to/from memory

   (with L2, should match h+j+l for non-inclusive cache:

   all L2 read misses + L2 write misses + writebacks from L2

Note: for inclusive cache, writebacks directly from L1 to memory due to invalidation should also be counted)

(without L2, should match b+d+f:

L1 read misses + L1 write misses + writebacks from L1)

7. Validation and Other Requirements

7.1. Validation requirements

Sample simulation outputs are provided. These are under "validation_runs" folder. You must run your simulator and debug it until it matches these sample outputs.

Each validation run includes:

1. Memory hierarchy configuration and trace filename.
2. The final contents of all caches.
3. All measurements described in Section 6.

Your simulator must print outputs to the console (i.e., to the screen). (Also see Section 7.2 about this requirement.)

Your output must match both numerically and in terms of formatting, because the TAs will literally "diff" your output with the correct output. You must confirm correctness of your simulator by following this step for each validation run:

- Test whether or not your outputs match properly, by running this linux command:

  diff -i -w <*your_output_file*> <*posted_output_file*>

The '-i -w' flags tell "diff" to treat files with case insensitive and 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.

7.2. 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. 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 files and the simulator executable. Example Makefiles are provided on Webcourses, which you can copy and modify for your needs.
2. Your simulator must accept exactly 8 command-line arguments in the following order:

   sim_cache <*BLOCKSIZE*> <*L1_SIZE*> <*L1_ASSOC*> <*L2_SIZE*> <*L2_ASSOC*>
   <*REPLACEMENT_POLICY*> <*INCLUSION_PROPERTY*> <*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).
- ***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).
- ***REPLACEMENT_POLICY:*** Positive integer. 0 for LRU, 1 for PLRU, 2 for Optimal.
- ***INCLUSION_PROPERTY:*** Positive integer. 0 for non-inclusive, 1 for inclusive.
- ***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, LRU replacement, non-inclusive cache (default), gcc trace:

sim_cache 32 8192 4 262144 8 0 0 gcc_trace.txt

3. 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. 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*
MR_{L1}	L1 miss rate.	From your simulator. See Section 6.
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 is posted on Webcourses.
HT_{L2}	Hit time of L2.	
Miss_Penalty	Time to fetch one block from main memory.	100ns for Block Size 32
A_{L1}	Die area of L1.	"Cache height x width (mm)" from CACTI tool.
A_{L2}	Die area of L2.	

For memory hierarchy *without* L2 cache:

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

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

$$AAT = HT_{L1} + \left(\frac{L1\ read\ misses + L1\ write\ misses}{L1\ reads + L1\ writes}\right)\cdot Miss\_Penalty$$

$$= HT_{L1} + MR_{L1}\cdot Miss\_Penalty$$

For memory hierarchy with L2 cache:

$$Total\ access\ time = (L1\ reads + L1\ writes)\cdot HT_{L1} + (L1\ read\ misses + L1\ write\ misses)\cdot HT_{L2}$$
$$+ L2\ read\ miss\ \cdot\ Miss\_Penalty$$

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

$$AAT = HT_{L1} + \left(\frac{L1\ read\ misses + L1\ write\ misses}{L1\ reads + L1\ writes}\right)\cdot HT_{L2}\ +\ \left(\frac{L2\ read\ misses}{L1\ reads + L1\ writes}\right)\cdot Miss\_Penalty$$

$$= HT_{L1} + MR_{L1}\cdot HT_{L2}\ +\ \left(\frac{L2\ read\ misses}{L1\ reads + L1\ writes}\right)\cdot Miss\_Penalty$$

$$= HT_{L1} + MR_{L1}\cdot\left(HT_{L2}\ +\ \left(\frac{1}{MR_{L1}}\right)\left(\frac{L2\ read\ misses}{L1\ reads + L1\ writes}\right)\cdot Miss\_Penalty\right)$$

$$= HT_{L1} + MR_{L1}\cdot\left(HT_{L2}\ +\ \left(\frac{L1\ reads + L1\ writes}{L1\ read\ misses + L1\ write\ misses}\right)\left(\frac{L2\ read\ misses}{L1\ reads + L1\ writes}\right)\cdot Miss\_Penalty\right)$$

$$= HT_{L1} + MR_{L1}\cdot\left(HT_{L2}\ +\ \left(\frac{L2\ read\ misses}{L1\ read\ misses + L1\ write\ misses}\right)\cdot Miss\_Penalty\right)$$

$$= HT_{L1} + MR_{L1}\cdot\left(HT_{L2}\ +\ \left(\frac{L2\ read\ misses}{L2\ reads}\right)\cdot Miss\_Penalty\right)$$

$$= HT_{L1} + MR_{L1}\cdot(HT_{L2}\ +\ MR_{L2}\cdot Miss\_Penalty)$$

The total area of the cache:

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

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

8.1. L1 cache exploration: SIZE and ASSOC

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

For this experiment:

- L1 cache: SIZE is varied, ASSOC is varied, BLOCKSIZE = 32.
- L2 cache: None.
- Replacement policy: LRU
- Inclusion property: non-inclusive

Plot L1 miss rate on the y-axis versus log2(SIZE) on the x-axis, for eleven different cache sizes: SIZE = 1KB, 2KB, …, 1MB, in powers-of-two. (That is, log2(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.

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. Discussion to include in your report:

(**Note:** The value of Access time of 1KB 8-way Cache with block size 32 is not available in Cacti Table, you can ignore this point in your graph.)

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

8.2. Replacement policy study

GRAPH #3 *(total number of simulations: 27)*

For this experiment:

- L1 cache: SIZE is varied, ASSOC = 4, BLOCKSIZE = 32.
- L2 cache: None.
- Replacement policy: varied
- Inclusion property: non-inclusive

Plot AAT on the y-axis versus log2(SIZE) on the x-axis, for nine different cache sizes: SIZE = 1KB, 2KB, … , 256KB, in powers-of-two. (That is, log2(SIZE) = 10, 11, …, 18.) The graph should contain three separate curves (i.e., lines connecting points), one for each of the following replacement policies: **LRU, Pseudo-LRU, Optimal**. All points for LRU replacement policy should be connected with a line, all points for Pseudo-LRU replacement policy should be connected with a line, etc.

Discussion to include in your report:

1. Discuss trends in the graph. Which replacement policy yields the best (i.e., lowest) AAT?

8.3. Inclusion property study

GRAPH #4 *(total number of simulations: 12)*

For this experiment:

- L1 cache: SIZE = 1KB, ASSOC = 4, BLOCKSIZE = 32.
- L2 cache: SIZE = 2KB – 64KB, ASSOC = 8, BLOCKSIZE = 32.
- Replacement policy: LRU
- Inclusion property: varied

Plot AAT on the y-axis versus log2(L2 SIZE) on the x-axis, for six different L2 cache sizes: L2 SIZE = 2KB, 4KB, … , 64KB, in powers-of-two. (That is, log2(L2 SIZE) = 11, 12, …, 16.) The graph should contain two separate curves (i.e., lines connecting points), one for each of the following inclusion properties: non-inclusive and inclusive. All points for non-inclusive cache should be connected with a line, all points for inclusive cache should be connected with a line.

Discussion to include in your report:

1. Discuss trends in the graph. Which inclusion property yields a better (i.e., lower) AAT?

9. What to Submit on Webcourses

You must hand in a single zip file called mp1.zip.

mp1.zip must only contain the following (any deviation from the following requirements may delay grading your project and may result in point deductions, late penalties, etc.):

1. Problem report. This must be a single PDF document named “report.pdf”. The report must include the following:
   - A cover page with the machine problem title, the Honor Pledge, and your full name as electronic signature of the Honor Pledge. A sample cover page is posted on webcourses.
   - See Section 8 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 7.2, item #2, for strict requirements. If you fail to meet these requirements, it may delay grading your project and may result in point deductions.

Note:

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

10. Grading

Table 2 shows the breakdown of points for MP1:

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

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

[20 points] **Experiments and report.**

Table 2. Breakdown of points.

[30 points] Substantial programming effort

Item	Points
Substantial simulator tuned in	30 points

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

Item		Points
L1 with LRU works	Validation run #0	4 points
	Validation run #1	4 points
	Mystery run A	5 points
L1 with Pseudo-LRU, Optimal works	Validation run #2	4 points
	Validation run #3	4 points
	Mystery run B	5 points
L1, L2 works	Validation run #4	4 points
	Validation run #5	4 points
	Mystery run C	4 points
L1, L2, inclusion property works	Validation run #6	4 points
	Validation run #7	4 points
	Mystery run D	4 points

[20 points] Experiments and report

Item		Points
Experiments and Report	GRAPH#1 + disc.	5 points
	GRAPH#2 + disc.	5 points
	GRAPH#3 + disc.	5 points
	GRAPH#4 + disc.	5 points

11. Submission Checklist

Machine Problem 1 Submission Checklist

Rule	Spec. Section	Did you follow this rule? (Yes/No)
The output from your simulator includes the memory hierarchy configuration and trace filename, the final contents of all caches, and all required measurements.	7.1	
The output from your simulator matches EXACTLY both the formatting and numerical values of the posted validation runs. The only exception is that you are permitted a variable amount of whitespace (tabs and spaces), although newlines must match (i.e., no extra lines).	7.1	
You explicitly confirmed that your outputs match EXACTLY (except for tabs and spaces) the posted validation runs, by running the command "diff -i -w …" as explained in the spec.	7.1	
You can just type "make" to build your simulator (via your Makefile)	7.2	
The simulator built by the Makefile actually runs correctly (some students introduce the Makefile at the last minute and don't actually test the simulator version built by it – don't make this mistake)	TA experience	
The compiled simulator is called "sim_cache"	7.2	
Your simulator "sim_cache" takes in arguments as specified in the spec: sim_cache <BLOCKSIZE> <L1_SIZE> <L1_ASSOC> <L2_SIZE> <L2_ASSOC> <REPLACEMENT_POLICY> <INCLUSION_PROPERTY> <trace_file>	7.2	
Your simulator prints output to the console (i.e., screen) only	7.2	
You have been vigilant about taking certain precautions: keeping backups, not touching files after the submission deadline (to keep timestamps valid, just in case we need to revisit the original files), etc.	7.3	
Submit only a single zip file	9	
The name of your submitted zip file is "mp1.zip"	9	
The size of your submitted zip file is less than 1MB (if you need to go over by a little bit, keep it reasonable.	9	
Include all the necessary files in your mp1.zip submission: *****Source code files *****Makefile *****Report called "report.pdf"	9	
Report is named "report.pdf" (PDF)	9	
Report consists of the provided cover page (with your name, signed Honor Pledge, etc.) and required content as explained in the spec.	8,9	