CS {4/6}290 & ECE {4/6}100 - Spring 2023
Project 1: Cache Simulator
Version 1.2

Professor Tom Conte

Due: February 16th 2023 @ 11:55 PM

Changelog

- **Version 1.2**, 2023-02-06: Fixing some words to reflect the vocabulary used in class. Adding test cases for gcc in checkpoint 1 for set associative and fully associative caches. Changes Highlighted
  - Replace PPN with PFN (this does not change the implementation at all)
  - Added ref outs for l1_s_gcc.out and l1_f_gcc.out
  - Added debug outs for l1_s_short_gcc.out and l1_f_short_gcc.out
- **Version 1.1**, 2023-01-29: A student noticed a bug in my implementation that creates substantial changes to Checkpoint 2 reference and debug outs. Changes Highlighted.
  - Added some minor clarifications/updated wording to reduce possible confusions. Checkpoint 1 is unaffected. Clarified that assumptions are made for the project, not the course as a whole
  - Complete rewrite of section 2.4 for clarity and motivation.

Rules

.

- **This is an individual assignment. ABSOLUTELY NO COLLABORATION IS PERMITTED.** All cases of honor code violations will be reported to the Dean of Students. See Appendix A for more details.
- The due date at the top of the assignment is final. Late assignments will not be accepted.
- Please use office hours for getting your questions answered. If you are unable to make it to office hours, please email the TAs.
- This is a tough assignment that requires a good understanding of concepts before you can start writing code. **Make sure to start early.**
- Read the entire document before starting. Critical pieces of information and hints might be provided along the way.
- Unfortunately, experience has shown that there is a high chance that errors in the project description will be found and corrected after its release. **It is your responsibility to check for updates on Canvas, and download updates if any.**
- Make sure that all your code is written according to **C99 or C++11** standards, using only the standard libraries

1 Checkpoints

This project will have you build a simulator for an L1 cache. You will be implementing this simulator in 2 different checkpoints. Checkpoint 1 is a simple L1 cache assuming the memory access trace has physical addresses, meaning your cache will use physical indices and physical tags. Checkpoint 2 will have you implement a simple Translation Lookaside Buffer (TLB) and Hardware Inverted Page Table (HWIVPT) that will enable you to interpret the addresses as virtual addresses and build a virtually indexed, physically tagged cache.

1.1 Checkpoint 1: Physically-Indexed Physically-Tagged L1 Cache

You will implement an L1 Cache, where you are provided C, B, and S as defined in class. This cache uses an LRU replacement policy and a Write-Back Write-Allocate write policy.

This configuration assumes the memory address trace consists of physical addresses. This means the cache is "Physically-Indexed Physically-Tagged" (PIPT).

You will need to implement the following functions:

- sim_setup: This is where you will configure and instantiate all of the structures you need to simulate the cache.
- sim_access: This will be called once for every access in the trace. This function takes in the address of the access and whether or not the access is a read or write. You will need to update the simulator statistics every time this function is invoked
- sim_finish: This will be called once the trace has been completely simulated. Here, you will need to calculate all of the final statistics like AAT, etc. You also need to deallocate the structures you used to simulate the cache.

1.2 Checkpoint 2: Virtually-Indexed Physically-Tagged L1 Cache with TLB and HW Inverted Page Table

You will extend your implementation for checkpoint 1 to support a Virtually-Indexed Physically-Tagged (VIPT) Cache. You will be provided the additional parameters for P (Page Size), T (TLB entries), and M (Number of Pages in Memory). Due to the way virtually indexed caches work, the index bits and block offset bits must fit in the page offset of a virtual address. This means that not all values of S are valid, you will implement a function to support checking and legalizing the input parameters. You are also given a flag for whether or not you are simulating a VIPT cache. Use this to make sure you don't break the Checkpoint 1 functionality.
You will need to implement/extend the following functions without breaking the checkpoint 1 functionality:

- legalize_s: If the configured value for S does not allow the index to fit within the page offset, increase S to the minimum value to support VIPT.
- sim_setup: Configure and instantiate all of the additional structures you need to simulate the TLB and HWIVPT.
- sim_access: Use the VIPT flag to simulate the cache, TLB, and HWIVPT without breaking the checkpoint 1 functionality.

- sim_finish: You also need to calculate additional statistics, and de-allocate any additional structures you needed for the TLB/HWIVPT simulation.

2 Background

Some course topics and new topics/context that will be important for this project.

2.1 LRU Replacement Policy

The LRU Replacement Policy is the most popular replacement policy and is widely used in caches today. A replacement policy tells you 2 things: how to choose the block to evict from the cache, and where to insert the new block in the cache. The LRU Replacement policy can thus be broken down into 2 pieces, an LRU eviction policy and an MRU insertion policy as defined in the following sections.

2.1.1 LRU Eviction Policy

When selecting a block to remove/evict from the cache, select the block in the appropriate set that was in the Least Recently Used position.

2.1.2 MRU Insertion Policy

When installing/inserting a block into the cache, insert it at the Most Recently Used position.

2.2 Write-Back Write-Allocate Write Policy

This policy describes the behavior of the cache on a write miss and in general on how to deal with writes issued by the processor.

2.2.1 Write-Allocate

When a write is issued by the CPU and the appropriate block is not in the cache, we fetch the block from memory and install it into the cache.

2.2.2 Write-Back

When a write is issued by the CPU, we update the data **only** in the cache. When a block is evicted from the cache, we need to update memory if the block is "dirty" (contains different/updated data compared to memory).

2.3 Virtual Memory

Take the following with a grain of salt as only the relevant portions of Virtual Memory will be explained here.
Programs often require more memory than is physically available on the system. Operating systems folks worked with architects to design a way to "emulate" or **virtualize** memory in such a way that a program thinks it has access to a larger region of memory than is physically available. The granulation to which virtualization is done is called a page. On typical systems, this is 4096 bytes (4 KBytes). There are usually more virtual pages of memory than there is

room for in physical memory so the OS swaps pages from memory to disk as needed by the programs. Programs operate in their virtual address space and these are translated to physical addresses by the hardware based on information configured by the operating system. Assume the virtual addresses are 64 bits and the physical addresses are $M + P$ bits in length. For the purpose of this project, there are 3 layers to the translation effort, the Translation Lookaside Buffer (TLB), the Hardware Inverted Page Table (HWIVPT), and the Dynamic Paging Daemon.

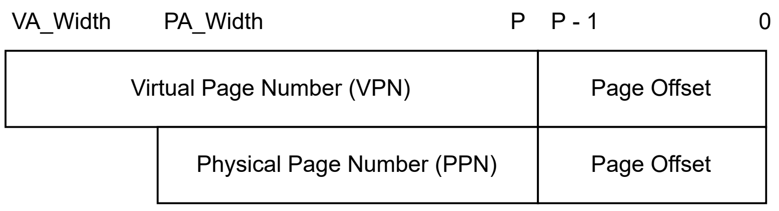

Figure 1: Anatomy of Virtual and Physical Addresses

2.3.1 Translation Lookaside Buffer (TLB)

The TLB is a small fully associative cache that maps the virtual page number (VPN) of an address to the physical frame number (PFN) of the associated physical address. Composing this with the page offset allows us to generate the physical address in memory that corresponds to the virtual address. The TLB holds fewer translations than there are pages/frames in memory. A miss in the TLB is called a Translation Fault, and invokes the HWIVPT to attempt to resolve the translation. The TLB uses an LRU replacement policy.

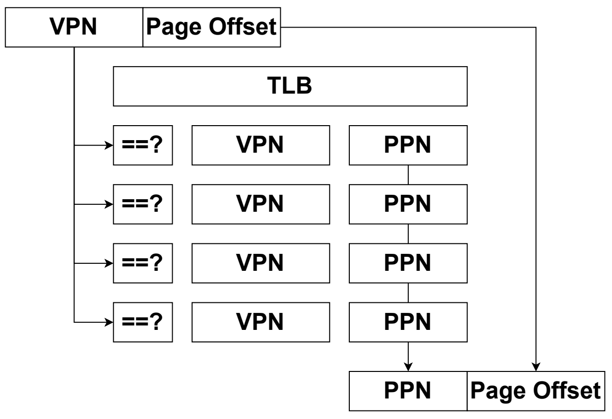

Figure 2: How a TLB replaces the VPN bits with the PFN bits using a fully associative structure.

2.4 Hardware Inverted Page Table

This section contains simplifications that are required for the project to be reasonable but are not necessarily true in the real world.

There are generally more virtual pages than there are physical frames. Because of this, the operating system needs to keep some pages in memory and some on disk. 2 structures exist to help the OS keep track of these pages, the page table and the inverted page table. A page table is an Operating System level structure that stores all currently valid VPN -> PFN mappings with an O(1) access time. An inverted page table is another OS level structure that stores all currently valid PFN -> VPN mappings with an O(1) access time. In the event of a TLB miss, the hardware has a mechanism for traversing these structures and finding the necessary translation for pages that are currently mapped to physical memory. In the event that it doesn't find a translation, a page fault is incurred and the operating system picks the LRU page in memory, copies it to disk, then copies the correct page from disk into the location in memory. This is known as a "page swap". It then updates the tables to keep them up to date with the new mappings. This is quite an expensive OS level function and results in substantial cache pollution.

For the purposes of this project, there is a fully associative Hardware Inverted Page Table (HWIVPT) that stores a bi-directional mapping for all currently valid VPN <-> PFN mappings. It is managed by the operating system. All user-code triggered cache evictions and memory fetches update the LRU status of the translations in the HWIVPT. When a VPN misses in the TLB, we search the HWIVPT for a match/mapping. If a valid mapping exists, from the VPN to the PFN, then we know that the page is currently mapped in memory and we can install the translation into the TLB at the MRU position. In the event that the table does not have a valid mapping, the hardware triggers a pagefault in the Dynamic Paging Daemon. The pagefault handler's cache pollution effect is modeled by flushing the entire L1 cache. During the cache flush time the HWIVPT is locked and the LRU status of the mappings is not updated. By the time the handler returns to the user program, the set buffer will have been reset. However, the necessary array lookup (for populating the set buffer) is done outside the critical path and will not be counted in your statistics.

2.4.1 Dynamic Paging Daemon

The Page Fault Handler in the Dynamic Paging Daemon will remove the LRU page (from the HWIVPT) from memory and replace it with the appropriate page. (you only need to update the translation stored in the HWIVPT, you do not actually need to model the page swap).
The Page Fault Handler in the operating system is substantial enough in length that for the purposes of this project, we will treat this event as completely flushing the L1 cache, writing back any dirty data and invalidating every cache entry. The TLB and HWIVPT will remain untouched aside from the installation of the new translation.

All user-code induced reads and writes that go to memory will update the LRU status of the associated block in the HWIVPT. When a page fault is incurred, the Tag Comparison for the current array lookup is abandoned. Since the cache is flushed by the pagefault, a cache miss is incurred. Please note that this project will use user-level time for AAT so we do not model the latency of the page fault.

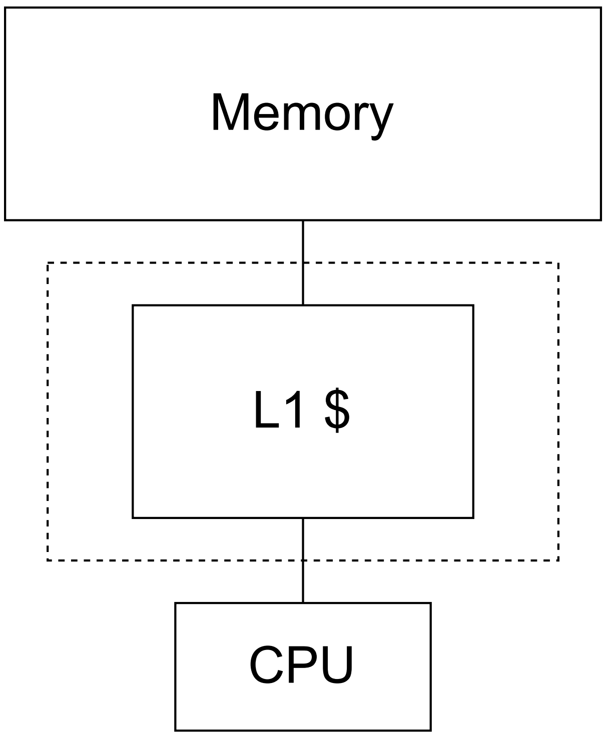

Figure 3: You will implement the region surrounded by the dotted lines

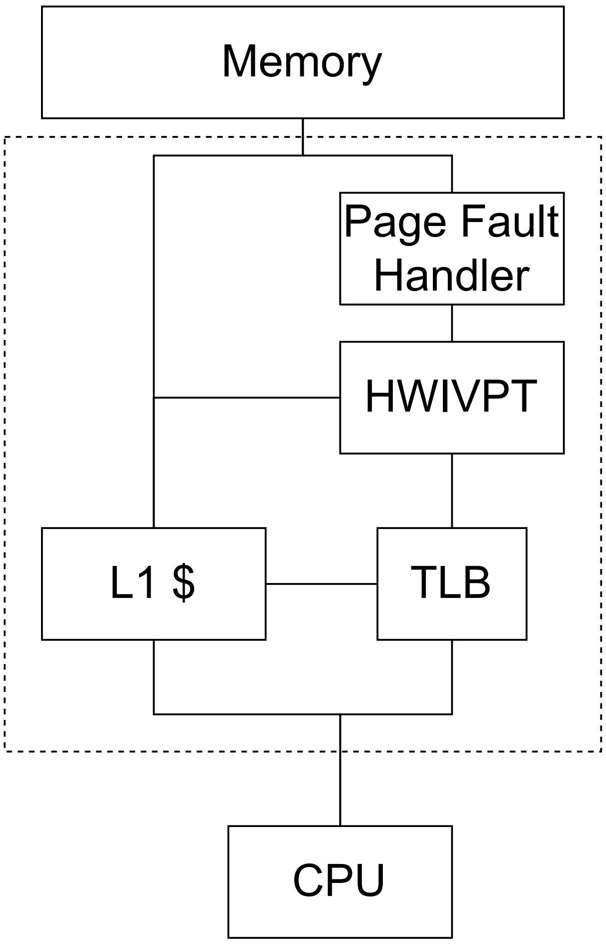

Figure 4: You will implement the region surrounded by the dotted lines

3 Simulator Specifications

In checkpoint 1, you will implement the hierarchy in Figure 3. This means the Write-Back, Write-Allocate, Physically-Indexed, Physically-Tagged L1 cache.

In checkpoint 2, you will implement the hierarchy in Figure 4. This means the Write-Back, Write-Allocate, Virtually-Indexed, Physically-Tagged L1 cache, the TLB, the HWIVPT, and the emulated Dynamic Paging Daemon behavior. The emulated behavior is not as complex as an actual page fault in an Operating System.

The driver will configure your simulator and invoke it for every access in the trace. Then it will prompt the simulator to calculate final statistics.

3.1 Simulator Configuration

The L1 cache, TLB, and HWIVPT follow an LRU replacement policy (LRU eviction + MRU insertion). They all follow Write-Back, Write-Allocate where applicable.

The TLB and HWIVPT are always fully associative.

The simulator will have the following configuration knobs:

- C: A total size of 2^C bytes

  Restriction: The cache size must be reasonable: $9 \leq C \leq 18$
- B: Block size of 2^B bytes.

  Restriction: The block size must be reasonable: $4 \leq B \leq 7$
- S: 2^S-way associativity (may be legalized when using VIPT)

The following knobs control the Virtual Memory related parts of the simulator.

- V: If specified, use VIPT (checkpoint 2). Else use PIPT
- P: The size of a page is 2^P bytes

  Restriction: The page size must be reasonable: $9 \leq P \leq min(14, C)$
- T: 2^T entries in TLB

  Restriction: The TLB must have a reasonable number of entries: $0 \leq T \leq (C - B - S)$
- M: 2^M pages in Memory

  Restriction: Number of pages must be reasonable: $P \leq M \leq 20$

  Restriction: total memory size must be reasonable (4GB): $0 \leq M \leq 32 - P$

3.2 Simulator Semantics

3.2.1 Serial Operation of Parallel Ideas

Although there may be plenty of parallelism in a real-life cache hierarchy, you will implement memory accesses in the simulator serially because it helps calculate the statistics we care about in this project. The sequence of checks for a given memory access should be:

1. L1 array access into set buffer
2. TLB check
3. L1 tag comparison/way select/etc (if check matches)
4. Check HWIVPT (if necessary)
5. Do Page replacement/eviction if necessary (if necessary)

For example, the L1 array access and the TLB check happen in parallel, we can't really simulate this easily without getting too technical.

3.3 Simulator Statistics

Your simulator will calculate and output the following statistics:
Recall that we do not count the time spent in the page-fault handler. (Otherwise we would need substantially longer traces to amortize the cost)

- Overall statistics:
  - Reads (`reads`): Total number of cache reads
  - Writes (`writes`): Total number of cache writes
- L1 statistics:
  - L1 accesses (`accesses_l1`): Number of times the L1 cache was accessed
  - L1 array lookups (`array_lookups_l1`): Number of times an array lookup populates the set buffer
  - L1 tag compares (`tag_compares_l1`): Number of times a TLB hit occurred and necessitated a tag comparison
  - L1 hits (`hits_l1`): Total number of L1 hits
  - L1 misses (`misses_l1`): Total number of L1 misses
  - L1 hit ratio (`hit_ratio_l1`): Hit ratio for L1, calculated using the first three L1 stats above
  - L1 miss ratio (`miss_ratio_l1`): Miss ratio for L1, calculated using the first three L1 stats above
  - AAT (`avg_access_time_l1`): See Section 3.3.1
- Virtual Memory Statistics:
  - TLB accesses (`accesses_tlb`): Number of times the TLB was checked
  - TLB hits (`hits_tlb`): Total number of TLB hits
  - TLB misses (`misses_tlb`): Total number of TLB misses (translation fault)

  - TLB hit ratio (`hit_ratio_tlb`): Hit ratio for TLB, calculated using the first three TLB stats above
  - TLB miss ratio (`miss_ratio_tlb`): Miss ratio for TLB, calculated using the first three TLB stats above
  - HWIVPT accesses (`accesses_hw_ivpt`): Number of times the HWIVPT was checked
  - HWIVPT hits (`hits_hw_ivpt`): Total number of HWIVPT hits after TLB miss
  - HWIVPT misses (`misses_hw_ivpt`): Total number of HWIVPT misses after TLB miss (page fault)
  - HWIVPT hit ratio (`hit_ratio_hw_ivpt`): Hit ratio for HWIVPT, calculated using the first three HWIVPT stats above
  - HWIVPT miss ratio (`miss_ratio_hw_ivpt`): Miss ratio for HWIVPT, calculated using the first three HWIVPT stats above
  - writebacks due to cache flush (`cache_flush_writebacks`): writebacks that occur due to the cache flush on page fault

3.3.1 Average Access Time

For the purposes of average access time (AAT) calculation, we assume that:

- For PIPT L1, hit time is $Array\_Time + Tag\_Compare\_Base\_Time + (S \times 0.2\,\text{ns})$
- The time to access DRAM is 100 ns

For VIPT, The TLB miss path sits in the way of the tag compare and causes substantial changes to the Hit-Time computation. Use the following formulas for calculating hit time.

- $HWIVPT\_PENALTY = (1 + HWIVPT\_ACCESS\_TIME\_PER\_M * M) * DRAM\_ACCESS\_PENALTY$
- $tag\_compare\_time = Tag\_Compare\_Base\_Time + (S \times 0.2\,\text{ns})$
- $HT = L1\_ARRAY\_LOOKUP\_TIME\_CONST + hit\_ratio\_tlb * tag\_compare\_time + miss\_ratio\_tlb * (HWIVPT\_PENALTY + tag\_compare\_time * hit\_ratio\_hw\_ivpt)$

The provided code includes constants for the values that are not already included in the statistics.

When computing the AAT, use the following equation from the lecture slides:

$$\text{AAT} = \text{HT} + \text{MR} \times \text{MP}$$

4 Implementation Details

We included a driver, `cachesim_driver.cpp`, which parses arguments, reads a trace from standard input, and invokes your `sim_access()` function in `cachesim.cpp` for each memory access in the trace. The driver takes a flag for each of the knobs described in Section 3.1; run `./cachesim -h` to see the list of options.

The provided `cachesim.cpp` template file also contains `legalize_s()`, `sim_setup()`, and `sim_finish()` functions you should complete for simulator setup and cleanup (including final stats calculations). By default, the provided `./run.sh` script invokes the `./cachesim` binary produced by linking `cachesim_driver.cpp` with `cachesim.cpp`.

You are discouraged from modifying the `Makefile`, `./run.sh`, `cachesim_driver.cpp`, or the header file `cachesim.hpp`, because it will make it much more difficult for TAs to help with debugging. However, it is allowed as long as `make && ./run.sh <cachesim args> < traces/X.trace` (equivalent to `make && ./cachesim <cachesim args> < traces/X.trace` if you have not changed `./run.sh`) produces byte-for-byte matching outputs as the solution outputs.

4.1 LRU Implementation Recommendations

LRU can be implemented with timestamps, but you may run into bugs/edge cases and is not recommended.

Instead, we suggest using either a C++ `list` or a doubly-linked list to implement LRU. For example, the head can represent MRU and the tail can represent LRU. If a hit occurs, the element can be moved to the MRU position.

Using an array or timestamps to emulate the LRU behavior may cause your runtime to grow substantially.

4.2 Docker Image

We have provided an Ubuntu 22.04 LTS Docker image for verifying that your code compiles and runs in the environment we expect — it is also useful if you do not have a Linux machine handy. To use it, install Docker (`https://docs.docker.com/get-docker/`) and extract the provided project tarball and run the `6290docker*` script in the project tarball corresponding to your OS. That should open an 22.04 `bash` shell where the project folder is mounted as a volume at the current directory in the container, allowing you to run whatever commands you want, such as `make`, `gdb`, `valgrind`, `./cachesim`, etc.

> **Note**: Using this Docker image is not required if you have a Linux system available and just want to make sure your code compiles on 22.04. We will set up a Gradescope autograder that automatically verifies we can successfully compile your submission.

5 Validation Requirements

We have provided six memory access traces in the `traces/` directory of the provided project tarball. Validation outputs for the default simulator configuration (both with and without the VIPT flag) for each of these traces are provided in the `ref_outs/` directory. Given these configurations, the output of your code must match these reference outputs byte-for-byte! We have included a script to compare the output of your simulator to these reference outputs using `make validate`.

We would advise against modifying `cachesim_driver`, which prints the final statistics, unless you are confident it will not cause differences from the solution output. We may grade by testing against other configurations or other traces.

Validation is the goal of the assignment. Submissions that partially match validation will not receive partial credit.

5.1 Debug Traces

Debugging this project can be difficult, particularly when looking only at final statistics, so we have also provided some verbose debug outputs corresponding to segments of the reference traces in `debug_outs`, they correspond to the traces found in `short_traces`
Please see the `README.txt` in `debug_outs` for more information.
You are not required to output or match these debug traces at all — they are only intended to be a debugging aid if needed.

6 Experiments

Once you have your simulator working, you should find the best VIPT cache configuration for each of the six traces that meets the following constraints:

1. Configuration should respect all the restrictions mentioned in Section 3.1
2. The budget for the L1 data store is 32 KiB

The best cache configuration would have one of the lowest average access times, while minimizing data store, metadata/tag storage, TLB size, and HWIVPT/Memory size. Identify points of diminishing returns in terms of cache parameters (C, B, S) and Virtual Memory Parameters (P, T, M). Consider that highly associative caches use substantially more area and power. Include any rationale, quantitative or qualitative, to justify your decisions.

In your report you must provide, in addition to the cache configuration and data store size, the total size of any associated metadata required to build the cache for your recommended configuration(s), the total size of the TLB, and the total size of the HWIVPT and Memory. For the L1 cache, the data storage size is 2^C **bytes**. The tag storage size is $2^{C-B} \times (PA-(C-S)+2)$ **bits**, where PA refers to the size of physical addresses and $PA = P + M$. You will need to come up with and explain the formulas for the TLB and HWIVPT sizes. Assume that maintaining LRU information has no implementation cost for the simplicity of the project.

You should submit a report in PDF format (inside your submission tarball) in which you describe your methodology, rationale, and results.

Feel free to use "`make FAST=1`" to enable compiler optimizations that will help your code run faster if you choose to search a subset of the space of valid configurations.

7 What to Submit to Gradescope

Please submit your project to Gradescope as a tarball containing the your experiments writeup as a PDF, as well as everything needed to build your project: the `Makefile`, the `run.sh` script, and all code files (including provided code). You can use the `make submit` target in the provided Makefile to generate your tarball for you. Please extract your submission tarball and check it before submitting!

We plan to enable a Gradescope autograder that will verify that your code compiles on 22.04 at submission time. The Gradescope autograder result is NOT indicative of your grade on the project and will only check that the assignment compiles.

8 Grading

You will be evaluated on the following criterion:

+0: You don't turn in anything (significant) by the deadline

+50: You turn in well commented significant code that compiles and runs but does **not** match the validation

+10: Your simulator **completely matches** the validation output for Checkpoint 1 configurations

+15: Your simulator **completely matches** the validation output for Checkpoint 2 configurations.

+20: You ran experiments and found the optimum configuration for the 'experiments' workload and presented sufficient evidence and reasoning

+5: Your code is well formatted, commented and does not have any memory leaks! Check out the section on helpful tools

Appendix A - Plagiarism

We take academic plagiarism very seriously in this course. Any and all cases of plagiarism are reported to the Dean of Students. You may not do the following in addition to the Georgia Tech Honor Code:

- Copy/share code from/with your fellow classmates or from people who might have taken this course in prior semesters.
- Look up solutions online. Trust us, we will know if you copy from online sources.
- Debug other people's code. You can ask for help with using debugging tools (Example: Hey XYZ, could you please show me how GDB works), but you may not ask or give help for debugging the cache simulator.
- You may not reuse any code from earlier courses even if you wrote it yourself. This means that you cannot reuse code that you might have written for this class if you had taken it in a prior semester. You must write all the code yourself and during this semester.

Academic Misconduct Disclaimer
(From the syllabus)

Preamble: *The goal of all assignments CS 4290/6290 and ECE 4100/6100 is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.*

1. **As a Georgia Tech student, you have read and agreed to the Georgia Tech Honor Code.** The Honor Code defines Academic Misconduct as "any act that does or could improperly distort Student grades or other Student academic records."
2. You must submit an assignment or project as your own work. **Absolutely no collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such behavior is Academic Misconduct.**
3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "submission of material that is wholly or substantially identical to that created or published by another person").
4. Publishing your assignments on public repositories accessible to other students is unauthorized collaboration and thus Academic Misconduct.
5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.
6. Students suspected of Academic Misconduct will be informed at the end of the semester: you will not know you were found cheating until the end of the class. Suspects receive an Incomplete final grade until the issue is resolved.
7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.
8. **If you are not sure about any aspect of this policy, please ask Dr. Conte**

Appendix B - Helpful Tools

You might the following tools helpful:

- gdb: The GNU debugger will prove invaluable when you eventually run into that segfault. The Makefile provided to you enables the debug flag which generates the required symbol table for gdb by default.
  - To pass a trace on standard in (as cachesim expects) while running in gdb, you can invoke gdb with `gdb ./cachesim` and then run `run <cachesim args> < traces/mcf.trace` at the gdb prompt
- Valgrind: Valgrind is really useful for detecting memory leaks. Use the following command to track all leaks and errors:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes \
    ./cachesim <cachesim args> < traces/mcf.trace
```