CS {4/6}290 & ECE {4/6}100 - Spring 2022
Project 3: Tomasulo Simulator

Dr. Thomas Conte

Due: April 1st 2022 @ 11:55 PM

Version: 1.2.1

Changelog

1. Version 1.2.1 (2022-03-22): Fixed a mistake in section 7.1 where the wrong benchmarks were indicated for including L=2 in the search space. Changes highlighted.
2. Version 1.2 (2022-03-21): Reduced the size of the search space for experiments. Provided guidelines for reducing the search space in a reasonable manner. Changes highlighted.
3. Version 1.1 (2022-03-15): Clarify some confusing phrasing in the PDF and `procsim.hpp` about `rob_stall_cycles` and our modified TSO. Changes highlighted.
4. Version 1.0 (2022-03-13): Initial release

1 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.
- Please see the syllabus for the late policy for this course.
- 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++17** standards, using only the standard libraries[1].

[1] If you choose to write your project from scratch in Java (highly discouraged), please use Java 11 and only the standard library for Java 11. Rust implementations (also discouraged) should be in Rust 1.57.0 and not depend on other cargo packages.

2 Introduction

In this project, you will implement a simulator for a CPU using tagged Tomasulo. The CPU has a write-through write-no-allocate direct-mapped data cache and a store buffer, and it ensures memory consistency using a modified version of TSO (Total Store Ordering). It can also handle precise interrupts. Please read the following sections carefully for simulator details.

Note: This project MUST be done individually. Please follow all the rules from Section 1 and review Appendix A.

3 Simulator Specifications

3.1 Basic Out of Order Architecture

Your simulator will implement the stages of an out-of-order (FOCO) processor using Tomasulo's Algorithm with tags, and support for periodic precise interrupts. Your simulator will take in a handful of parameters that are used to define the processor configuration. The provided code will call your per-stage simulator functions in reverse order to emulate pipelined behavior. Figure 1 is a diagram of the general architecture of the model you will simulate. All edges passing through a dotted line will write to their relevant buffers/latches on the clock edge.

3.2 Simulator Parameters

The following command line parameters can be passed to the simulator:

- `-C`: The size of the direct mapped data cache (C from the (C, B, S) nomenclature defined in class). B is fixed at 6 and S is fixed at 0
- `-A`: The number of ALU units in the Execute stage
- `-M`: The number of Multiply (MUL) units in the Execute stage
- `-L`: The number of Load/Store (LSU) units in the Execute stage
- `-S`: The number of reservation stations **per** function unit. (You will model a unified scheduling queue with $S * (A + M + L)$ reservation stations)
- `-D`: This flag has no argument; if it is passed, periodic interrupts are disabled
- `-F`: The "Fetch Width" is the number of instructions fetched/added to the dispatch queue per cycle
- `-R`: The number of entries in the Re-Order Buffer (ROB)
- `-W`: The retire bandwidth: the maximum number of instructions that can exit the ROB (retire) in a given cycle
- `-I`: The input trace file
- `-H`: Print the usage information

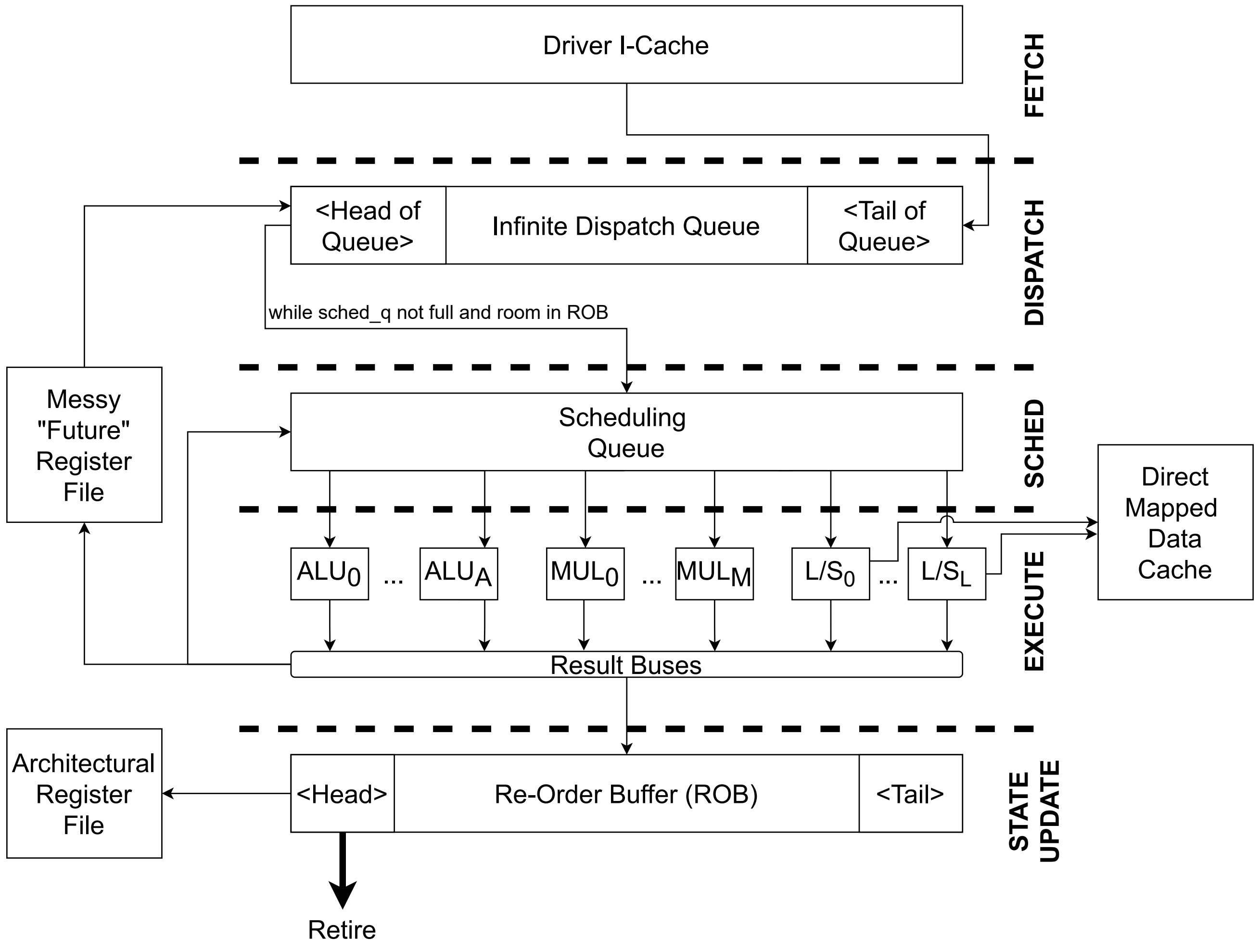

Figure 1: Overview of the Out of Order Architecture. An arrow that crosses a dotted line indicates that the value is latched at the end of a cycle. An arrow that does not cross a dotted line indicates that the associated action occurs within the clock cycle

.

3.3 Trace Format

Each line in a trace represents the following:
`<Address> <Opcode> <Dest Reg #> <Src1 Reg #> <Src2 Reg #> <LD/ST Addr>`
That is, the instructions are effectively already decoded for you. Each register number is in the range $[0, 31]$, except when a register is -1, which indicates (for example) there is no destination register. The opcodes map to the following operations and Function Units:

Opcode	Operation	Function Unit
2	Add	ALU unit
3	Multiply	MUL unit
4	Load	LSU unit
5	Store	LSU unit
6	Branch	ALU unit

3.4 The Data-Cache

You will emulate a direct-mapped write-through, write-no-allocate data cache (dcache) with `B = 6` and `C` from the parameters. The hit time for this cache is 1 cycle, the miss penalty is fixed at 9 cycles (`L2_LATENCY_CYCLES`) due to a 100% accurate L2 cache that you will not model in this project.

This dcache will be used by each Load/Store unit (LSU). Because we do not model a store buffer or load/store queue, all stores take a single cycle. All loads will take either 1 cycles or 10 cycles. You will only use the dcache for load instructions.

The cache will always have enough ports for the number of LSUs. Additionally, if a pipeline flush takes place while an instruction is in the LSU (e.g., for an interrupt), the memory request to satisfy the load is abandoned, meaning the cache is not updated.

3.5 Fetch and The Driver

The driver exposes the instruction cache (icache) to you via the function `procsim_driver_read_inst()`. You can assume the icache never misses. Every cycle, you will read `F` instructions from the driver and append them to the infinite-sized dispatch queue. The driver will periodically mark an instruction as `interrupt`; the implications of this will be apparent in the State Update stage.

Assume that there is a 100% accurate branch predictor that results in every instruction fetched by `procsim_driver_read_inst()` being a valid instruction to execute. Despite this, branch instructions will still progress through the pipeline.

3.6 Dispatch

In dispatch we will attempt to fill the reservation stations in the scheduling queue with instructions from the head of the dispatch queue. Remember that instructions are dispatched in program order. Dispatch will use the tags and ready bits from the messy register file to set up the reservation station in the scheduling queue. Dispatch also allocates a new tag for each instruction regardless of whether it has a destination register. This is done to maintain program order in the ROB and across the processor where necessary. You will represent the tag using a `uint64_t` that is strictly increasing in value. (Optionally, to match the debug traces, start these tags at 0.) Within the length of traces we use, you should not overflow this value.

The dispatch stage will allocate room in the ROB for the instruction as it places it into a reservation station. If there is not sufficient room in the ROB, Dispatch will stall.

3.7 Schedule

The schedule stage searches the scheduling queue for instructions that are ready to fire (have all of their operands ready) and fires them. If multiple instructions can be fired at the same time, they are fired in program order (but still within the same cycle). This means that if 2 instructions are both ready for a single function unit, the instruction that comes first in program order would fire first.

3.7.1 Load/Store Disambiguation and Consistency

Because the superscalar processor may have more than one store unit, we need to ensure that any out of order memory transactions do not cause program correctness issues. We include a modified version of the Intel Total Store Ordering (TSO) for this project.

A Load or Store instruction can be fired **if** there is **no** Load or Store instruction in the scheduling queue that **both** has an index conflict in the cache **and** comes earlier in program order (tag order). This allows stores to fire out of program order by ensuring there is no address conflict. Even though your simulator implementation need not modify the cache when executing stores, this check is still necessary because in the system we model, stores modify the cache.

3.8 Execute

In the execute stage you will move every instruction through its associated function unit.

3.8.1 ALU

This is a single-stage unit.

3.8.2 MUL

This is a 3-stage unit that is pipelined, meaning that up to 3 instructions may be worked on at any given time in a single MUL unit.

3.8.3 LSU

The Load/Store unit is a single-stage unit that will access the data cache. If there is a cache hit, the instruction takes 1 cycle to execute in total; if there is a cache miss, the instruction takes 10 cycles to execute in total. Remember that the cache is Direct-Mapped and Write-Through Write-No-Allocate, and we assume a store buffer. This means that for the purposes of this project, stores take 1 cycle to execute and do not interact with the cache.

3.8.4 Result Buses

As an instruction finishes executing in its function unit, it will pass its values to the result buses. These values will be used to update the messy "future" file and update all of the scheduling queue entries. Completed instructions are removed from the scheduling queue. These busses will also place the instruction in the ROB in program (tag) order. All of this happens within the same cycle!

3.9 State Update

State Update is where we will update the architectural state of the processor. Every cycle, you will, in program order, retire up to `W` instructions and commit their destination register values (if present) to the Architectural Register File. (Since we aren't modeling data in our pipeline, you don't actually need to model this! You only need to increment the `arf_writes` statistic.)

If you encounter an interrupting instruction in the ROB, further instructions cannot be retired. All queues and function units must be flushed along with the remainder of the ROB. However, you should **not** flush the dcache. You will "restore" the state of the messy register file by using the architectural register file and marking all registers as ready. Please do not reset the "next tag" counter to zero, especially if you want to match the debug outputs.

We assume flushing is synchronous, so instructions cannot begin to fill the pipeline until the following cycle.

3.10 Interrupts

You will be implementing precise interrupts in the simulator as well. The interrupts have been mentioned throughout the assignment description but for ease of understanding, the impacts of interrupts are discussed here. When an interrupt is encountered and retired, the following events occur in `flush_pipeline()`:

- All ROB entries are discarded.
- All Function Units and their pipelines are flushed and reset, and any in-progress loads that are pending due to a cache miss are abandoned, and their memory requests are not satisfied.
- All scheduling queue entries are discarded.
- Each register in the messy "future" file is reset with new tags and set to ready.
- The dispatch queue entries (instructions) are discarded. (**Do not `free()` any `inst_t*` as they are allocated by the driver. Doing so will likely cause severe issues.**)

Note the dcache is **not** reset/flushed on an interrupt. Also, the pipeline does not begin to refill until the following cycle after an interrupting instruction is retired (we assume a synchronous reset).

3.11 Initial Conditions

All registers in the messy register file are ready.
All function units are empty/ready.
All queues and the ROB are empty.
All dcache blocks are invalid.

4 Statistics

You will need to keep track of the following statistics:

- `cycles`: The number of cycles that have passed since the processor was started. (The provided code increments this statistic for you)
- `interrupts`: The number of interrupts the processor encounters. (The provided code increments this statistic for you)
- `instructions_fetched`: The number of instructions fetched by the fetch stage, regardless of whether or not they retire
- `instructions_retired`: The number of instructions that exit the ROB
- `arf_writes`: The number of retired instructions that write to the architectural register file (ARF).
- `dcache_reads`: number of reads in the L1 cache
- `dcache_read_misses`: number of read misses in the L1 cache
- `no_fire_cycles`: cycles where nothing in the scheduling queue could be fired

- `rob_stall_cycles`: Cycles in which dispatch stops putting instructions into the scheduling queue only because of the constraint on the maximum number of ROB entries
- `dispq_max_usage`: The maximum number of instructions in the dispatch queue during execution. You should update this at the end of `procsim_do_cycle()`
- `schedq_max_usage`: The maximum number of instructions in the scheduling queue during execution. You should update this at the end of `procsim_do_cycle()`
- `rob_max_usage`: The maximum number of instructions in the ROB during execution. You should update this at the end of `procsim_do_cycle()`
- `dispq_avg_usage`: The average number of instructions in the dispatch queue during execution. You should update this at the end of `procsim_do_cycle()`
- `schedq_avg_usage`: The average number of instructions in the scheduling queue during execution. You should update this at the end of `procsim_do_cycle()`
- `rob_avg_usage`: The average number of instructions in the ROB during execution. You should update this at the end of `procsim_do_cycle()`
- `dcache_read_miss_ratio`: The Miss Ratio for the L1 cache
- `dcache_read_aat`: The AAT for the L1 cache. Note that the hit time for the dcache is 1 cycle
- `ipc`: The average number of instruction retired per cycle
- `instructions_in_trace`: The number of instructions in the trace (this is handled by the driver)

The details of the statistics are also available in `procsim.hpp`.

5 Implementation Details

You have been provided with the following files:

- `procsim.cpp` - Your simulator implementation will go here
- `procsim.hpp` - **You do not need to modify this file.** Header file containing definitions shared between your simulator and the driver
- `procsim_driver.hpp` - **You do not need to modify this file.** Provided code that reads a trace, maintains a pointer to the next instruction in the trace, and allows your simulator code to read it via `procsim_driver_read_inst()`. The simulator invokes your `procsim_do_cycle()` function repeatedly until all trace instructions have been retired
- `run.sh`: A shell script that invokes your simulator. You only need to change this if you have made the highly discouraged choice to write your simulator without the provided framework.
- `validate.sh`: A shell script which runs your simulator and compares its outputs with the reference outputs. If you want to test some particular configurations instead of all configurations, you can pass them as arguments, like: `./validate.sh big tiny`. **You do not need to modify this file.**

- `Makefile`: A Makefile that contains the logic needed to compile your code. It has some useful features:
  - `make validate` will compile your code and run `validate.sh`
  - `make submit` will generate a submission tarball for Gradescope
  - `make clean` will clean out all compiled files
  - `make FAST=1` will compile with `-O2` (You should run `make clean` first)
  - `make DEBUG=1` will compile with the preprocessor definition `DEBUG` defined. (You should run `make clean` first.) This will cause code gated with `#ifdef DEBUG ... #endif` to compile, which you may find useful for generating your own debug traces
  - `make PROFILE=1` will compile your code for use with `gprof`. (You should run `make clean` first.) If your simulator is working correctly (namely, it terminates) but runs slowly, this may help diagnose where the bottleneck is. After you compile with this flag and run your simulator as normal, you can run `gprof procsim` to see profiling results
- `traces/`: A directory containing execution traces from real SPEC 2017 programs; their format is detailed above in Section 3.3. The driver will read these for you
- `ref_outs/`: A directory containing the output of the TA simulator(s) for some select configurations (`make validate` will print their configuration flags)

5.1 Provided Framework

The provided `procsim.cpp` includes detailed comments for implementing your simulator functions. Look for "TODO" comments, which indicate places you need to add code. In short, we suggest dividing your simulator logic into the provided (but initially empty) functions representing different pipeline stages. The provided implementation of `procsim_do_cycle()`, which we recommend using, invokes these pipeline stage functions in reverse order to prevent you from needing to manage pipeline buffers by hand.

Additionally, the provided header file `procsim.hpp` contains comments describing the meaning of statistics your simulator needs to collect in the `procsim_stats_t` struct. If you are unsure about the meaning of some statistics, please see Section 4 for more details.

5.2 Implementation in Java or Rust (Strongly Discouraged)

If you make the highly discouraged choice to write your project from scratch in Java 11 or Rust 1.57.0, you need to set up the `Makefile` with a default target that will compile your code with `javac` or `rustc`, respectively. Please also include a `clean` target that will remove any compiled files (e.g., `.class` files). You will also need to set up `./run.sh` to run your Java or Rust simulator, which must accept all the same configuration flags as `procsim_driver.cpp` (`-I`, `-F`, `-C`, etc.). Your code must also parse the provided trace files, or any other trace files of the same format. Your code must depend only on any libraries or functions included in the standard library for Java 11 or Rust 1.57.0.

5.3 Docker Image

We have provided an Ubuntu 18.04 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 18.04 `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`, `./procsim`, etc.

> **Note**: Using this Docker image is not required if you have a Linux system (or environment) available and just want to make sure your code compiles on 18.04. We will set up a Gradescope autograder that automatically verifies we can successfully compile your submission.

6 Validation Requirements

You must run your simulator and debug it until the statistics from your solution **perfectly (100 percent)** match the statistics in the reference outputs for all test configurations. This requirement must be completed before you can proceed to Section 7 (Experiments).

You can run `make validate` to compare your output with the reference outputs. If you want to test only one configuration for all four benchmarks, you can use the `validate.sh` script directly and pass it a configuration name it understands, like `./validate.sh tiny`.

We do not have a hard efficiency requirement in this assignment, but please make sure your simulator finishes a simulation for one of the provided traces in less than a minute or two. (For reference, the TA solution finishes a simulation in a few seconds.) Otherwise, it will be difficult for the TAs to verify your code produces matching outputs.

6.1 Debug Outputs

We have provided debug outputs for you in a separate tarball on Canvas. We produced these debug traces by running the following configurations (here, we use the 64,000-instruction traces, not the two-million–instruction traces):

- The `med` configuration for each trace
- The `tiny`, `med`, `med_noint`, and `big` configurations for `gcc602`

Note that for the sake of brevity (and disk space), the debug outputs may not include the contents of all data structures used in the TA solution. We have also provided `debug_printfs.txt` which contains all of the print statements in the simulator used to generate the debug outputs. The print statements already written in the provided template are not included in `debug_prints.txt`. To enable debug output generation in your own code, use `make DEBUG=1` (you should `make clean` first!) which will allow you to use preprocessor directives to control whether or not your code generates print statements, like this:

```
#ifdef DEBUG
printf("dcache hit for instruction with tag %" PRIu64
       " (dcache index: %" PRIx64 ", dcache tag: %" PRIx64 ")\n",
       my_var1, my_var2, my_var3);
#endif
```

Please do not submit code that always generates debug outputs (that is, it prints without the `#ifdef DEBUG ... #endif` directives), as this will break the autograder and cause you not to match reference outputs.

6.2 Debugging

To debug, please use GDB and Valgrind as demonstrated in Appendix B. We recommend running a 64,000-instruction trace, e.g., passing `-I traces/mcf605_64K.trace` to `procsim`, since the two-million–instruction traces will likely take a while to run under GDB or Valgrind. We also encourage comparing with the debug outputs (the tool `diff` is useful) before coming to office hours for help debugging your code.

7 Experiments

Once your simulator has been validated and matches the `ref_outs`, you will need to find the optimum pipeline configuration for each trace (Only consider the 2M traces: `traces/*_2M.trace`). This means you will submit 4 configurations. **An optimal pipeline has the best possible IPC (Instructions Per Cycle) with the lowest resource utilization.**

Here are the constraints for your experiments:

- Interrupts are enabled
- The Data store of the cache is at least 8KB and at most 32 KiB
- The maximum number of ALU units is 3
- The maximum number of MUL units is 2
- The maximum number of LSU units is 3
- The maximum number of reservation stations per FU is 4
- The maximum size of the ROB is 96 entries and must be a multiple of 16
- You may fetch at most 5 instructions per cycle
- You may retire at most F+2 instructions per cycle

While we do not expect you to search the entire space of configurations, you will need to search a large portion of it to ensure you are not missing information that is crucial to your decision making process.

Within these constraints, you are expected to find a configuration that performs at least 90% as well as the best performer and uses as few resources as possible. You should consider the impact of each architectural knob on the architecture as a whole. When evaluating whether a configuration is optimal, consider the statistics you are calculating beyond just IPC. You are responsible for defining "resources" however you see fit. So long as your justification and reasoning are sound, many interpretations will be accepted. Your report must contain a justification for why your chosen configuration is ideal. Include evidence and analysis in terms of plots, explanations, research, and logic.

Ensure that the report is in a file named `<last name>_report.pdf`. Please submit a PDF and not other file types (no Microsoft Word documents, please).

7.1 Search Space Pruning

As defined, the search space for the experiments produces roughly 20,000 configurations per trace with some naive pruning. This section will detail some simple ways you can reduce the search space.

(Some of these decisions will cause you to miss a few viable configurations, an expense we are willing to pay to allow a reasonable run-time for space exploration)

An easy way to determine some bounds is to first look at performance of the largest configuration per trace. (Unlike with branch predictors, the assumptions we make with the out of order machine indicate that the largest machine performs the best **in simulation**, this need not be true once physically implemented.) We see that the IPC on each trace is greater than 3, and for the `bwaves` benchmark, the 90% threshold also falls above 4. This gives us lower bounds for the fetch and retire rates:

- `bwaves` minimum F: 5
- minimum F for remaining benchmarks: 4
- `bwaves` minimum W: 5
- minimum W for remaining benchmarks: 4

Now let us check out each of the parameters manually to determine their lower bounds. First we start with the cache. By setting all other parameters to the maximum, and starting C at the lowest (13), we find that all benchmarks produce at least one result that exceeds the threshold except `perlbench`, which has a minimum C of 14 to exceed the IPC threshold.

- `perlbench` minimum C: 14
- minimum C for remaining benchmarks: 13

By doing the same with the ALU, we find that we need at least 3 ALUs for each benchmark.

Following on with the MUL unit, we see that there are configurations with 1 MUL unit that meet the threshold, so we must search the range [1,2] for `-M`.

With the load unit, we see something interesting, `bwaves` and `gcc` both have `L=2` configurations that meet the threshold, while the other two benchmarks do not.

- `bwaves` minimum L: 2
- `gcc` minimum L: 2
- minimum L for remaining benchmarks: 3

Arbitrarily, at the expense of losing a small number of well performing configurations, we can set the minimum S to 2 and the minimum ROB size (R) to 32. While this does cause us to lose a few good configurations, it helps to trim the search space substantially and having such a small scheduling queue or ROB is very uncommon in modern microarchitectures.

Based on the search space pruning above, there are much fewer configurations to run for each trace:

- `bwaves`: ∼550 configurations
- `gcc`: ∼1300 configurations
- `mcf`: ∼650 configurations
- `perlbench`: ∼450 configurations

One way you may wish to generate and run these configurations is to write a script that goes through the pruned search space and generates run commands for you in a bash script. Then you can filter the output to generate a csv and organize your results. This is not the only method for running the search space. A reasonable implementation of the project should run any configuration within ∼10 seconds when compiled using `make FAST=1`, running this reduced search space on a single core with no parallelization should take less than ∼12 hours.

8 What to Submit to Gradescope

Please run `make submit` and submit the resulting tarball (`tar.gz`) to Gradescope. Do not submit the assignment PDF, traces, or other unnecessary files (using `make submit` avoids this). Running `make submit` will include PDFs in the project directory in the tarball, but please make sure it worked properly and your experiments report PDF is present in your submission tarball. We will create a simple Gradescope autograder that will verify that your code compiles and matches a subset of reference traces. This autograder is a smoke test to check for any incompatibilities or big issues; it is not comprehensive.
Make sure you untar and check your submission tarball to ensure that all the required files are present in the tar before making your final submission to Gradescope!

9 Grading

You will be evaluated on the following criteria:

+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
+30 :	Your simulator **completely matches** the validation outputs
+15 :	You ran experiments and found the optimum configuration matching the constraints in the Experiments section and presented sufficient evidence and reasoning
+5 :	Your code is well formatted, commented and does not have any memory leaks! Check out Appendix B for some useful tools

Points for the experiments and/or the memory leak check cannot be awarded without first matching all validation traces. This is non-negotiable.

Appendix A - Plagiarism

Preamble: The goal of all assignments in this course 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 copying is Academic Misconduct. NOTE: Debugging someone else's code is (inappropriate) collaboration.

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 are informed at the end of the semester. 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.
  - You can invoke gdb with `gdb ./procsim` and then run `run -I traces/mcf605_64K.trace <more procsim args>` 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 \
    ./procsim -I traces/mcf605_64K.trace <more procsim args>
```