Lab 3: Memory Request Scheduling

Instructor: Prof. Onur Mutlu

TAs: Mohammed Alser, João Dinis Ferreira, Rahul Bera,
Geraldo Francisco De Oliveira Junior, Can Firtina, Juan Gómez Luna,
Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Jeremie Kim,
Nika Mansouri Ghiasi, Haiyu Mao, Lois Orosa Nogueira, Jisung Park,
Minesh Patel, Gagandeep Singh, Kosta Stojiljkovic, Abdullah Giray Yaglikci

Assigned: Tuesday, October 27, 2020
Due: **Tuesday, November 17, 2020** (Midnight)

1. Introduction

In this lab, you will implement and evaluate two memory scheduling policies: ATLAS [1] and BLISS [2]. To this end, you will extend Ramulator [3], which is a publicly-available architectural memory simulator, to model the two memory scheduling policies. Ramulator implements many DRAM standards (e.g., DDR4, LPDDR4, WideIO). It also implements a simple out-of-order processor frontend that enables Ramulator to simulate workloads as a standalone tool. We provide more information on Ramulator and guide you through how to use it in Section 2.

The version of Ramulator that we provide you with already models memory scheduling policies such as first-come-first-serve (FCFS) and first-ready-first-come-first-serve (FRFCFS). In this assignment, your job is to implement the ATLAS and BLISS scheduling policies as specified later in this handout.

2. A Short Ramulator Tutorial

Please follow this short tutorial to learn how to use Ramulator.

Here is the outline of the tutorial:

- Downloading and Building Ramulator
- Simulation Modes
- Simulator Output
- Configuration Files
- Simulating multi-programmed workloads
- Code Organization

2.1. Downloading and Building Ramulator

Ramulator [3] is a fast and cycle-accurate DRAM simulator that supports a wide array of commercial, as well as academic, DRAM standards. Please see the README after downloading Ramulator using the link below.

Exercise 0/6: In Lab 3 on Moodle, we provide you a version of Ramulator that includes memory traces that you will use in this assignment. Please download and extract the tarball:

```
https://moodle-app2.let.ethz.ch/pluginfile.php/994834/mod_assign/introattachment/0/
ramulator_source.zip?forcedownload=1
```

Ramulator requires a *C++11* compiler (e.g., *clang++*, *g++-5*). By default, Ramulator will build using *clang++*. However, you can modify the line starting with `CXX:=` in *'Makefile'* in the Ramulator directory to pick a different compiler.

Exercise 1/6 Compiling Ramulator

You can easily build Ramulator by running *Make*:

```
$ make -j
```

After successful compilation, you will find a binary executable file called *ramulator* in the same directory with the Makefile. You can run the binary to print the help message to the terminal:

Exercise 2/6 Printing Ramulator Help Message

```
$ ./ramulator

Usage: ./ramulator <configs-file> --mode=cpu,dram [--stats <filename>]
    <trace-filename1> <trace-filename2>
Example: ./ramulator ramulator-configs.cfg --mode=cpu cpu.trace cpu.trace
```

2.2. Simulation Modes

Ramulator supports three different usage modes:

- **Memory Trace Driven:** In this mode, Ramulator is provided with an input trace file that contains *main memory requests* of an application to simulate. Ramulator sequentially processes these requests based on the selected DRAM standard (e.g., DDR4). This mode does not model any system in sufficient detail to perform timing simulations. Because of that, Memory Trace Driven mode is better suited for testing the functionality of newly added features. In this assignment, we will *not* use this mode. Still, you can find more information about this mode in the public Ramulator repository [4] in case you are interested.
- **Gem5 Driven:** Gem5 [5] is a full-system simulator that models CPU architecture in detail. Ramulator can be attached to *gem5* to simulate the main memory component of the system. In this assignment, will *not* use this mode. If interested, you can take a look at gem5's homepage for more information about this simulator. You can also find out how to attach Ramulator to gem5 here.
- **CPU Trace Driven:** This is the simulation mode that you will need to use in this lab assignment. In this mode, Ramulator directly reads CPU instruction traces from a file, and simulates a simplified out-of-order CPU core model that generates memory requests to the DRAM subsystem. Such trace files contain non-memory instructions and memory requests. Depending on how the trace file is generated, the memory requests in the trace file may correspond to a certain cache level or directly to the main memory. If the trace contains main memory requests, we call it a *cache-filtered trace.* **When simulating a cache-filtered trace, Ramulator should be configured to not instantiate any caches.** Each line in the CPU trace file represents a memory request, and can have one of the following three formats:
  - **⟨num-cpuinst⟩ ⟨addr-read⟩:** If a line contains two tokens, the first token `⟨num-cpuinst⟩` represents the number of CPU (i.e., non-memory) instructions that precede a read request. The second token `⟨addr-read⟩` specifies the memory address of the read request.
  - **⟨num-cpuinst⟩ ⟨addr-read⟩ ⟨addr-writeback⟩:** The first two tokens in a line with three tokens are the same as in the first format. The third token `⟨addr-writeback⟩` is the decimal address of the writeback request, which is the dirty cache-line eviction caused by the read request before it.
  - **⟨unused⟩ ⟨unused⟩ ⟨num-cpuinst⟩ ⟨type⟩ ⟨addr⟩:** `⟨unused⟩` tokens are used to pass additional information to the simulation, which are not relevant to this assignment and ignored by Ramulator. `⟨num-cpuinst⟩` represents the number of CPU (i.e., non-memory) instructions that precede a memory request. `⟨type⟩` indicates the type of the memory request, which can be a load (L) request or a store (S) request. `⟨addr-read⟩` specifies the memory address of the request.

Exercise 3/6 Running Ramulator in CPU Trace Driven mode:

```
$ ./ramulator configs/test-config.cfg --mode=cpu cpu.trace

tracenum: 1
trace_list[0]: cpu.trace
Warmup complete! Resetting stats...
Starting the simulation...
[0]retired: 26, clk, 224
Simulation done. Statistics written to DDR4.stats
```

In the above command, *'configs/DDR4-config.cfg'* specifies a Ramulator configuration file that contains several parameters related to the architecture to simulate. The second argument, `--mode=cpu`, tells Ramulator to run in CPU Trace Driven mode. The last argument, *'cpu.trace'*, specifies the path to the input trace file to simulate. *'cpu.trace'* is a very short trace that do not represent any real application but rather is used to demonstrate how to run Ramulator.

2.3. Simulator Output

Ramulator reports a series of statistics for every run. These statistics are written to a file. By default, the filename will be *'⟨standard_name⟩.stats'* (e.g., *'DDR4.stats'*). You can output the statistics using a custom filename by adding `--stats ⟨filename⟩` to the command line after the `--mode` argument.

Exercise 4/6: Understanding the stats file:

The simulation in the previous task should have created the stats file *'DDR4.stats'*. Open the stats file with your favorite text editor (e.g., *emacs*, *gedit*, *vim*, *kate*) and find out which statistics Ramulator reports by default. Fill in the values for the metrics below:

Note: you do not have to submit the stats that you filled in above. This is just an exercise to help you get familiar with Ramulator's stats files.

1. *Executed Instructions:*
2. *CPU Cycles:*
3. *IPC:*
4. *Row Misses:*
5. *Row Hits:*
6. *Row Conflicts:*
7. *Average access latency:*
8. *Read Bandwidth:*
9. *Write Bandwidth:*

2.4. Configuration Files

Ramulator enables cycle-accurate simulation of a diverse set of memory technologies. It uses various configuration files to simulate different memory technologies. Now, let's analyze important parameters in the configuration files.

The pre-defined configuration files are available in the *'configs/'* directory. Ramulator is capable of simulating standard *DDRx* memories (e.g., *DDR3-config.cfg*, *DDR4-config.cfg*, *GDDR5-config.cfg*), new 3D-stacked memories (e.g., *HBM-config.cfg*, *WideIO2-config.cfg*), Non-Volatile Emerging Memory Technologies (e.g., *PCM-config.cfg*, *STTMRAM-config.cfg*), and academic proposals (e.g., *SALP-config.cfg*, *DSARP-config.cfg*). For a recent study that shows Ramulator's capabilities in evaluating workload-DRAM configurations, please see the SIGMETRICS 2019 paper entitled *"Demystifying Complex Workload-DRAM Interactions: An Experimental Study"* by Ghose et al. [6].

In this assignment, you will simulate a DDR4-based main memory. If you open the DDR4 configuration file, you will see several parameters that describe the system to simulate. Here is a description of the most important parameters:

1. **Memory-specific parameters:** These are parameters that specify how the memory device will be configured. Important parameters are:
   - *standard*: Specifies the DRAM standard (e.g., *DDR4*).
   - *channels, ranks*: These specify the number of DRAM channels and ranks per channel of the simulated DRAM subsystem.
   - *speed*: This parameter specifies the timing properties of the simulated DRAM device (e.g., *DDR4_2400R*). It sets the frequency of the DRAM device and internal DRAM timings.
   - *org*: Specifies internal organization of the DRAM device (e.g., number of banks, rows, columns).
2. **Core-specific parameters:** These specify the configuration of the CPU cores. Ramulator implements a simple out-of-order core model, and a non-coherent cache hierarchy. The number of cores is determined by the number of trace files passed as arguments during execution (e.g., the first trace is assigned to the first core). The cache hierarchy can be configured using the `cache` knob with `no` for no cache, `L1L2` for private L1 and L2 caches, `L3` for only L3 cache shared among the cores, and `all` for private L1 and L2 and shared L3. In this assignment, you should not instantiate any caches since we provide you cache-filtered traces.
3. **Simulation-specific parameters:** These parameters control the simulation. Since simulation of a program can take orders of magnitude more time to complete than executing the same program on a real system, one common practice is to limit the total number of instructions simulated (`expected_limit_insts`). Setting this parameter will cause the simulation to finish as soon as *all* cores retire at least `expected_limit_insts` instructions. Please make sure to keep `early_exit = off` as otherwise the simulation will finish when *any* core retires `expected_limit_insts` number of instructions.

2.5. Simulating Multi-programmed Workloads

Ramulator allows simulating multi-programmed workloads. A multi-programmed workload is composed of multiple individual applications, each of which is assigned to a CPU core. These applications do not communicate with each other. Therefore, a cache coherence mechanism is not required. However, the individual applications still interfere with each other at different levels of the memory hierarchy since they share the memory sub-system.

To create a multi-programmed simulation, you just need to specify multiple trace files in the command line, separated by space. Ramulator assigns each trace to a different core. For example, if you provide two trace files (e.g., trace1 trace2), Ramulator will automatically instantiate two CPU cores and assign each trace to a different core.

In the *'./traces/'* directory, we provide two CPU trace files. One of them represents an application that accesses main memory a lot (i.e., it has high memory intensity) and the other makes fewer requests to main memory.

Exercise 5/6: Running the multi-programmed workload:

The following command starts simulation with one instance of each trace file we provide in the *'./traces/'* directory.

```
$ ./ramulator configs/DDR4-config.cfg --mode=cpu --stats \
   multi-programmed-simulation.stats ./traces/high-mem-intensity.trace \
   ./traces/low-mem-intensity.trace
```

Note that, in the stats file, some of the statistics are collected and displayed in the output separately for each core.

2.6. Code Organization

Ramulator abstracts basics DRAM operations to provide an easy-to-extend design. You can read Section 2 in the Ramulator paper [3] to understand how the code is organized in detail.

In this section, we give a high-level view of how different Ramulator modules communicate with each other. Figure 1 provides a simplified view of Ramulator's functionalities. We describe CPU Trace Driven simulation in two simulation phases: processor and memory.

Processor-side simulation. The processor-side phase of the simulation consists of 1) reading the trace file; 2) issuing bubble instructions (i.e., non-memory instructions); 3) issuing memory instructions. The processor implements a simple out-of-order core model in *'src/Core.h'* and a cache subsystem in *'src/Cache.h'*. It works as follows.

1. *'src/Main.cpp'* reads the configuration file and instantiates the processor cores and memory controllers. The number of core objects created depends on the number of trace files passed to the simulation and the number of controller objects created depends on the DRAM channels in the configuration.
2. `run_cputrace()` starts running, and then controlling the execution of the simulation.
3. The processor module is ticked (`proc.tick()`), which advances the simulation to the next clock cycle.

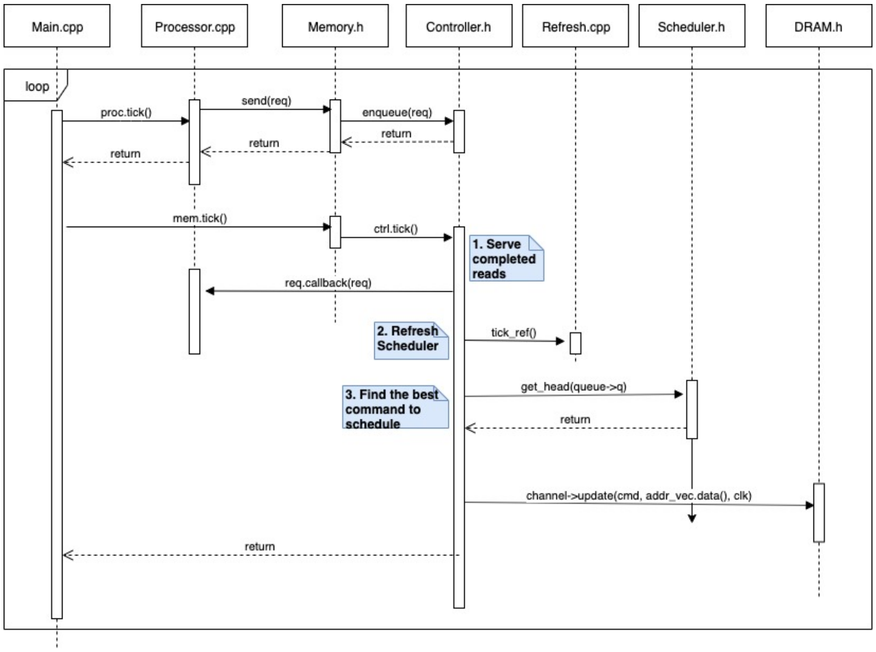

Figure 1. Sequence diagram describing how Ramulator operates.

4. The processor ticks the cores and the cache subsystem. On each clock cycle, each core reads instructions from its trace file. When a core receives a memory instruction from the simulated trace, the core calls `send()` to forward a memory request to the corresponding cache (if the configuration file defines caches) or memory controller. `send()` implements the functionality to determine which controller should service the request. Then, once a controller completes servicing a request, it informs the core that issued the request by calling `callback()`, which is a function passed along with the request object as a member variable.

Memory-side simulation. The memory-side phase of the simulation has three main tasks: 1) to serve completed reads; 2) to refresh the memory device; 3) to schedule DRAM commands. The memory module implements a memory controller (one for each DRAM channel) that is responsible for performing these three tasks. The memory-side simulation works as follows.

1. *'src/Main.cpp'* ticks the memory module (`memory.tick()`), which advances the simulation to the next clock cycle. The memory module then ticks each memory controller.
2. **Serving completed reads.** The controller checks if any request is completed, and if so, it informs the core that issued the requests by calling the corresponding `req.callback(req)` function.
3. **Refreshing the memory device.** The controller calls the refresh module, which is responsible for issuing refresh commands at the defined refresh interval.
4. **Scheduling DRAM commands.** The memory scheduler determines which request in the memory request queue should be serviced next. The selected request depends on the scheduling policy that is in use. Every cycle, the scheduler uses the scheduling policy to scan the requests in the queue to find the most appropriate request to service next, as dictated by the scheduling policy in use. If there is any such request ready to be serviced, the scheduler forwards this request to the controller, and the controller issues the appropriate DRAM command (e.g., `READ`, `ACTIVATE`) for servicing the request. Then, the the controller calls `channel->update()` to update the state of the DRAM device to reflect the effect of the issued DRAM command. It is important to point out that the timing of each DRAM command is defined in the standard DRAM specification (i.e., datasheets). Each memory technology file (e.g., *'src/DDR4.h'*) defines timing specifications for different device types that implement the standard. For the timing parameters, you can see the `SpeedEntry` structure in *'src/DDR4.h'*.

Now, let's see how different memory scheduling policies impact performance. Open *'src/Scheduler.h'* with your favorite text editor. By default, Ramulator employs the *FRFCFS_Cap* scheduler [7]. You can change this scheduler by modifying the line that is shown below:

```
type = Type::FRFCFS_Cap; //Change this line to change scheduling policy
```

Exercise 6/6: Modifying Ramulator source code:

Change the appropriate source files to make Ramulator to use the *FCFS* (First-Come First-Serve) memory scheduler. After the changes, run `make` again to compile Ramulator with the latest changes. Then, simulate the high memory intensity trace and compare the execution time of *FCFS* to the default *FRFCFS_Cap* policy.

3. Your Task 1/3: Implementing ATLAS

Your goal is to extend Ramulator by implementing the *Adaptive per-Thread Least-Attained-Service (ATLAS)* scheduler [1]. The key idea of ATLAS is to periodically order threads based on the service they have attained from the memory controllers, and prioritize threads that have attained the least service compared to the others in each period. This technique significantly reduces the time the CPU cores stall and, as a result, improves system throughput, as shown by Kim et al. [1].

Your task is to extend Ramulator with the ATLAS scheduling policy, as described in Sections 3-5 in the paper that proposed ATLAS [1]. Although you should stick to the exact mechanisms described in the paper, it is your task to figure out how to implement ATLAS in Ramulator. There are multiple ways to extend Ramulator with a new memory scheduling policy. For example, you can add ATLAS as a new scheduler type in *'Scheduler.h'* (similar to other policies such as *FCFS*) and modify/add functions in that header file. **You will not be provided with a specific software design and you are free to implement ATLAS in Ramulator as you find appropriate.**

Although we do not restrict you to a specific software implementation, you should make sure your ATLAS implementation is functionally equivalent to the mechanism described in the paper. Also, please use the default configuration of the ATLAS mechanism that is provided in the paper at the end of Section 6 (i.e., *quantum length* = 10 million cycles, α = 0.875, and T = 100K cycles).

4. Your Task 2/3: Implementing BLISS

In this second task, your goal is to extend Ramulator by implementing the *BLISS* scheduler [2]. The key idea of BLISS is to separate applications in two groups, one containing application with high memory intensity and another that includes applications that access the memory less. The BLISS scheduler achieves its grouping by identifying applications that access a row many times in repetition and deprioritizing them for a determined amount of time. As shown by Subramanian et al. [2], BLISS reduces the interference between the two groups and improves system throughput and fairness.

Your task is to extend Ramulator with the BLISS scheduling policy, as described in Sections 4-5 in the paper that proposed BLISS [2]. Similar to Task 1, you will not be provided a specific way of implementation in Ramulator and you are free to implement BLISS in Ramulator as you find appropriate.

Although we do not restrict you to a specific software implementation, you should make sure your BLISS implementation is functionally equivalent to the mechanism described in the paper. Also, please use the default configuration of the BLISS mechanism provided in the paper at the end of Section 6.5 (i.e., *Blacklisting Threshold* = 4, *Clearing Interval* = 10K cycles).

5. Your Task 3/3: Evaluating ATLAS and BLISS and Comparing Them to Conventional Memory Scheduling Policies

Your task is now to evaluate the *instruction throughput (IT)* and *maximum slowdown (MS)* that your ATLAS and BLISS implementations provide compared to three baseline scheduling policies: FCFS, FRFCFS, and FRFCFS_Cap. Use the following definitions of *instruction throughput (IT)* and *maximum slowdown (MS)*:

$$Instruction\ Throughput\ (IT) = \frac{\sum_{c=0}^{NumCores} InstructionsRetired(c)}{CPUCycles}$$

which is basically the sum of all instructions retired in each core divided by the total number of CPU cycles the simulation took to complete.

$$Maximum\ Slowdown\ (MS) = \max_{a \in Applications} \frac{CPUCycles_{shared}(a)}{CPUCycles_{alone}(a)}$$

To calculate the slowdown of a single application, simply divide the execution time of the application when running *together* with other applications in the same system by the execution time of the application when running the application *alone* on the same system. Maximum slowdown (MS) is the maximum of the single application slowdowns within a multi-programmed workload.

As a first part of the evaluation, **you will have to add instruction throughput as a statistic to Ramulator** such that the output *'.stats'* contains a new *instruction_throughput* line. To do so, you will first need to use the *ScalarStat* class and create an instance of it in a way similar to other statistics that already exist in Ramulator. You must calculate instruction throughput, as defined in the equation above.

It is not possible to directly add *MS* as a statistic to Ramulator as it is required to run Ramulator multiple times (once with all applications together, and once the target application alone) to collect the required information to calculate MS. Thus, **you will need to calculate MS manually or write a script that will read multiple Ramulator stats files and return the MS of each application**.

Do the following when running the simulations:

1. **Make sure you do not change parts of the processor and memory configuration other than those specifically mentioned that you can change in this assignment.**
2. Run simulations until every core retires 20 million instructions.
3. For each scheduling policy, run the following multi-programmed workloads:
   - Workload 1: HLLL (four-core)
   - Workload 2: HHLL (four-core)
   - Workload 3: HHHH (four-core)
   - Workload 4: HHHHHHHH (eight-core)

   where `H` stands for an instance of the trace with high memory intensity and `L` stands for the trace with low memory intensity. We provide both traces, as explained in Section 2.1.

Run Ramulator using the following memory schedulers, collect the instruction throughput and MS of these runs, and analyze the results.

- **FCFS (First-Come First-Serve):** The first memory request to be inserted into the memory request queue is serviced first.
- **FRFCFS (First-Ready First-Come First-Serve):** Similar to FCFS but requests in the request queue that target already open rows are prioritized.
- **FRFCFS_Cap:** Similar to FRFCFS but the number of read/write requests that can be serviced from an already open row is limited to prevent starvation of other requests that target different rows in the same bank. In other words, with this policy, once a row is activated, it could only serve a certain number of read/write requests and then it must be closed (do not change the default parameter of FRFCFS_Cap). See [7] for a more detailed description and evaluation of this policy.
- **ATLAS:** This is your implementation of the ATLAS scheduler described in [1].
- **BLISS:** This is your implementation of the BLISS scheduler described in [2].

Evaluate each workload using each scheduling policy listed above. Collect the *instruction throughput (IT)* and *maximum slowdown (MS)* results and plot them as shown in the template in Figure 2. Note that you should show IT and MS results in separate graphs for all five scheduling policies and four multi-programmed workloads.

Based on your analysis, submit answers to the following questions with your lab report.

1. Provide two graphs, one for IT and another for MS, depicting the metrics for 4 different workloads and 5 different scheduling policies.
2. Explain the two graphs.
3. How does each of the throughput and MS metrics change when using each scheduling policy? Explain why.
4. Do the results match your expectations? Clearly explain what kind of difference each scheduling policy you expect to make. If the results do not match your expectations, try to reason why you may not be seeing the expected results.

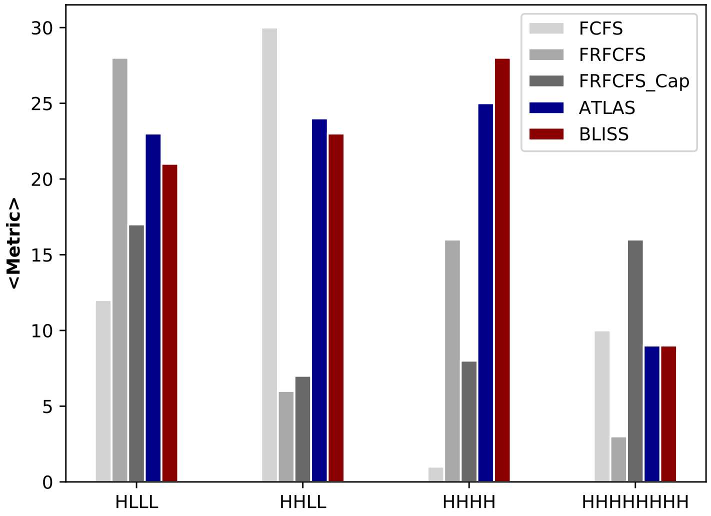

Figure 2. A template showing how *IT* and *MS* results should be plotted.

6. Bonus Task: Designing Your Own Memory Scheduler

In this task, your goal is to come up with a *new* memory scheduling idea (or multiple ones) that hopefully performs better than the existing five scheduling policies. To generate a new idea, you may want to find and study prior work in more detail and cover the research in the area. Alternatively, you can exercise your creativity and insight. You are free to come up with any kind of memory scheduling idea as long as it is your own.

Evaluate your idea in a way similar to how you evaluated ATLAS and BLISS in Task 3. Compare your new idea against all five scheduling policies we mentioned.

Submit 1) a detailed description of your idea, 2) Ramulator implementation of the idea, and 3) the results of your new policy. You may create graphs similar to those you created for Task 3.

You can receive 1.5% of the entire course grade if you come up with a good memory scheduling idea that outperforms the five scheduling policies mentioned in this assignment. You may also receive credit for particularly creative and insightful ideas.

7. Tips

- **Please do not distribute the provided program files. These are for exclusive individual use of each student of the Computer Architecture course. Distribution and sharing violates the copyright of the software provided to you.**
- **Read this handout in detail.**
- **If needed, please ask questions to the TAs using the online Q&A forum in Moodle.**
- When you encounter a technical problem, please first read the error messages. A search on the web can usually solve many debugging issues, and error messages.

8. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/`). You should submit:

- All the files needed to compile your code (including Ramulator source files that you did not change).
- A report as a single PDF file that contains two main sections: 1) section that briefly explains what changes you made in Ramulator to implement the new scheduling policies and 2) section about your analysis from Task 3, including the plotted results.
- All .stat files that are related to your analysis in Task 3.

- **Please do NOT submit the Ramulator trace files provided to you on Moodle.**
- **Also, please do not submit compiled files (e.g., Ramulator executable, .obj files).**

Please submit the above files in a single tarball (with the name 'lab3_⟨*YourSurname*⟩_⟨*YourName*⟩.tar.gz').

References

[1] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[2] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling. *TPDS*, 2016.

[3] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. In *CAL*, 2015.

[4] SAFARI Research Group. Ramulator: A DRAM Simulator — GitHub Repository. `https://github.com/CMU-SAFARI/ramulator`.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 Simulator. *ACM SIGARCH Computer Architecture News*, 2011.

[6] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. Demystifying Complex Workload-DRAM Interactions: An Experimental Study. In *SIGMETRICS*, 2019.

[7] Onur Mutlu and Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.