(Bonus) Lab 5: Memory Request Scheduling

Instructor: Prof. Onur Mutlu, Dr. Juan Gómez Luna, Dr. Mohammad Sadrosadati

TAs: Dr. Yu Liang, Can Firtina, Geraldo Francisco De Oliveira Junior, Abdullah Giray Yaglikci, Ataberk Olgun, Rahul Bera, Konstantinos Kanellopoulos, Nika Mansouri Ghiasi, Rakesh Nadig, Joël Lindegger, Haocong Luo, Nisa Bostanci, Banu Cavlak, Zülal Bingol, Ismail Emir Yuksel

Assigned: Friday, December 1, 2023
Due: **Friday, December 31, 2023** (Midnight)

1. Introduction

In this lab, you will implement and evaluate two memory request scheduling policies: ATLAS [1] and BLISS [2]. To this end, you will extend Ramulator 2 [3], the successor to Ramulator [4], to model the two memory scheduling policies and evaluate their impact to system performance.

In section 2, we guide you through the basics of Ramulator 2, including a simple warm-up task of implementing a First-Come-First-Serve (FCFS) memory request scheduling policy. In section 3 and 4, we give you detailed instructions on how to implement ATLAS and BLISS, respectively. In section 5, we tell you how to evaluate and analyze the system performance impact of the implemented scheduling policies. **Be sure to read the tips (section 7) and the submission guidelines (section 8) first before you start to work on this lab.**

2. Your Task 1/4: Getting Your Hands on Ramulator 2 (20%)

In this section, we provide you with a step-by-step tutorial on how to:

- Build Ramulator 2
- Configure and run a simulation in Ramulator 2
- Understand the Software Design of Ramulator 2
- Extend and modify Ramulator 2

You will encounter **Exercises** that you need to follow, **Notes** that you should pay attention to, and **Questions** that **you need to give answers to in your lab report**, in the following forms.

Exercise 0/6: Download Ramulator 2's source files

Please use the following link to download the lab version of Ramulator 2:

`https://moodle-app2.let.ethz.ch/pluginfile.php/1946934/mod_assign/introattachment/0/ramulator2-lab.zip?forcedownload=1`

Note: We assume that you are using Linux for this lab. Ramulator 2 requires the following tools to build:

- A `c++20`-capable compiler (we have tested Ramulator 2 with `g++-12` and `clang++-15`)
- `cmake 3.14+`

If you are not using Linux or could not meet the prerequisites, we provide you with a Docker image that already contains all the dependencies to build and run Ramulator 2. You can get the image from Docker Hub: `richardluo831/ramulator2:latest` (we also provide a `compose-dev.yaml` in the code repository that uses this image for you to directly initialize a Docker dev environment).

Question 0: What is the build type used when following the above steps to build Ramulator 2? What is the purpose of this build type?

2.1. Building Ramulator 2

Exercise 1/6: Extract the tarball containing Ramulator 2's source files

```
$ tar xvzf ./ramulator2_lab.zip
$ cd ramulator2_lab
```

Note: Ramulator 2 organizes its source code in the following way:

```
ext # External libraries
src # Source code of Ramulator 2.0
 <component1> # Source code of all interfaces and implementations related to the component
   impl # Source code of all implementations of the component
     com_impl.cpp # Source file of a specific implementation
   com_interface.h # Header file that defines an interface
   CMakeList.txt # Component-level CMake configuration
 ...
 CMakeList.txt # Top-level CMake configuration of all Ramulator 2.0's source files
CMakeList.txt # Project-level CMake configuration
```

You can build Ramulator 2 with the commands below:

Exercise 2/6 Build Ramulator 2

```
$ mkdir build
$ cd build
$ cmake ..
$ make -j4
```

Note: By default (i.e., if you follow the instructions above), Ramulator 2 will be built in the "Debug" build type. The binary built this way will have the debug symbols added and compiler optimizations disabled. If you have already debugged your implementation of the memory request scheduling mechanisms, you should build Ramulator in "Release" type to enable compiler optimizations to save simulation time. To do so, add `-D CMAKE_BUILD_TYPE=Release` when invoking `cmake`.

After successful compilation, you should have the executable file `ramulator2` in the `build/` directory.

2.2. Configuring and Running a Simulation in Ramulator 2

Ramulator 2 uses YAML[1] to specify all its configurations in the form of hierarchical *key-value pairs*. Every key-value pair specifies a system component modeled in Ramulator 2. The main configuration file that you will be using in this lab `lab5_config.yaml` is provided in the root directory of Ramulator 2.

Exercise 3/6 Check the baseline configuration of Ramulator 2 that you will use in this lab

```
$ cd ..
$ less ./lab5_config.yaml
```

Question 1: What kind of a memory system does this configuration specify? Name as many specifications as you can about this system.

At a high-level, Ramulator 2 consists of two main components: A simulation *frontend* (specified by the `FrontEnd` key in the configuration) and a *memory system* (specified by the `MemorySystem` key in the configuration). The frontend generates memory requests and sends them to the memory system.

Currently, Ramulator 2 implements the following three frontends:

- **Memory Trace Frontend:** This frontend reads an input trace file that contains *main memory requests* of an application. It *sequentially* sends these requests to the memory system. This mode does not

[1] You can learn more about YAML here: `https://yaml.org/spec/1.2.2/`

model any system in sufficient detail to perform timing simulations. Because of that, the Memory Trace Frontend is better suited for testing/debugging the functionality of newly added features. In the main tasks of this assignment, we will *not* use this frontend. Still, feel free to use this mode to test and debug your newly implemented features. Currently, Ramulator 2 implements the following two memory trace frontends:

  - **Load-Store Trace:** This frontend expects a trace file containing load (L) or store (S) requests with a *physical* address (i.e., from the OS). Each line in the trace has the following format: **L/S** ⟨**addr**⟩
  - **Read-Write Trace:** This frontend expects a trace file containing read (R) or store (W) requests with a *DRAM address vector* (i.e., which channel, rank, bank, row, column). Each line in the trace has the following format: **R/W** ⟨**comma-separated addr vector**⟩**:**

- **gem5 Frontend:** Gem5 [5] is a full-system simulator that models CPU architecture in detail. Ramulator 2 can be attached to *gem5* to simulate the main memory component of the system when using the gem5 frontend. In this assignment, will *not* use this frontend. 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 2 to gem5 here.
- **Simlpe Out of Order (SimpleO3) CPU Trace Frontend:** This is the simulation frontend you will need to use in this lab. This frontend models a simplistic out-of-order processor with a last-level cache that supports two kinds of instructions: memory instructions and non-memory instructions. It reads in application trace file(s) that each corresponds to a core in the processor (i.e., if you specify four trace files, it will simulate a four-core multiprogrammed workload). Each line in the trace can have one of the following two formats:
  - ⟨**num-cpuinst**⟩ ⟨**addr-read**⟩**:** The first token ⟨`num-cpuinst`⟩ represents the number of 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 address of the write-back request, which is the dirty cache-line eviction (from L1 and L2 to the last-level cache) caused by the read request before it.

A memory system typically consists of the following three components:

- **The DRAM Device:** This is the functional and timing model of the simulated DRAM memory device. Its key parameters include:
  - The organization of the DRAM memory (e.g., the storage density of each chip, how many channels, ranks, banks, rows, columns are there)
  - The timing constraints, usually based on a preset but allows the user to override certain timing parameters by specifying them individually in the configuration
- **The Memory Controller:** This is the memory controller that is responsible for decoding the memory requests into low-level DRAM commands, and then schedules the commands according to the functional and timing constraints of the DRAM device to serve the memory requests. The most important component inside the memory controller is the request scheduler.
- **The Address Mapper:** This is responsible for mapping the physical address from the frontend to the DRAM address vector (e.g., which channel, rank, bank, row, column).

Now, let us run Ramulator 2 with the given example configuration file. The standard output (stdout) should print logs, including the statistics at the end of the simulation.

Exercise 4/6 Run Ramulator 2 with the given configuration

```
$ ./ramulator -f ./lab5_config.yaml
```

Question 2: How do you change the input trace file(s) for the SimpleO3 frontend? How do you change how many CPU cycles that Ramulator 2 is going to simulate?

Question 3: Run a four-core multiprogrammed simulation with two cores running the "L" trace (i.e., `low-mem-intensity.trace`) and the other two running the "H" trace (i.e., `high-mem-intensity.trace`). Hint: the `traces` key expects a *list* as its value in the configuration file.

Compare the instruction per cycle (IPC) of the application that is running alone (i.e., single-core) versus running together with other applications (i.e., multi-core), how different are they? Briefly explain why.

2.3. Understanding the Software Design of Ramulator 2

Ramulator 2.0 models all components in a DRAM-based memory system with two fundamental concepts, *Interface* and *Implementation*, to achieve high modularity and extensibility. An interface is an abstract C++ class defined in a `.h` header file that models the common high-level functionality of a component as seen by *other* components in the system. An interface class defines virtual functions that *other* components can call. An implementation is a concrete C++ class defined in a `.cpp` file that inherits from an interface, modeling the actual behavior of a component. An implementation class provides concrete implementations of its interface's virtual functions. Components interact with each other through pointers to each other's interfaces stored in the implementations. With such a design, the functionality of a component can be changed by instantiating a different implementation for the same interface, involving *no* changes in the code of unrelated components.

Figure 1 is a simplified sequence diagram that shows 1) the high-level software architecture of Ramulator 2 using an example DDR4 system configuration, and 2) how a memory request is served by Ramulator 2. The dark boxes are the interfaces, and the white boxes above them are their typical implementations. The arrows illustrate the relationships among different components in the simulated system (i.e., how they call each other's interface functions). We highlight the memory request path with red arrows and DRAM command path with blue arrows. With the SimpleO3 CPU Trace FrontEnd that we are going to use in this lab, a typical execution of the simulation is as follows:

1. First, memory requests (cache misses and evictions) are sent ❶ from the frontend to the memory system.
2. The address mapper maps ❷ the physical address of the request into the DRAM address vector.
3. The request is then enqueued ❸ into the corresponding DRAM memory controller.
4. The DRAM memory controller queries the request scheduler ❹ to find the best request, among all the requests that are being queued, to serve.
5. The request scheduler decodes ❺ the memory requests into corresponding DRAM commands based on the type of the request, the current state of the DRAM device, the timing constraints, etc.
6. The request scheduler decides on the best request to schedule according to its scheduling policy, and then returns the corresponding DRAM command ❻ to the DRAM controller.
7. The DRAM controller issues ❼ the scheduled DRAM command, which updates the behavior and timing information of the DRAM device model.
8. Finally, if there is a completed memory request, the memory controller calls its callback ❽ to notify the frontend of the completion of the request.

In Ramulator 2, we observe that many modeled functions in the memory controller (e.g., controller-based

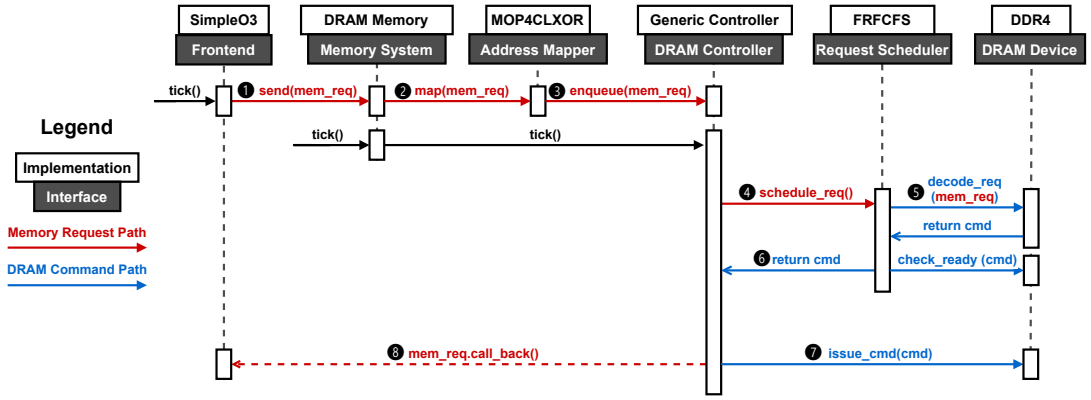

Figure 1. Simplified sequence diagram showing the high-level software architecture of Ramulator 2.0 using an example DDR4 system configuration

RowHammer mitigations that tracks the issued activation commands) and utilities needed for evaluation (e.g., collecting statistics from the issued DRAM commands and analyzing the memory access patterns) are triggered (updated) by the issued DRAM command ❼. To avoid having many similar memory controller implementations for every single such modeled function and utility, we model these functions as plugins to the memory controller. The plugin interface has a simple `update()` function that the controller calls to notify the plugin implementations about the request scheduled by the memory controller (along with the DRAM command and address that will be issued).

Exercise 5/6: Instantiate a memory controller plugin:

Instantiate a "TraceRecorder" memory controller plugin to the DRAM memory controller by changing the configuration file to the following:

```
MemorySystem:
#...
  Controller:
#...
    plugins:
      - ControllerPlugin:
          impl: TraceRecorder
          path: ./example.dram.trace
#...
```

Question 4: What does the "TraceRecorder" plugin do?

2.4. Extending and Modifying Ramulator 2

Interface and implementation class definitions in Ramulator 2 should follow certain structures. An example interface class `ExampleIfce` named "ExampleInterface" (defined in `example_interface.h`) should be defined like the following:

```
// example_interface.h
#ifndef RAMULATOR_EXAMPLE_INTERFACE_H
#define RAMULATOR_EXAMPLE_INTERFACE_H

// Defines fundamental data structures and types of Ramulator 2. Must include for all interfaces
#include "base/base.h"

namespace Ramulator {
class ExampleIfce {
  // One-liner macro to register this "ExampleIfce" interface with the name "ExampleInterface" to Ramulator 2
  RAMULATOR_REGISTER_INTERFACE(ExampleIfce, "ExampleInterface", "An example of an interface class.")
  public:
    // Model common behaviors of the interface with virtual functions
    virtual void foo() = 0;
};
} // namespace Ramulator

#endif // RAMULATOR_EXAMPLE_INTERFACE_H
```

An example implementation class `ExampleImpl` named "ExampleImplementation" that implements `ExampleIfce` (defined in `example_impl.cpp`) should be defined like the following:

```
// example_impl.cpp
#include <iostream>

// An implementation should always include the header of the interface that it is implementating
#include "example_interface.h"

namespace Ramulator {
// An implementation class should always inherit from *both* the interface it is implementating, and the "Implementation" base class
class ExampleImpl : public ExampleIfce, public Implementation {
  // One-liner macro to register and bind this "ExampleImpl" implementation with the name "ExampleImplementation" to the "ExampleIfce" interface.
  RAMULATOR_REGISTER_IMPLEMENTATION(ExampleIfce, ExampleImpl, "ExampleImplementation", "An example of an implementation class.")
  public:
    // Implements concrete behavior
    virtual void foo() override {
      std::cout << "Bar!" << std::endl;
    };
};
} // namespace Ramulator
```

Ramulator 2 will search for the names you give to the interface and implementation through the one-liner macro (i.e., "ExampleInterface" and "ExampleImplementation") when parsing the YAML configuration file. For example, it will automatically construct an `ExampleImpl` object when seeing the following configuration:

```
#...
ExampleInterface:
    impl: ExampleImplementation
#...
```

To add a new implementation to an existing interface in Ramulator 2, simply create a new `.cpp` file containing the new implementation class. Be sure to follow the structures (e.g., what headers to include, how should the new implementation class inherit from other base classes, how to use the macro to register the new implementation) as the examples have shown above. Do not forget to add the new `.cpp` file to the `cmake` build system of Ramulator 2.

Exercise 6/6: Add a First-Come-First-Serve (FCFS) scheduler to Ramulator 2:

Create a First-Come-First-Serve (FCFS) scheduler implementation to the existing `IScheduler` interface (defined in `src/dram_controller/scheduler.h`).

Hint: Ramulator 2 already implements a First-Ready-First-Come-Firse-Serve (FRFCFS) scheduler in `src/dram_controller/impl/scheduler/FRFCFS.cpp`.

Question 5: What is the difference between a FCFS scheduler and FRFCFS scheduler? How does it impact system performance?

3. Your Task 2/4: Implementing ATLAS (30%)

Your goal is to extend Ramulator 2 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 2 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 2. **Although we recommend you follow the interface-implementation design of Ramulator 2, you will not be provided with a specific software design and you are free to implement ATLAS in Ramulator as you find appropriate.**

Note that 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 3/4: Implementing BLISS (30%)

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

Note that 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 4/4: Evaluating ATLAS and BLISS and Comparing Them to Conventional Memory Scheduling Policies (20%)

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 and FRFCFS. 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 2** such that the statistics printed to the standard output contains a new *instruction_throughput* entry. You must calculate instruction throughput, as defined in the equation above. Hint: Take a look at how are the existing statistics printed in the standard output implemented.

It is not possible to directly add *MS* as a statistic to Ramulator 2 as it is required to run Ramulator 2 multiple times with different configurations (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 2 output statistics and return the MS of each configuration**.

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 2 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.
- **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 appropriately. Note that you should show IT and MS results in separate graphs for all four 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 four different workloads and four different scheduling policies.

2. Explain the two plots.
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.

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 4. Compare your new idea against all four scheduling policies we mentioned.

Submit 1) a detailed description of your idea, 2) Ramulator 2 implementation of the idea, and 3) the results of your new policy. You may create plots similar to those you created for Task 4.

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 2 source files that you did not change).
- A report as a single PDF file that contains three main sections: 1) section your answers to all the questions in task 1, 2) section that briefly explains what changes you made in Ramulator to implement the new scheduling policies and 3) section about your analysis from Task 5, including the plotted results.
- All standard output produced by Ramulator 2 that are related to your analysis in Task 5.
- **Please do NOT submit the trace files provided to you on Moodle.**
- **Also, please do not submit compiled files (e.g., Ramulator2 executable, .obj files).**

Please submit the above files in a single tarball (with the name 'lab5_⟨*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] Haocong Luo, Yahya Can Tuğrul, F. Nisa Bostancı, Ataberk Olgun, A. Giray Yağlıkçı, and Onur Mutlu. Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator. arXiv:2308.11030 [cs.AR], 2023.

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

[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.