

Project - 18

DRAM Scheduling Algorithm –

Project Final Report



Submitted By:

Group - 9

Anish Patil - 2024H1030042G
Rohit Sartandel - 2024H1030034G
Aditya Arkasali - 2024H1030046G

To:
Dr. Kanchan Manna

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI
– K. K. BIRLA GOA CAMPUS

INTRODUCTION

This project aims to enhance the way memory requests are managed in DRAM systems by introducing more efficient scheduling strategies. Memory scheduling is essential for determining the order in which requests are handled, significantly impacting system speed, responsiveness, and fairness. Traditional methods often suffer from drawbacks like increased latency, unequal memory access, and performance bottlenecks. To overcome these limitations, the project focuses on designing innovative scheduling techniques that prioritize requests more effectively and distribute the workload evenly.

To assess the performance of these new methods, the system will be tested under various workloads and memory setups. Key performance indicators such as request latency, row buffer hit rate, incoming requests, serving request, memory bandwidth, etc. will be evaluated. These results will highlight how the proposed techniques compare to conventional scheduling methods.

Ultimately, by refining memory request handling, the project seeks to boost overall system performance, minimize delays, and promote smoother, more efficient memory operations. Such advancements are especially beneficial in environments where multiple applications demand simultaneous memory access, including high-performance computing, gaming, and real-time data processing.

A good DRAM scheduling algorithm can significantly improve system performance by:

- **Reducing Latency:** Getting data to the CPU faster.
- **Increasing Throughput:** Handling more memory requests per unit of time.
- **Improving Energy Efficiency:** By optimizing DRAM operations, we can also reduce power consumption.

PROJECT SETUP

To study and evaluate new scheduling techniques, this project uses Ramulator, a high-performance, open-source, cycle-accurate DRAM simulator.

Ramulator is widely used due to the following reasons:-

- **Support for multiple DRAM standards** (DDR3, DDR4, LPDDR4, HBM, GDDR, etc.)
- **Flexible and modular design**, allowing easy integration of custom scheduling policies
- **Trace-driven and full-system simulation compatibility**

Ramulator provides essential performance metrics such as:

- **Request latency**
- **Row buffer hit/miss rate**
- **Memory bandwidth**
- **Command scheduling delays**

These capabilities make it an ideal platform for testing and comparing novel scheduling strategies under controlled and repeatable conditions.

We have implemented four DRAM Scheduling algorithms and are focusing on attributes that are crucial for understanding the performance of these algorithms.

1. FCFS (First-Come, First-Served)
2. FRFCFS (First-Ready, First-Come, First-Served)
3. FRFCFS_Cap (FRFCFS with Row Hit Capacity)
4. FRFCFS_PriorHit (FRFCFS with Priority to Row Hits)

1. **FCFS (First-Come, First-Served):-** In this algorithm, requests are processed strictly in the order of their arrival in the request queue. While straightforward to implement and inherently fair by treating all requests equally based on arrival time, FCFS is often inefficient. It doesn't take into account the internal characteristics of DRAM, such as row buffer locality or readiness. This lack of awareness can lead to suboptimal performance as it may not prioritize requests that are immediately serviceable or those that could benefit from existing open rows.
2. **FRFCFS (First-Ready, First-Come, First-Served):-** FRFCFS enhances efficiency by prioritizing ready requests. A request is considered ready when the DRAM is capable of immediately serving it, meaning there are no timing constraints preventing its execution. Among all requests that are ready to be served, FRFCFS still maintains the original FCFS policy, ensuring fairness amongst those eligible for immediate processing. By focusing on ready requests, FRFCFS effectively reduces unnecessary waiting time and increases the overall efficiency and throughput of the DRAM system.
3. **FRFCFS_Cap (FRFCFS with Row Hit Capacity):-** This algorithm incorporates a capacity limit on the number of consecutive row hits that can be served to a single DRAM row. After a certain threshold of row hits is reached for a particular row, the scheduler might be compelled to consider other requests, even if they are not row hits themselves. This capacity mechanism is designed to prevent a single row from monopolizing the DRAM resources, promoting fairness across different requests and mitigating potential row starvation issues.
4. **FRFCFS_PriorHit (FRFCFS with Priority to Row Hits):-** This policy explicitly prioritizes requests that are identified as row hits, meaning they access a row that is already open in a DRAM bank. When the scheduler examines ready requests, it first checks for the presence of any row hits. If row hit requests are available, they are given precedence over other ready requests. By aggressively prioritizing row hits, FRFCFS_PriorHit aims to minimize the average latency of memory accesses, leveraging the significantly faster access time of row hits compared to row misses.

Attributes that are crucial for understanding the performance of these algorithms:-

1. **Row Misses per channel per core:** The number of times a memory request requires data from a row that is not currently open in the DRAM bank, and no other row is open either. This usually means a new row needs to be opened.
2. **Row Conflicts per channel per core:** The number of times a memory request needs to access a row, but a different row is currently open in the same DRAM bank. This necessitates closing the currently open row (using a PRECHARGE command) and then opening the desired row (ACTIVATE).
3. **Active Cycle:** The number of DRAM cycles the DRAM channel spends in an active state. A channel is active when at least one bank within it has an open row (due to an ACTIVATE command).

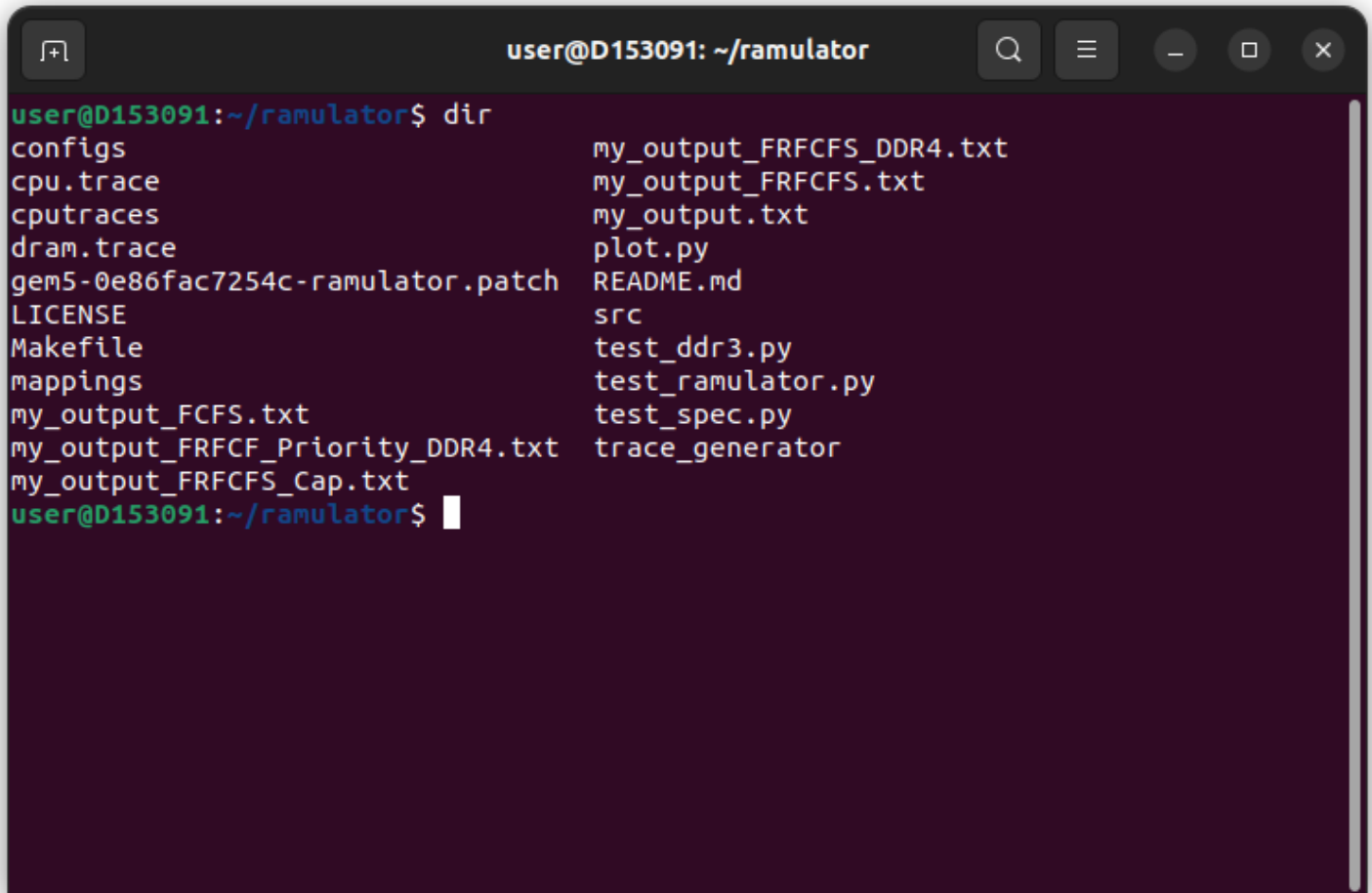
4. **Busy Cycle:** Similar to Active Cycle, but it's a broader measure. It represents the number of DRAM cycles the DRAM channel is busy processing any command (including ACTIVATE, PRECHARGE, READ, WRITE, REFRESH, etc.).
5. **Serving Request:** The average number of requests actively being served by the DRAM channel at any given time. This could be requests that have been issued and are in the process of being executed by the DRAM chips.
6. **Read Latency:** The average time (in DRAM cycles) it takes for a read request to be completed, from the time it's enqueued in the controller to the time the data is returned.
7. **Row Hits for the read Request per channel per core:** Specifically for read requests, the number of row hits encountered.
8. **Row Misses for the read Request per channel per core:** Specifically for read requests, the number of row misses encountered.
9. **Row Conflicts for the read Request per channel per core:** Specifically for read requests, the number of row conflicts encountered.
10. **Incoming Requests:** The total number of requests (reads and writes) that were enqueued into the DRAM controller's queues during the simulation.

We should focus on these attributes because they directly reflect the efficiency and performance of your DRAM scheduling algorithms. By comparing these metrics across different algorithms (like FCFS, FRFCFS, etc.) we can:

- **Quantify Performance Differences:** See which algorithms lead to lower latency, higher throughput, and potentially better energy efficiency (though energy is not directly measured in these attributes, latency and conflicts are strongly related to energy consumption).
- **Identify Strengths and Weaknesses:** Understand which algorithms are better at exploiting row hits, reducing conflicts, and managing DRAM resources effectively.
- **Validate Algorithm Design:** Confirm whether your implemented algorithms are working as expected and if the design choices are leading to performance improvements.
- **Make Data-Driven Decisions:** Use the simulation results to guide the design and optimization of DRAM scheduling algorithms for real-world systems.

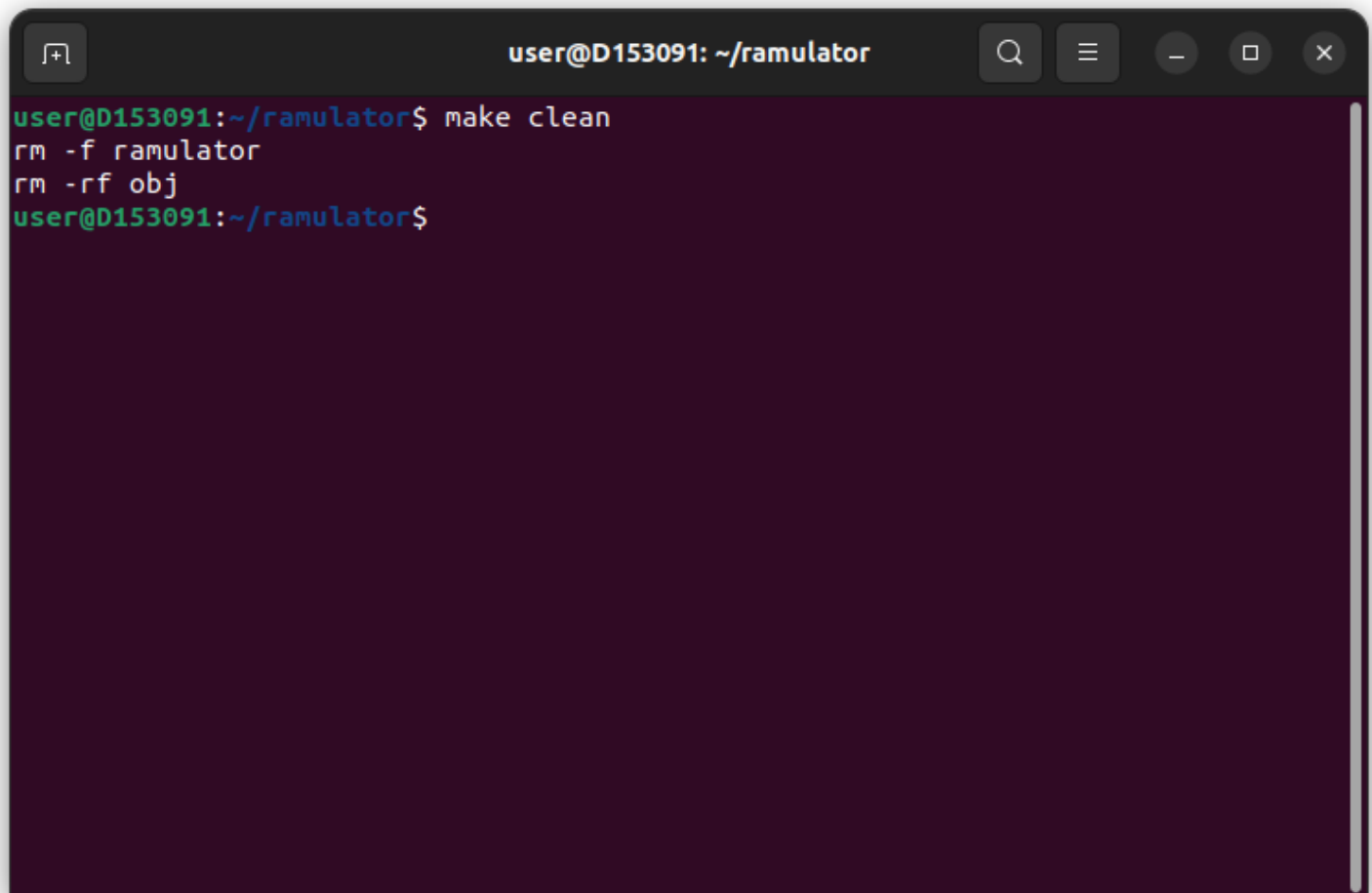
OUTPUT:-

1. Ramulator Directory:-



```
user@D153091: ~/ramulator
user@D153091:~/ramulator$ dir
configs
cpu.trace
cputraces
dram.trace
gem5-0e86fac7254c-ramulator.patch
LICENSE
Makefile
mappings
my_output_FCFS.txt
my_output_FRFCF_Priority_DDR4.txt
my_output_FRFCFS_Cap.txt
my_output_FRFCFS_DDR4.txt
my_output_FRFCFS.txt
my_output.txt
plot.py
README.md
src
test_ddr3.py
test_ramulator.py
test_spec.py
trace_generator
user@D153091:~/ramulator$
```

2. Make clean command:-



```
user@D153091: ~/ramulator
user@D153091:~/ramulator$ make clean
rm -f ramulator
rm -rf obj
user@D153091:~/ramulator$
```

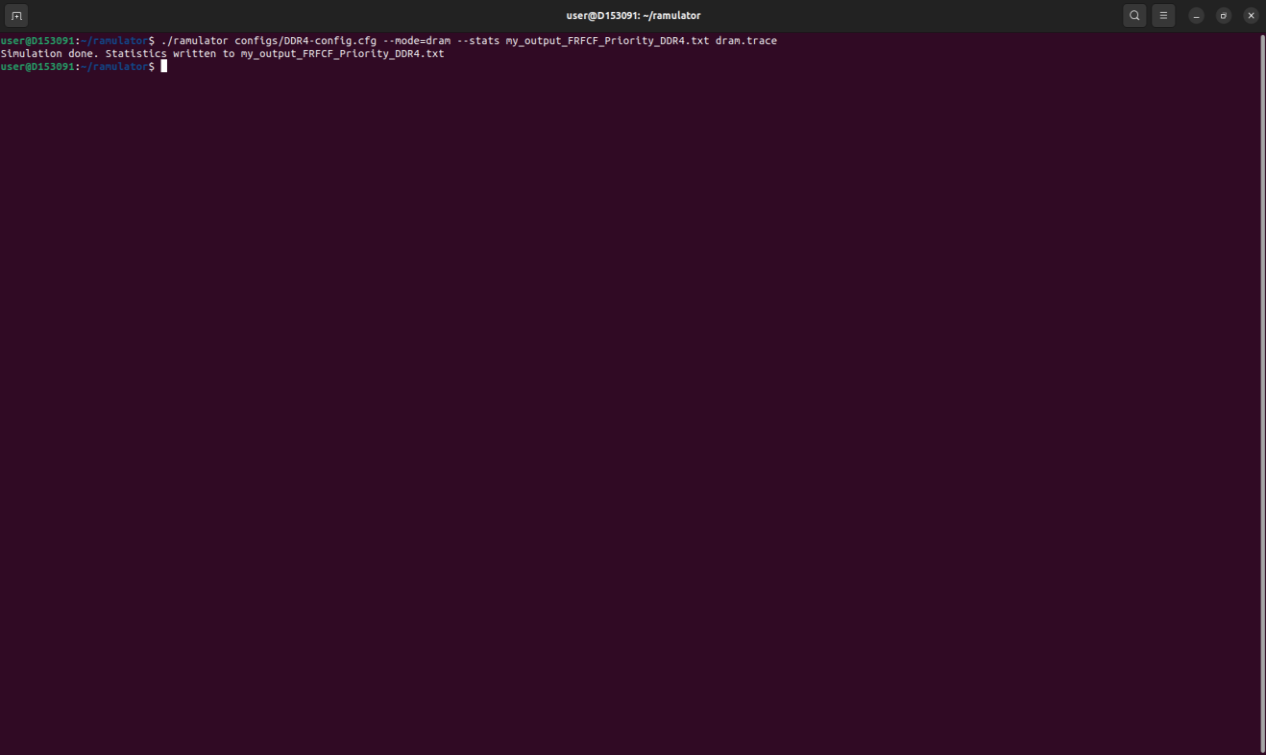
A terminal window with a dark background and light-colored text. The window title bar shows 'user@D153091: ~/ramulator' and standard window control buttons (search, menu, zoom, close). The terminal content shows the user entering 'make clean', which triggers two 'rm' commands: 'rm -f ramulator' and 'rm -rf obj'. The prompt returns to 'user@D153091:~/ramulator\$'.

3. Compile & Build:- make -j3

Run:- ./ramulator configs/DDR3-config.cfg--mode=dram--stats my_output.txt dram.trace

```
user@D153091: ~/ramulator
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/ALDRAM.o src/ALDRAM.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/Cache.o src/Cache.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/Config.o src/Config.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/Controller.o src/Controller.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/DDR3.o src/DDR3.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/DDR4.o src/DDR4.cpp
In file included from src/Controller.cpp:1:
In file included from src/Controller.h:16:
src/Scheduler.h:137:17: warning: misleading indentation; statement is not part of the previous 'for' [-Wmisleading-indentation]
    count++;
    ^
src/Scheduler.h:135:13: note: previous statement is here
    for (auto itr = next(q.begin(), 1); itr != q.end(); itr++)
    ^
src/Scheduler.h:157:17: warning: misleading indentation; statement is not part of the previous 'for' [-Wmisleading-indentation]
    count++;
    ^
src/Scheduler.h:155:13: note: previous statement is here
    for (auto itr = next(q.begin(), 1); itr != q.end(); itr++)
    ^
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/DSARP.o src/DSARP.cpp
2 warnings generated.
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/GDOR5.o src/GDOR5.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/HBM.o src/HBM.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/LPDDR3.o src/LPDDR3.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/LPDDR4.o src/LPDDR4.cpp
clang++ -O3 -std=c++11 -g -Wall -DRAMULATOR -c -o obj/MemoryFactory.o src/MemoryFactory.cpp
In file included from src/MemoryFactory.cpp:1:
In file included from src/MemoryFactory.h:9:
In file included from src/Memory.h:7:
In file included from src/Controller.h:16:
src/Scheduler.h:137:17: warning: misleading indentation; statement is not part of the previous 'for' [-Wmisleading-indentation]
    count++;
    ^
src/Scheduler.h:135:13: note: previous statement is here
    for (auto itr = next(q.begin(), 1); itr != q.end(); itr++)
    ^
src/Scheduler.h:157:17: warning: misleading indentation; statement is not part of the previous 'for' [-Wmisleading-indentation]
    count++;
    ^
src/Scheduler.h:155:13: note: previous statement is here
    for (auto itr = next(q.begin(), 1); itr != q.end(); itr++)
    ^
src/Scheduler.h:244:10: warning: lambda capture 'this' is not used [-Wunused-lambda-capture]
    [this] (ReqIter req1, ReqIter req2) {
    ^
src/Scheduler.h:107:5: note: in instantiation of default member initializer 'ramulator::Scheduler<ramulator::WideIO2>::compare' requested here
    Scheduler(Controller<T>* ctrl) : ctrl(ctrl) {}
    ^
src/Controller.h:109:23: note: in instantiation of member function 'ramulator::Scheduler<ramulator::WideIO2>::Scheduler' requested here
    scheduler(new Scheduler<T>{this}),
    ^
src/MemoryFactory.h:48:13: note: in instantiation of member function 'ramulator::Controller<ramulator::WideIO2>::Controller' requested here
    ctrls.push_back(new Controller<T>(configs, channel));
    ^
```


4. Running Simulator by selecting specific Algorithm:-



```
user@D153091: ~/ramulator
user@D153091:~/ramulator$ ./ramulator configs/DDR4-config.cfg --node=dran --stats my_output_FRFCF_Priority_DDR4.txt dran.trace
Simulation done. Statistics written to my_output_FRFCF_Priority_DDR4.txt
user@D153091:~/ramulator$
```

RESULTS:-

DRAM 3 -

ATTRIBUTE	FCFS	FRFCFS	FRFCFS WITH PRIORITY	FRFCFS_CAP
Row hits per channel per core	105	90	95	88
Row misses per channel per core	5	6	3	4
Row conflicts per channel per core	2	4	3	12
Active Cycle	721	400	360	440
Busy Cycle	721	400	360	440
Serving Request	1400	1001	932	1003
Read Latency	740.33	217.287879	227.67	220.287879
Row Hits for the read Request per channel per core	63	72	54	63
Row Misses for the read Request per channel per core	3	4	3	4
Row Conflicts for the read request	21	2	3	7
Incoming Requests	110	110	110	110

Simulation Output Text File:-

Scheduler.h			my_output_FRFCFS_DDR4.txt		
77	ramulator.busy_cycles_0_0_3_1	0	# Total active cycles for level _0_0_3_1	# (All-bank refresh only. busy cycles only include refresh time in rank level) The sum of cycles that	
78	the DRAM part is active or under refresh for level _0_0_3_1	0	# The sum of read and write requests that are served in this DRAM element per memory cycle for level		
79	ramulator.serving_requests_0_0_3_1	0	# The average of read and write requests that are served in this DRAM element per memory cycle for		
80	ramulator.average_serving_requests_0_0_3_1	0.000000	# Total active cycles for level _0_0_3_2	# (All-bank refresh only. busy cycles only include refresh time in rank level) The sum of cycles that	
81	ramulator.active_cycles_0_0_3_2	0	# The sum of read and write requests that are served in this DRAM element per memory cycle for level		
82	ramulator.busy_cycles_0_0_3_2	0	# The average of read and write requests that are served in this DRAM element per memory cycle for		
83	the DRAM part is active or under refresh for level _0_0_3_2	0	# Total active cycles for level _0_0_3_3	# (All-bank refresh only. busy cycles only include refresh time in rank level) The sum of cycles that	
84	ramulator.serving_requests_0_0_3_2	0.000000	# The sum of read and write requests that are served in this DRAM element per memory cycle for level		
85	ramulator.average_serving_requests_0_0_3_2	0.000000	# The average of read and write requests that are served in this DRAM element per memory cycle for		
86	ramulator.active_cycles_0_0_3_3	16	# Total active cycles for level _0_0_3_3	# (All-bank refresh only. busy cycles only include refresh time in rank level) The sum of cycles that	
87	ramulator.busy_cycles_0_0_3_3	16	# The sum of read and write requests that are served in this DRAM element per memory cycle for level		
88	the DRAM part is active or under refresh for level _0_0_3_3	16	# The average of read and write requests that are served in this DRAM element per memory cycle for		
89	ramulator.serving_requests_0_0_3_3	0.030534	# The total byte of read transaction per channel		
90	ramulator.read_transaction_bytes_0	4224	# The total byte of write transaction per channel		
91	ramulator.write_transaction_bytes_0	2816	# Number of row hits per channel per core		
92	ramulator.row_hits_channel_0_core	105	# Number of row misses per channel per core		
93	ramulator.row_misses_channel_0_core	5	# Number of row conflicts per channel per core		
94	ramulator.row_conflicts_channel_0_core	0	# Number of row hits for read requests per channel per core		
95	ramulator.read_row_hits_channel_0_core	63	# Number of row misses for read requests per channel per core		
96	ramulator.read_row_misses_channel_0_core	3	# Number of row conflicts for read requests per channel per core		
97	ramulator.read_row_conflicts_channel_0_core	0	# Number of row hits for write requests per channel per core		
98	ramulator.write_row_hits_channel_0_core	42	# Number of row misses for write requests per channel per core		
99	ramulator.write_row_misses_channel_0_core	2	# Number of row conflicts for write requests per channel per core		
100	ramulator.write_row_conflicts_channel_0_core	0	# The memory latency cycles (in memory time domain) sum for all read requests in this channel		
101	ramulator.read_latency_sum_0	14341	# Average of read and write queue length per memory cycle per channel.		
102	ramulator.req_queue_length_avg_0	36.629771	# Sum of read and write queue length per memory cycle per channel.		
103	ramulator.req_queue_length_sum_0	19194	# Read queue length average per memory cycle per channel.		
104	ramulator.read_req_queue_length_avg_0	27.276718	# Read queue length sum per memory cycle per channel.		
105	ramulator.read_req_queue_length_sum_0	14293	# Write queue length average per memory cycle per channel.		
106	ramulator.write_req_queue_length_avg_0	9.353053	# Write queue length sum per memory cycle per channel.		
107	ramulator.write_req_queue_length_sum_0	4901	# record read hit count for this core when it reaches request limit or to the end		
108	ramulator.record_read_hits	0	# record_read_miss count for this core when it reaches request limit or to the end		
109	ramulator.record_read_misses	0	# record_read_conflict count for this core when it reaches request limit or to the end		
110	ramulator.record_read_conflicts	0			

Milestone 2 PLAN:-

1. To improve the performance, we will be implementing two algorithms- ATLAS and BLISS.
ATLAS improves performance by adaptively prioritizing requests based on their waiting time and row hits, thus reducing overall latency and improving responsiveness. BLISS, on the other hand, enhances performance by ensuring bank-level fairness and limiting request streaks, which reduces bank contention and improves throughput. Together, they move beyond basic scheduling by actively managing latency (ATLAS) and bank utilization (BLISS) to optimize DRAM efficiency.
2. We will test the performance of all the algorithms on DRAM 4 configurations and will observe improvements if any over DRAM 3 configuration.

MILESTONE II

1. OVERVIEW & PURPOSE

Creation and implementation of additional DRAM scheduling algorithms **ATLAS** and **BLISS**.

2. PRE-REQUISITES

- Build on Ubuntu 22.04 using Oracle VirtualBox
- Install the library using command `sudo apt-get install clang+`

3. OBJECTIVES

1. Review the source code
2. Create additional variables
3. Implement ATLAS and BLISS

4. SOURCE FILES

I. Request.h

The ``Request`` header file encompasses information about requests, including their type, address, and arrival, and departure times. It features various constructor overloads to accommodate different types of requests, allowing for flexibility in initialization. The class includes member variables like ``is_first_command``, ``addr``, and ``coreid``, along with an enumeration ``Type`` defining request types such as read, write, and refresh.

Additionally, it supports callback functionality through the ``callback`` member variable, enabling the execution of custom actions upon request completion. This file is a great foundation for the rest of ramulator in creating requests based on the trace files

```

class Request
{
public:
    bool is_first_command;
    long addr;
    // long addr_row;
    vector<int> addr_vec;
    // specify which core this request sent from, for virtual address translation
    int coreid;

    enum class Type
    {
        READ,
        WRITE,
        REFRESH,
        POWERDOWN,
        SELFREFRESH,
        EXTENSION,
        MAX
    } type;

    long arrive = -1;
    long depart = -1;
    function<void(Request&)> callback; // call back with more info

    Request(long addr, Type type, int coreid = 0)
        : is_first_command(true), addr(addr), coreid(coreid), type(type),
        callback([](Request& req){}) {}

    Request(long addr, Type type, function<void(Request&)> callback, int coreid = 0)
        : is_first_command(true), addr(addr), coreid(coreid), type(type), callback(callback) {}

    Request(vector<int>& addr_vec, Type type, function<void(Request&)> callback, int coreid = 0)
        : is_first_command(true), addr_vec(addr_vec), coreid(coreid), type(type), callback(callback) {}

    Request()
        : is_first_command(true), coreid(0) {}
};

```

II. Scheduler.h

This Scheduler header file defines a memory scheduler class with public and private member functions. It declares a `typedef` for an iterator type `ReqIter` to simplify the manipulation of a list of `Request` objects. The core of the class is an array of lambda functions stored in a `compare` member variable. These lambdas implement various scheduling algorithms like First Come First Serve (FCFS), First Ready First Come First Serve, FRFCFS with a capacity limit, and FRFCFS with priority given to hits in the row buffer.

Each lambda takes two iterators pointing to `Request` objects and returns the iterator of the request that should be scheduled next based on the specific scheduling policy. This header file streamlines the processing for all the requests at play.

```

typedef list<Request>::iterator ReqIter;
function<ReqIter(ReqIter, ReqIter)> compare[int(Type::MAX)] = {
    // FCFS
    [this] (ReqIter req1, ReqIter req2) {
        if (req1->arrive <= req2->arrive) return req1;
        return req2;},

    // FRFCFS
    [this] (ReqIter req1, ReqIter req2) {
        bool ready1 = this->ctrl->is_ready(req1);
        bool ready2 = this->ctrl->is_ready(req2);

        if (ready1 ^ ready2) {
            if (ready1) return req1;
            return req2;
        }

        if (req1->arrive <= req2->arrive) return req1;
        return req2;},

    // FRFCFS_CAP
    [this] (ReqIter req1, ReqIter req2) {
        bool ready1 = this->ctrl->is_ready(req1);
        bool ready2 = this->ctrl->is_ready(req2);

        ready1 = ready1 && (this->ctrl->rowtable->get_hits(req1->addr_vec) <= this->cap);
        ready2 = ready2 && (this->ctrl->rowtable->get_hits(req2->addr_vec) <= this->cap);

        if (ready1 ^ ready2) {
            if (ready1) return req1;
            return req2;
        }

        if (req1->arrive <= req2->arrive) return req1;
        return req2;},

    // FRFCFS_PriorHit
    [this] (ReqIter req1, ReqIter req2) {
        bool ready1 = this->ctrl->is_ready(req1) && this->ctrl->is_row_hit(req1);
        bool ready2 = this->ctrl->is_ready(req2) && this->ctrl->is_row_hit(req2);

```

III. Controller.h

The controller header file includes functions for handling command execution, checking readiness, and row buffer status. The `execute_command` function manages the execution of commands associated with requests. It determines whether a request is complete and handles read and write operations accordingly, updating completion times and serving requests as needed. The `is_ready` functions check if a request is ready for execution based on the current command or specific command types and address vectors.

Similarly, the ``is_row_hit`` and ``is_row_open`` functions determine if a request's address is present in the row buffer and if the row buffer is open, respectively. These functionalities collectively enable the memory controller to efficiently manage memory operations and optimize access patterns

```

        queue->q.erase(req);
    }

    bool is_ready(list<Request>::iterator req)
    {
        typename T::Command cmd = get_first_cmd(req);
        return channel->check(cmd, req->addr_vec.data(), clk);
    }

    bool is_ready(typename T::Command cmd, const vector<int>& addr_vec)
    {
        return channel->check(cmd, addr_vec.data(), clk);
    }

    bool is_row_hit(list<Request>::iterator req)
    {
        // cmd must be decided by the request type, not the first cmd
        typename T::Command cmd = channel->spec->translate[int(req->type)];
        return channel->check_row_hit(cmd, req->addr_vec.data());
    }

    bool is_row_hit(typename T::Command cmd, const vector<int>& addr_vec)
    {
        return channel->check_row_hit(cmd, addr_vec.data());
    }

    bool is_row_open(list<Request>::iterator req)
    {
        // cmd must be decided by the request type, not the first cmd
        typename T::Command cmd = channel->spec->translate[int(req->type)];
        return channel->check_row_open(cmd, req->addr_vec.data());
    }

    bool is_row_open(typename T::Command cmd, const vector<int>& addr_vec)
    {
        return channel->check_row_open(cmd, addr_vec.data());
    }

```


5. BRIEF ON ATLAS & BLISS

I. ATLAS –

The ATLAS algorithm has two main rules for efficient memory management -

Rule 1 focuses on how each memory controller prioritizes requests: it gives attention to long-waiting requests, prioritizes threads based on a performance measure (LAS), emphasizes hitting specific memory areas efficiently, and handles older requests before newer ones.

Rule 2 outlines the coordination process at the end of a timeframe (quantum): each Memory Controller communicates the recent service of threads to a central controller

II. BLISS –

The memory controller keeps track of the last scheduled application ID and the number of requests served from that application. If the number of requests served exceeds the threshold, the application is blacklisted. Periodic clearing is done to reset the blacklist information. We will integrate these functions into our memory controller logic

6. RUNNING THE BUILD

- We first make the build of the entire code using: `make -j3`
- Next thing we run the following line which indicates the configuration, mode, and stats of the built code: `./ramulator configs/DDR3-config.cfg --mode=dram --stats my_output.txt dram.trace`
- Note: for changing configurations we use `DDR4-config.cfg` instead

7. IMPLEMENTATION

The Scheduler header file includes functions for handling the way a request is dealt with, and most of the changes to the request and controller function did not seem to work properly and/or give us the demanded results.

To implement the functions of ATLAS and BLISS, we do the following:

1. Create the following variables which help us to perform the private functions

- A Global counter
- Core ID tracker
- Streak counter
- CPU cycles
- Boolean arrays to keep track of each sub-request

```
//Global Counter
int count = 0;

//BLISS variables
// PrevPID for previous Core ID,
// oldestStreak for most times a coreID appears,
// Mark for array of boolean representation
int PrevPID = 0;
int oldestStreak = 0;
bool B_mark[40];

//ATLAS variables
// Mark the array of boolean representation
// CPU_cycles to keep track of current cpu
// Threshold value based on arrival times
bool A_mark[40];
long CPU_cycles = this->ctrl->clk;
long CPU_threshold = 200;
```

2. Initialize the array values to reflect which sub-request is handled and how

I. ATLAS

The array is initialized on the basis of when the request is arriving and whether it is arriving before the threshold value

```
if(type == Type::ATLAS){
    if (!q.size())
        return q.end();

    count = 0;
    auto head = q.begin();
    for(auto itr = next(q.begin(), 1); itr != q.end(); itr++){
        A_mark[count] = ( abs(CPU_cycles - itr->arrive) > CPU_threshold );
        count++;
    }

    count = 1;
    for (auto itr = next(q.begin(), 1); itr != q.end(); itr++)
        head = compare[int(type)](head, itr);
    count++;

    return head;
}
```

II. BLISS

The array is initialized on the basis of the streak on the CoreID and whether the sub-request doesn't overflow the threshold value. If it's under the limit, the array element is true else false(default).

```
if(type == Type::BLISS){
    if (!q.size())
        return q.end();

    count = 0;
    auto head = q.begin();
    for(auto itr = next(q.begin(), 1); itr != q.end(); itr++){
        if(itr->coreid == PrevPID && oldestStreak < cap){
            B_mark[count] = true;
            oldestStreak += 1;
        }
        else if(itr->coreid == PrevPID && oldestStreak == cap){
            B_mark[count] = true;
            oldestStreak = 1;
        }
        else{
            oldestStreak = 1;
        }
        PrevPID = itr->coreid;
        count++;
    }

    count = 1;
    for (auto itr = next(q.begin(), 1); itr != q.end(); itr++)
        head = compare[int(type)](head, itr);
        count++;

    return head;
}
```

3. Implement the private function of the algorithms

I. ATLAS

The array is initialized on the basis of the streak on the CoreID and whether the sub-request doesn't overflow the threshold value. If it's under the limit, the array element is true else false(default).

```
// ATLAS
[this] (ReqIter req1, ReqIter req2) {
    bool marked1 = A_mark[count-1];
    bool marked2 = A_mark[count];
    if (marked1 ^ marked2){
        if (marked1) return req1;
        else return req2;
    }

    bool hit1 = this->ctrl->is_row_hit(req1);
    bool hit2 = this->ctrl->is_row_hit(req2);
    if (hit1 ^ hit2){
        if (hit1) return req1;
        else return req2;
    }

    if (req1->arrive <= req2->arrive) return req1;
    else return req2;
},
```

II. BLISS

The request comparator compares the requests on the basis of the array value of that specific element if it is not fulfilled, in the case where they match we check if the row hit occurs and compare them accordingly. In the worst case, compare them based on the arrival time.

```
// BLISS
[this] (ReqIter req1, ReqIter req2) {
    bool marked1 = B_mark[count-1] != 1;    //Mark based on streak and cap
    bool marked2 = B_mark[count] != 1;
    if (marked1 ^ marked2){
        if (marked1 != true) return req1;
        else return req2;
    }

    bool hit1 = this->ctrl->is_row_hit(req1);
    bool hit2 = this->ctrl->is_row_hit(req2);
    if (hit1 ^ hit2){
        if (hit1) return req1;
        else return req2;
    }

    if (req1->arrive <= req2->arrive) return req1;
    else return req2;
}
```

8. DRAM SCHEDULING RESULTS

Configuration: DRAM-3

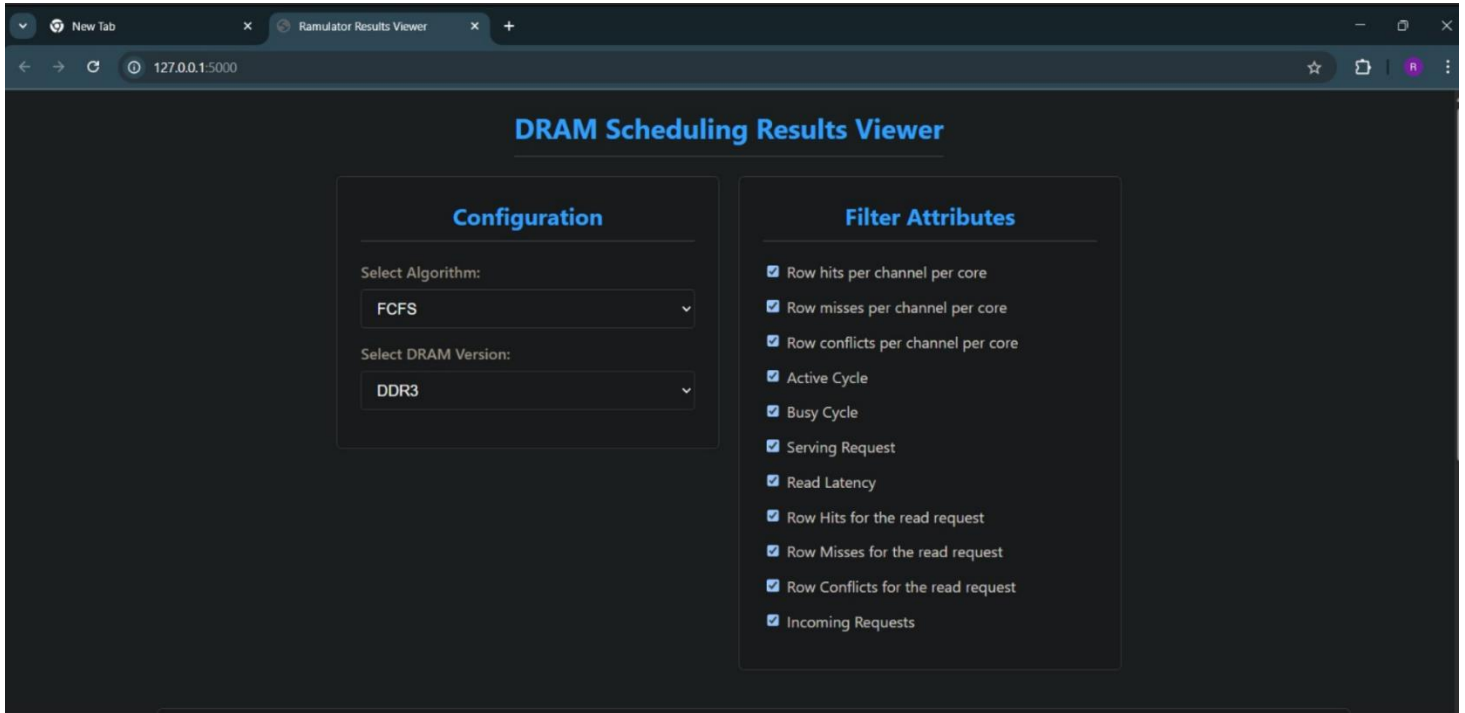
Attribute	FCFS	FRFCFS	FRFCFS WITH PRIORITY	FRFRCS WITH CAP	BLISS	ATLAS
Row hit per channel per core	33	29	33	29	35	35
Row misses per channel per core	6	12	11	12	6	8
Row conflicts per channel per core	65	63	60	63	63	61
Active Cycle	1359	918	852	918	1354	1336
Busy Cycle	1359	918	852	918	1354	1336
Serving Request	2906	3820	3710	3820	2744	2957
Read Latency	508.883117	275.701299	265.714286	275.701299	523.610390	508.649351
Row Hits for the read request	33	27	31	27	33	33
Row Misses for the read request	6	9	6	9	6	7
Row Conflicts for the read Request	38	41	40	41	38	37
Incoming Requests	104	104	104	104	104	104

Configuration: DRAM-4

Attribute	FCFS	FRFCFS	FRFCFS WITH PRIORITY	FRFRCS WITH CAP	BLISS	ATLAS
Row hit per channel per core	33	30	30	30	35	35
Row misses per channel per core	12	18	17	18	12	12
Row conflicts per channel per core	59	56	57	56	57	57
Active Cycle	1692	682	698	682	1681	1694
Busy Cycle	1692	682	698	682	1681	1694
Serving Request	3801	5538	5078	5538	3730	4992
Read Latency	601.662338	232.038961	228.63634	232.038961	610.675325	634.285714
Row Hits for the read request	33	28	28	28	33	33
Row Misses for the read request	10	13	13	13	10	10
Row Conflicts for the read Request	34	36	36	36	34	34
Incoming Requests	104	104	104	104	104	104

OUTPUT:-

1. ATLAS



Results	
Attribute	Value
Row hits per channel per core	33
Row misses per channel per core	6
Row conflicts per channel per core	65
Active Cycle	1359
Busy Cycle	1359
Serving Request	2906
Read Latency	508.883117
Row Hits for the read request	33
Row Misses for the read request	6
Row Conflicts for the read request	38
Incoming Requests	104

2. BLISS

DRAM Scheduling Results Viewer

Configuration

Select Algorithm:

ATLAS

Select DRAM Version:

DDR4

Filter Attributes

- ☒ Row hits per channel per core
- ☒ Row misses per channel per core
- ☒ Row conflicts per channel per core
- ☒ Active Cycle
- ☒ Busy Cycle
- ☒ Serving Request
- ☒ Read Latency
- ☒ Row Hits for the read request
- ☒ Row Misses for the read request
- ☒ Row Conflicts for the read request
- ☒ Incoming Requests

Results

Attribute	Value
Row hits per channel per core	35
Row misses per channel per core	12
Row conflicts per channel per core	57
Active Cycle	1694
Busy Cycle	1694
Serving Request	4992
Read Latency	634.285714
Row Hits for the read request	33
Row Misses for the read request	10
Row Conflicts for the read request	34
Incoming Requests	104

CONCLUSIONS:-

- The choice of DRAM scheduling algorithm has a profound impact on memory performance, significantly affecting latency and resource utilization (hits, conflicts, cycles).
- Simple FCFS scheduling is highly inefficient as it fails to exploit DRAM row buffer locality.
- Locality-aware policies like FRFCFS provide substantial latency reductions. Explicitly prioritizing row hits (FRFCFS_PriorHit) yields the lowest average latency in our tests. Row-hit capping (FRFCFS_Cap) showed little difference from FRFCFS on this workload but might improve fairness on others.
- More complex algorithms like BLISS and ATLAS demonstrate different strengths. BLISS showed good performance, possibly managing bank conflicts effectively. ATLAS provides a balance, likely trading some raw latency for adaptivity or fairness.
- Performance characteristics persist across DRAM generations (DDR3 vs. DDR4), but absolute values change due to different timing parameters. Faster clock speeds don't always guarantee proportionally lower latency for all algorithms, as internal timings and scheduling interactions play a major role. BLISS/ATLAS showed higher average latency on DDR4 for this trace, warranting further investigation.
- The optimal scheduler depends on the specific goals: minimizing average latency (FRFCFS_PriorHit), maximizing parallelism/throughput (potentially BLISS), or achieving a balanced, adaptive performance (potentially ATLAS).

REFERENCES

1. Y. Kim, D. Han, O. Mutlu and M. Harchol-Balter, "**ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers**," HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, Bangalore, India, 2010, pp. 1-12, doi: 10.1109/HPCA.2010.5416658. keywords: {Scheduling algorithm;Control systems;Throughput;Multicore processing;Queueing analysis;Bandwidth;Feeds;Algorithm design and analysis;Programmable control;Adaptive control}
2. L. Subramanian, D. Lee, V. Seshadri, H. Rastogi and O. Mutlu, "**BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling**," in IEEE Transactions on Parallel and Distributed Systems, vol. 27, no. 10, pp. 3071-3087, 1 Oct. 2016, doi: 10.1109/TPDS.2016.2526003. keywords: {Interference;Complexity theory;Hardware;System performance;Multicore processing;Random access memory;Scheduling},

[GitHub Link](#)