CS550 ADVANCED OPERATING SYSTEMS

PROJECT REPORT

A SCALABLE AND HIGH PERFORMANCE SCHEDULER FOR MULTI-CORE SYSTEMS

Kevin Tchouate

INDEX

Abstract

With computers having more and more cores but their memory and I/O capacity not following at the same speed, computer scientists are facing the problem of memory walls. Thus, accessing large amounts of data becomes a bottleneck for many applications [1]. This raises the problem of memory bandwidth and scheduling in the overall system performance. One good policy to use for memory scheduling algorithms to have a good system throughput is the Adaptive per-Thread Least-Attained-Service memory scheduling (ATLAS) [2].

1. Introduction

1.1 Project name

ATLAS (Adaptive per-Thread Least-Attained-Service memory scheduling).

1.2 Goal

The goal of this project is to address the problem of throughput in a multi-core system by designing and implementing a scalable and high performance memory scheduling without significant coordination between (MC) memory controllers.

1. 3 Description

By ranking each thread by the least attained service from the memory controllers since execution began, ATLAS prioritizes the threads that have attained the least service when making scheduling decisions. This implementation of thread prioritization reduces significantly the time stalling of cores and significantly improves system throughput, as a result, improves system throughput, as shown by Kim et al. [2].

2. Problem statement

Memory latency significantly reduces the performance of our modern systems. When an instruction misses in the last-level cache and needs to access memory, the processor soon stalls once its instruction window becomes full [3]. This problem is harsher in multithreaded systems that share memory since each thread request experiences additional delay. To reduce the impact of this problem memory scheduling needs to minimize the time each thread spends stalling.

Some scheduling algorithms do not require coordination in multicore systems, that result in poor system performance (throughput). Other scheduling algorithms provide fairly good system throughput but require some coordination to achieve this performance on multiple memory controller systems. The previously best algorithm in terms of performance is parallelism-aware batch scheduling (PAR-BS) [4], but it is not scalable, It requires a lot of information exchange between different MCs. This incapacity to find a compromise between overall good performance and scalability it's what I want to address in this project.

3. Existing solution

3.1 PAR-BS

There have been many developments of scheduling algorithms. As I mentioned previously parallelism-aware batch scheduling (PAR-BS) is one of those, it attempts to minimize the average stall time by scheduling the thread with the shortest stall time first at a given time during execution.

3.2 FCFS

First-come-first-serve (FCFS) is another scheduling algorithm, it favors memory intensive threads since they appear naturally older to the memory controller as they arrive more frequently than non intensive threads.

3.3 STFM

Stall-time fair memory scheduler (STFM) prioritizes the most slowed down thread by estimating the slowdown of each thread and comparing it to when it is run alone.

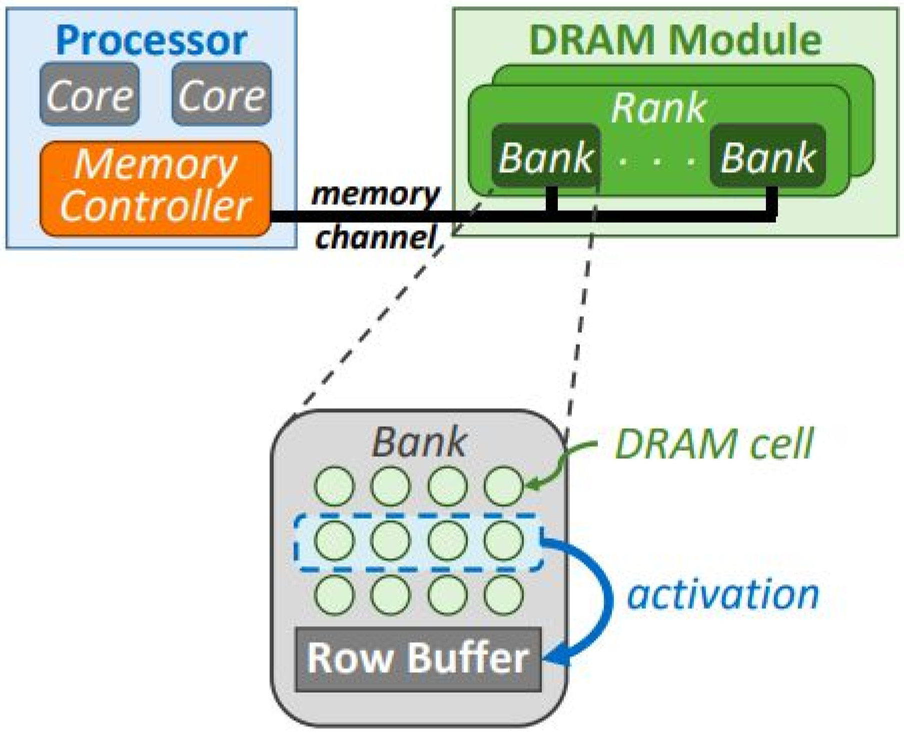
4. Solution

In this project, I will implement and evaluate a memory scheduling policy ATLAS. To achieve my goal, I will extend the architectural memory simulator Ramulator [5], to model the memory scheduling policy. By ranking each thread by the least attained service from the memory controllers since execution began, ATLAS prioritizes the threads that have attained the least service when making scheduling decisions. This implementation of thread prioritization reduces significantly the time stalling of cores and significantly improves system throughput, as a result, improves system throughput, as shown by Kim et al. [2]. The proposed solution comprises the:

- Implementation of ATLAS (C++ code)
- Evaluation of the implementation
- Evaluation of complex applications using my implementation

5. Implementation and results

5.1 Brief DRAM overview



Source : https://people.inf.ethz.ch/omutlu/pub/Workload-DRAM-Interaction-Analysis_sigmetrics19-talk.pdf

A Dynamic Random Access Memory (DRAM) is one of the most used memory in modern desktop and laptop computers today. In the DRAM, data is stored in data cells. Cell stores charge in a capacitor to represent a one-bit data value. When it receives a memory request it decodes the address of the data of interest to access it. Before reading or writing, a full row of cells must be activated (opened). Data cells are grouped into banks and banks are grouped into ranks. Processors access the memory via a memory controller.

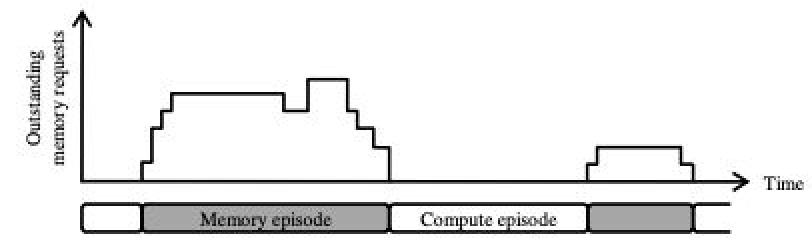
A processor sends memory requests to the memory controller. The memory controller then manages and schedules the treatment of the requests of each processor. The

Memory controller issues commands to the memory through a memory channel. Many systems have multiple independent channels. Banks can operate in parallel and many banks share one channel.

5.2 ATLAS mechanism overview

During its life cycle, a thread recursively switches between two episodes as shown in the figure :

Memory vs. compute episodes in a thread's execution time



Source : ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. Yoongu Kim, Dongsu Han, Onur Mutlu, Mor Harchol-Balter, Carnegie Mellon University.

1) Memory episode, where the thread is waiting for at least one memory request.

2) Compute episode, where there are no memory requests by the thread.

Instruction throughput (Instructions Per Cycle) is high during the compute episode, but low during the memory episode. When a thread is in its memory episode, it is waiting for at least one memory request to be serviced and is likely to be stalled, hence degrading core utilization. Therefore, to maximize system throughput, we have to minimize the time threads spend in their memory episodes.

Attained service is defined as the total amount of memory service (in cycles) that a memory episode has received since it started. The key point that we exploit is that the attained service of a memory episode is an excellent predictor of the remaining length (service) of the memory episode, if the memory episode lengths follow a distribution with decreasing hazard rate (DHR). Since the Memory Controller (MC) knows is the attained service of an episode, we can relatively easily rank the threads based on the service they have attained and prioritize threads that have attained the least service.

Favoring threads whose memory episodes will end soonest will certainly maximize system throughput in the short term. However it does not take longer-term thread behavior into account. To take into account both short-term and long-term optimizations, we generalize the notion of Least Attained Service (LAS) to include a larger time interval than just a single episode. The ATLAS (Adaptive per-Thread LAS) memory controller divides time into large but fixed-length intervals called quanta. During each quantum, the memory controller tracks each thread's total attained service for that quantum. At the beginning of the next quantum, the memory controller ranks the threads based on their attained service in the past, weighting the attained service in the recent past quanta more heavily than the attained service in older quanta. Specifically, for any given thread, we define:

$$TotalAS_i = \alpha TotalAS_{i-1} + (1-\alpha)AS_i$$

AS_i: Attained service during quantum i alone (reset at the beginning of a quantum)
$TotalAS_i$: Total attained service summed over all quanta up to the end of quantum i (reset at a context switch)

Here α is a parameter 0 ≤ α < 1, where lower α indicates a stronger bias towards the most recent quantum. During the course of quantum i + 1, the controller uses the above thread ranking based on TotalASi, favoring threads with lower TotalASi, that is lower total attained service. When there are multiple memory controllers, the attained service of a thread is the sum of the individual service it has received from each MC. As a result, at the beginning of a new quantum, each MC needs to coordinate with other MCs to determine a consistent thread ranking across all MCs. Controllers achieve this coordination by sending the local attained service of each thread to a centralized agent in the on-chip network that computes the global attained service of each thread, forms a ranking based on least-attained-service, and broadcasts the ranking back to each controller.

ATLAS scheduling policy guarantees a starvation freedom and maximizes system throughput within a quantum by minimizing the time spent in memory episodes. In addition, it ensures that a thread's concurrent requests are serviced in parallel in the memory banks instead of being serialized due to inter-thread interference, thereby preserving the bank-level parallelism of each thread.

5.3 Ramulator overview

Ramulator is a fast and cycle-accurate DRAM simulator that supports a wide area of commercial, as well as academic, DRAM standards: DDR3 (2007), DDR4 (2012), LPDDR3 (2012), LPDDR4 (2014), GDDR5 (2009), WIO (2011), WIO2 (2014), HBM (2013), SALP [6], TL-DRAM [7], RowClone [8], DSARP [9].

It allows three different usage modes :

- ❏ Memory Trace Driven: Ramulator directly reads memory traces from a file, and simulates only the DRAM subsystem. Each line in the trace file represents a memory request, with the hexadecimal address followed by 'R' or 'W' for read or write.
    - 0x12345680 R
    - 0x4cbd56c0 W
    - ...
- ❏ CPU Trace Driven: Ramulator directly reads instruction traces from a file, and simulates a simplified model of a "core" that generates memory requests to the DRAM subsystem. Each line in the trace file represents a memory request, and can have one of the following two formats.
    - <num-cpuinst> <addr-read>: For a line with two tokens, the first token represents the number of CPU (i.e., non-memory) instructions before the memory request, and the second token is the decimal address of a read.

    - <num-cpuinst> <addr-read> <addr-writeback>: For a line with three tokens, the third token is the decimal address of the writeback request, which is the dirty cache-line eviction caused by the read request before it.

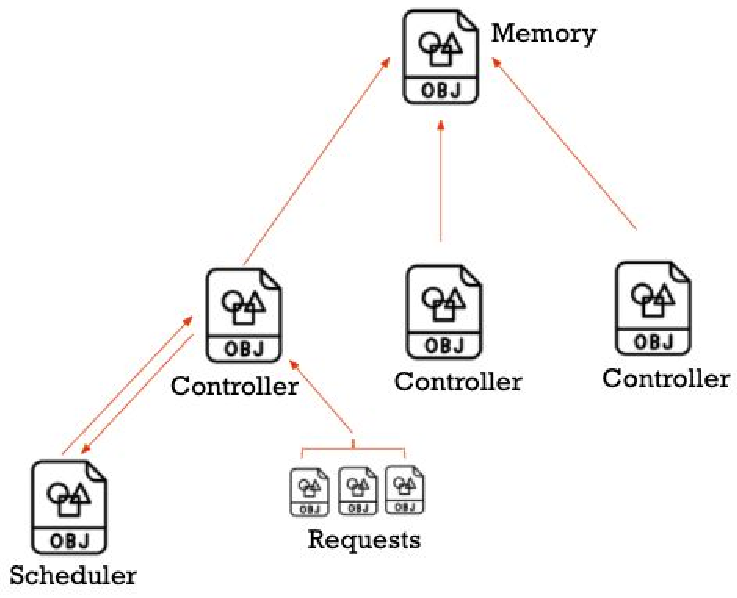
- ❏ gem5 Driven: Ramulator runs as part of a full-system simulator (gem5 [10]), from which it receives memory requests as they are generated.

For some of the DRAM standards, Ramulator is also capable of reporting power consumption by relying on either VAMPIRE [11] or DRAMPower [12] as the backend.

5.4 ATLAS implementation

Ramulator helped me to simulate a DRAM, and to do so, I had to extend different parts of the system. The modules I modified to implement the ATLAS were the Memory, the controllers, and the schedulers. On the graph, we can see the different interactions between the modules. We can see that the memory controllers are represented by the controller and each memory controller is associated to a scheduler. Memory requests arrive to the memory and the memory sends each memory request to the right controller.

ATLAS implementation



Each controller increments the attained service of each core using the coreid. At the end of the quantum each memory controller sends each thread's local attained service to the memory (the central meta controller). The meta controller maintains a table with all the old total attained services and uses received attained services to compute the new total AS value. The meta controller ranks the cores and then the ranking is broadcasted to all controllers.

5.5 Tests and results

The quantum was set at 10000000 cycles, alpha to 0.875 and T = 100K.

I used a DDR4, with the following configuration : early_exit = off, expected_limit_insts = 1M, warmup_insts = 0, and cache = no.

To run a test the following command was used :

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

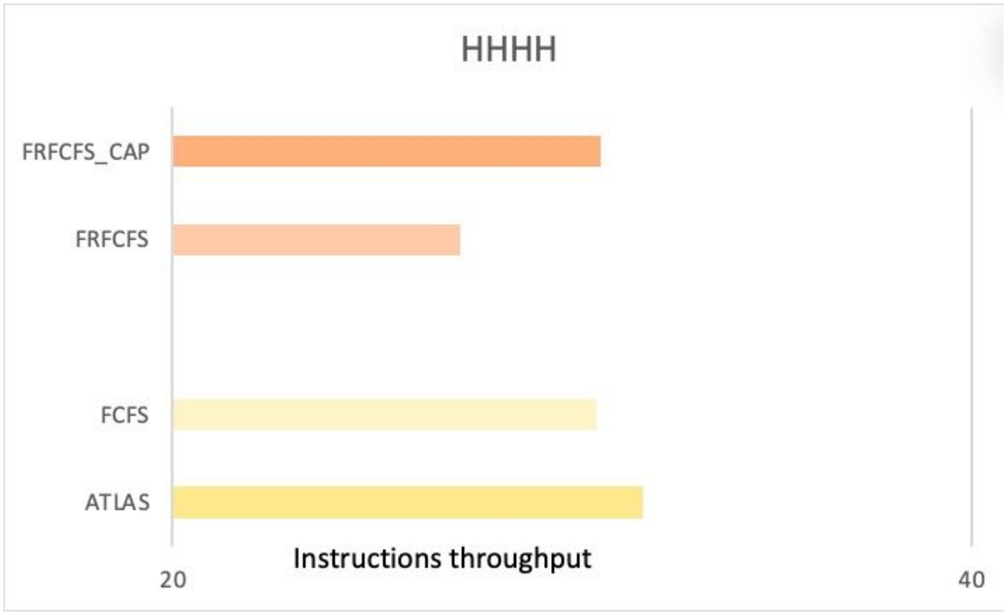
I used the default cpu.trace as high-mem-intensity.trace to simulate a more memory intensive process than the process simulated by low-mem-intensity.trace which is a cpu-trace with fewer lines. I run the test on three different combinations of high and low memory intensity:

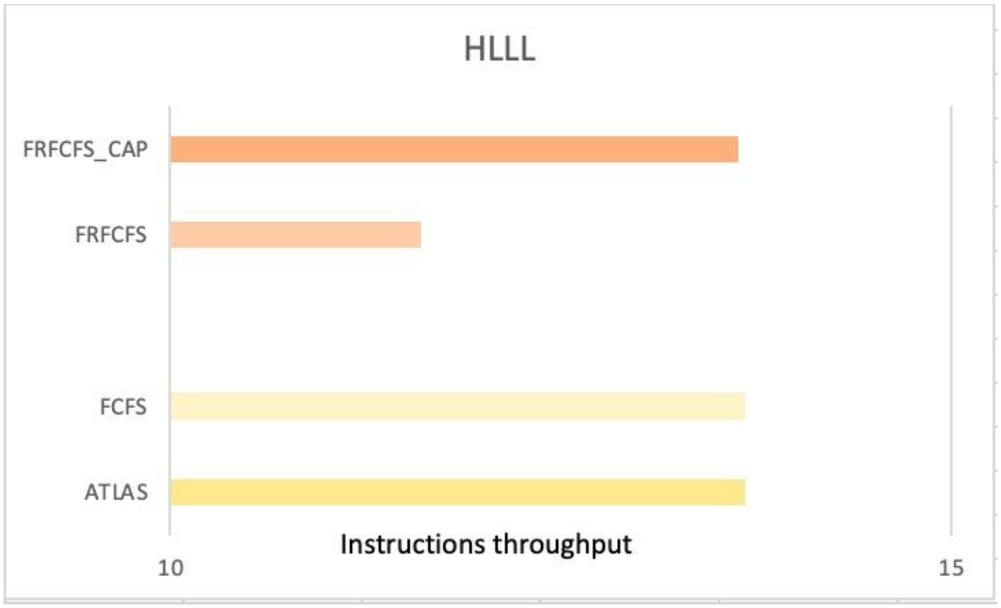
- HLLL : high-mem-intensity.trace low-mem-intensity.trace low-mem-intensity.trace low-mem-intensity.trace.
- HHLL: high-mem-intensity.trace high-mem-intensity.trace low-mem-intensity.trace low-mem-intensity.trace.
- HHHH : high-mem-intensity.trace high-mem-intensity.trace high-mem-intensity.trace high-mem-intensity.trace.

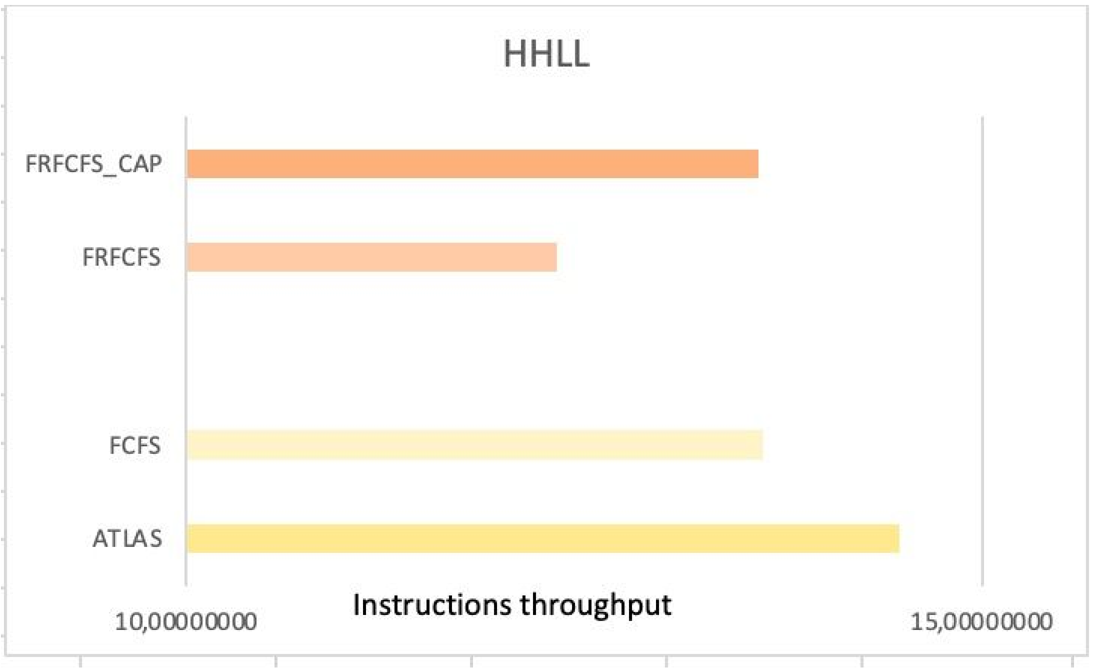
Using the mentioned configurations, I evaluated the instruction throughput given by the formula :

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

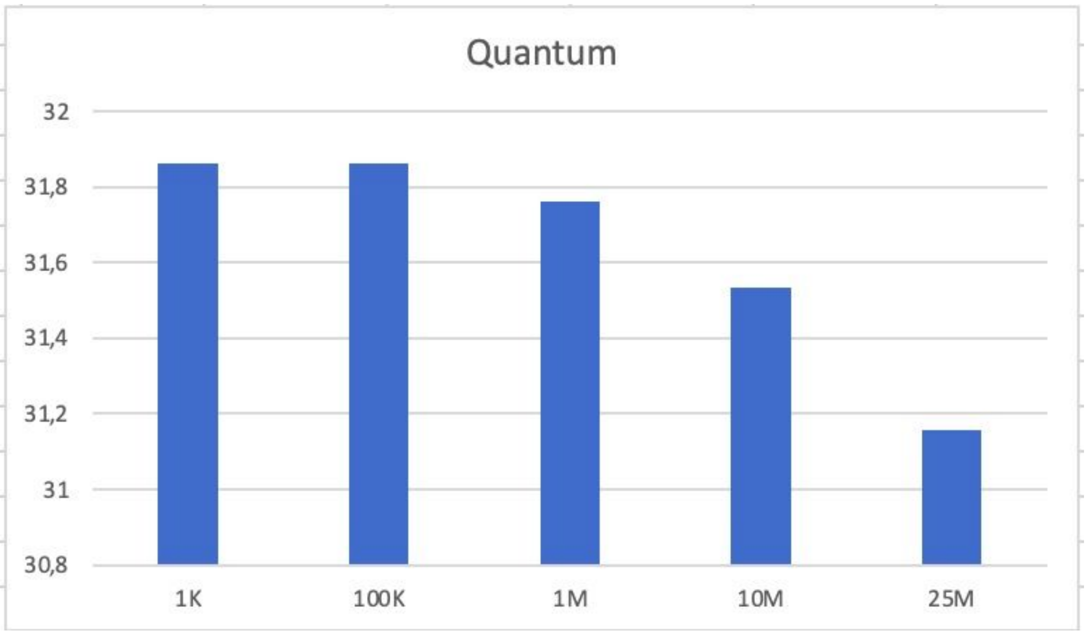
I obtained the following results :







I was also interested in studying the impact of the quantum on the instructions throughput, so I made it vary and studied its effect as shown by the following plot :



6. Analysis and conclusion

As we can see from the plots of various memory workloads, ATLAS usually gives the best performance. That is especially true for memory intensive workloads as shown by HHHH combination, where ATLAS gives the best performance.
However, the performance of ATLAS highly depends on the choice of its parameter (T, alpha and quantum). The plot of the instruction throughput against the quantum shows that the performance of ATLAS decreases as the quantum increases for non intensive memory workloads. We have to note that our tests were not really carried on memory intensive processes since I used cpu.trace as my base memory workload. On the other hand, that was enough to get the sense of the power of ATLAS.

This project was very interesting and helpful in many aspects. It allows me to get my hands on DRAM architecture and to better understand the computer as a whole. Furthermore, through this project, I got the opportunity to learn a new programming language (C++) and overcome the challenges that come with a programming project. In addition, I got a chance to manipulate ramulator and to better understand the impact of the DRAM on our computers. To continue, this project allowed me to get to know various scheduling policies, their strengths and their weaknesses; and I got to successfully implement ATLAS policy on my own.

7. Reference

[1] A. Choudhary, W. Liao, K. Gao, A. Nisar, R. Ross, R. Thakur, and R. Latham, “Scalable I/O and analytics,” J . Phys. Conf. Ser., vol. 180, p. 012048, Jul. 2009.

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

[3] T. Karkhanis and J. E. Smith. A day in the life of a data cache miss. In WMPI-2, 2002.

[4] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: En-hancing both performance and fairness of shared DRAM systems. In ISCA-36, 2008.

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

[6] Kim et al. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. ISCA 2012.

[7] Lee et al. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. HPCA 2013.

[8] Seshadri et al. RowClone: Fast and Energy-Efficient In-DRAM Bulk Data Copy and Initialization. MICRO 2013.

[9] Chang et al. Improving DRAM Performance by Parallelizing Refreshes with Accesses. HPCA 2014.

[10] The gem5 Simulator System.

[11] Ghose et al. What Your DRAM Power Models Are Not Telling You: Lessons from a Detailed Experimental Study. SIGMETRICS 2018.

[12] Chandrasekar et al. DRAMPower: Open-Source DRAM Power & Energy Estimation Tool. IEEE CAL 2015.