ABSTRACT

Title of Dissertation: SCALABLE AND ACCURATE MEMORY SYSTEM SIMULATION

Shang Li
Doctor of Philosophy, 2014

Dissertation directed by: Professor Bruce Jacob
Department of Electrical & Computer Engineering

Memory systems today possess more complexity than ever. On one hand, main memory technology has a much more diverse portfolio. Other than the main stream DDR DRAMs, a variety of DRAM protocols have been proliferating in certain domains. Non-Volatile Memory(NVM) also finally has commodity main memory products, introducing more heterogeneity to the main memory media. On the other hand, the scale of computer systems, from personal computers, server computers, to high performance computing systems, has been growing in response to increasing computing demand. Memory systems have to be able to keep scaling to avoid bottlenecking the whole system. However, current memory simulation works cannot accurately or efficiently model these developments, making it hard for researchers and developers to evaluate or to optimize designs for memory systems.

In this study, we attack these issues from multiple angles. First, we develop a fast and validated cycle accurate main memory simulator that can accurately model almost all existing DRAM protocols and some NVM protocols, and it can be easily

extended to support upcoming protocols as well. We showcase this simulator by conducting a thorough characterization over existing DRAM protocols and provide insights on memory system designs.

Secondly, to efficiently simulate the increasingly paralleled memory systems, we propose a lax synchronization model that allows efficient parallel DRAM simulation. We build the first ever practical parallel DRAM simulator that can speedup the simulation by up to a factor of three with single digit percentage loss in accuracy comparing to cycle accurate simulations. We also developed mitigation schemes to further improve the accuracy with no additional performance cost.

Moreover, we discuss the limitation of cycle accurate models, and explore the possibility of alternative modeling of DRAM. We propose a novel approach that converts DRAM timing simulation into a classification problem. By doing so we can make predictions on DRAM latency for each memory request upon first sight, which makes it compatible for scalable architecture simulation frameworks. We developed prototypes based on various machine learning models and they demonstrate excellent performance and accuracy results that makes them a promising alternative to cycle accurate models.

Finally, for large scale memory systems where data movement is often the performance limiting factor, we propose a set of interconnect topologies and implement them in a parallel discrete event simulation framework. We evaluate the proposed topologies through simulation and prove that their scalability and performance exceeds existing topologies with increasing system size or workloads.

SCALABLE AND ACCURATE MEMORY SYSTEM SIMULATION

by

Shang Li

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor Bruce Jacob, Chair/Advisor
Professor Donald Yeung
Professor Manoj Franklin
Professor Jeffery Hollingsworth
Professor Alan Sussman

To my mom, Chunjie Li;

and my grandparents, Liankun Li and Shuxian Wang.

Acknowledgments

First of all, I would like to thank my family, especially my mom, without whose unwavering support I will never be where I am today. I am grateful for having grown up in this family where everyone takes care of each other. I owe everything I have achieved to them.

Special thanks to my girlfriend, Ke Xie, who shared my stress and anxiety over the last two years, and countered it with happiness and joy. She also supported this research by allowing me to build a computer rig in her apartment. I look forward to the next chapter of our lives.

Next, I would like to thank my advisor Prof. Jacob, who led me into the world of computer architectures. He has always been there looking out for me, providing whatever I needed for research, and guided me along the way through my PhD years with his wisdom and vision.

I also want to thank my committee for devoting their time and efforts into my defense and dissertation and providing valuable feedbacks, especially Prof. Yeung, who was also in weekly group meetings with me and inspired a whole chapter of this dissertation.

Finally, I would like to thank my friends and colleagues, past and present, at the University of Maryland. I would like to give my best wishes to my labmates, Luyi, Meena, Brendan, Devesh, Candace, and Daniel, for their PhD career.

Table of Contents

Dedication ii

Acknowledgements iii

List of Tables vii

List of Figures viii

List of Abbreviations xii

1 Overview 1

2 Cycle Accurate Main Memory Simulation 6
- 2.1 Memory Technology Background . . . 6
  - 2.1.1 DDRx SDRAM . . . 11
  - 2.1.2 LPDDRx SDRAM . . . 13
  - 2.1.3 GDDR5 SGRAM . . . 14
  - 2.1.4 High Bandwidth Memory (HBM) . . . 15
  - 2.1.5 Hybrid Memory Cube (HMC) . . . 16
  - 2.1.6 DRAM Modeling . . . 18
- 2.2 Simulator Design & Capability . . . 18
  - 2.2.1 Simulator Design & Features . . . 18
  - 2.2.2 Bridging Architecture and Thermal Modeling . . . 22
  - 2.2.3 Thermal Models . . . 25
    - 2.2.3.1 Transient Model . . . 25
    - 2.2.3.2 Steady State Model . . . 28
    - 2.2.3.3 Thermal Model Validation . . . 29
- 2.3 Evaluation . . . 30
  - 2.3.1 Simulator Validation . . . 30
  - 2.3.2 Comparison with Existing DRAM Simulators . . . 31
- 2.4 Conclusion . . . 32

3 Performance and Power Comparison of Modern DRAM Architectures 34
3.1 Introduction . . . 34
3.2 Experiment Setup . . . 40
3.2.1 Simulation Setup . . . 40
3.3 Results . . . 42
3.3.1 Overall Performance Comparisons . . . 44
3.3.2 Access Latency Analysis . . . 48
3.3.3 Power, Energy, and Cost-Performance . . . 50
3.3.4 Row Buffer Hit Rate . . . 55
3.3.5 High Bandwidth Stress Testing . . . 56
3.4 Conclusion and Future Work . . . 59

4 Limitations of Cycle Accurate Models 61
4.1 Introduction . . . 61
4.2 Background . . . 63
4.2.1 CPU Simulation Techniques . . . 63
4.2.2 DRAM Simulation Techniques . . . 65
4.3 Empirical Study . . . 66
4.3.1 Quantifying DRAM Simulation Time . . . 68
4.3.2 Synchronization Overhead . . . 69
4.3.3 Compatibility: A Case Study of ZSim . . . 70
4.3.4 Event-based Model . . . 75
4.4 Conclusion . . . 77

5 Parallel Main Memory Simulation 78
5.1 Naive Parallel Memory Simulation . . . 78
5.1.1 Implementation . . . 78
5.1.2 Evaluation . . . 80
5.2 Multi-cycle Synchronization: MegaTick . . . 82
5.2.1 Performance Evaluation . . . 83
5.2.2 Accuracy Evaluation . . . 86
5.2.2.1 CPI Accuracy Evaluation . . . 87
5.2.2.2 LLC Accuracy Evaluation . . . 89
5.2.2.3 Memory Accuracy Evaluation . . . 90
5.2.2.4 Summary . . . 92
5.3 Accuracy Mitigation . . . 92
5.3.1 CPI Errors after Mitigation . . . 94
5.3.2 LLC Errors after Mitigation . . . 96
5.3.3 Memory Impact after Mitigation . . . 97
5.3.4 Summary . . . 98
5.4 Discussion . . . 99
5.5 Conclusion . . . 99

6 Statistical DRAM Model 100
6.1 Propositions . . . 100
6.2 Proposed Model . . . 103
6.2.1 Classification . . . 103
6.2.2 Latency Recovery . . . 105
6.2.3 Dynamic Feature Extraction . . . 106
6.2.4 Model Training . . . 109
6.2.5 Inference . . . 112
6.2.6 Other Potential Models . . . 113
6.3 Experiments & Evaluation . . . 114
6.3.1 Hyperparameters . . . 114
6.3.2 Feature Importance . . . 116
6.3.3 Accuracy . . . 116
6.3.4 Performance . . . 119
6.3.5 Multi-core Workloads . . . 121
6.4 Discussion . . . 125
6.4.1 Implications of Using Fewer Features . . . 125
6.4.2 Interface & Integration . . . 127
6.5 Conclusion & Future Work . . . 128

7 Memory System for High Performance Computing Systems 130
7.1 Introduction . . . 130
7.2 Background and Related Work . . . 133
7.2.1 Existing Interconnect Topologies . . . 135
7.3 Fishnet and Fishnet-Lite Topologies . . . 139
7.3.1 Topology Construction . . . 139
7.3.2 Routing Algorithm . . . 141
7.3.3 Valiant Random Routing Algorithm (VAL) . . . 142
7.3.4 Adaptive Routing . . . 143
7.3.5 Deadlock Avoidance . . . 145
7.4 Experiment Setup . . . 146
7.4.1 Simulation Environments . . . 146
7.4.2 Network parameters . . . 148
7.4.3 Workloads . . . 149
7.5 Synthetic Cycle-Accurate Simulation Results . . . 151
7.6 Detailed Simulation Results . . . 153
7.6.1 Link Bandwidth . . . 154
7.6.2 Link Latency . . . 159
7.6.3 Performance Scaling Efficiency . . . 160
7.6.4 Stress Test . . . 162
7.7 Conclusion . . . 164

8 Conclusions 166

Bibliography 168

List of Tables

2.1 Supported Protocols & Features of DRAMsim3 22

3.1 Gem5 Simulation Setup 40
3.2 DRAM Parameters 41

4.1 Simulation Setup 66

6.1 Features with Descriptions 110
6.2 Hyperparameters of the decision tree model. 114
6.3 Hyperparameter Values of Best Accuracy 115
6.4 Randomly mixed multi-workloads. 122

7.1 Simulated Configurations and Workloads 147

List of Figures

2.1 Stylized DRAM internals, showing the importance of the data buffer between DRAM core and I/O subsystem. Increasing the size of this buffer, i.e., the fetch width to/from the core, has enabled speed increases in the I/O subsystem that do not require commensurate speedups in the core. . . . 6
2.2 DRAM read timing, with values typical for today. The burst delivery time is not drawn to scale: it can be a *very* small fraction of the overall latency. *Note:* though precharge is shown as the first step, in practice it is performed at the end of each request to hide its overhead as much as possible, leaving the array in a precharged state for the next request. . . . 7
2.3 DDR SDRAM Read timing. . . . 11
2.4 DDRx DIMM Bus . . . 12
2.5 LPDDR Bus . . . 12
2.6 GDDR Bus . . . 14
2.7 HBM Interface . . . 16
2.8 HMC Interface . . . 17
2.9 Software Architecture of DRAMsim3 . . . 19
2.10 An example of how DRAM internal structures can be rearranged. (a) shows column 8–10 in DRAM controller's view (b) show the corresponding physical columns internally in DRAM subarrays. . . . 24
2.11 Illustration of (a) the 3D DRAM, (b) memory module with 2D DRAM devices and (c) layers constituting one DRAM die . . . 26
2.12 Illustration of the thermal model . . . 27
2.13 (a) The original power profile, (b) the transient result for for the peak temperature, (c) the temperature profile at 1s calculated using our thermal model and (d) the temperature profile at 1s calculated using the FEM method . . . 29
2.14 Simulation time comparison for 10 million random & stream requests of different DRAM simulators . . . 32
2.15 Simulated cycles comparison for 10 million random & stream requests of different DRAM simulators . . . 33

3.1 **Top**: as observed by Srinivasan [1], when plotting system behavior as latency per request vs. actual bandwidth usage (or requests per unit time). . . . 39
3.2 CPI breakdown for each benchmark. Note that we use a different y-axis scale for GUPS. Each stack from top to bottom are *stalls due to bandwidth, stalls due to latency, CPU execution overlapped with memory*, and *CPU execution*. . . . 43
3.3 Average access latency breakdown. Each stack from top to bottom are *Data Burst Time, Row Access Time, Refresh Time, Column Access Time* and *Queuing Delay*. . . . 47
3.4 Access latency distribution for GUPS. Dashed lines and annotation show the average access latency. . . . 49
3.5 Average power and energy. The top 2 figure and the lower left one show the average power breakdown of 3 benchmarks. The lower right one shows the energy breakdown of GUPS benchmark. . . . 51
3.6 Average power vs CPI. Y-axis in each row has the same scale. Legends are split into 2 sets but apply to all sub-graphs. . . . 52
3.7 Row buffer hit rate. HMC is not shown here because it uses close page policy. GUPS has very few "lucky" row buffer hits. . . . 56
3.8 Tabletoy (random), left; STREAM (sequential), right. 64 bytes per request. . . . 58

4.1 Simulation time breakdown, CPU vs DRAM. Upper graph is cycle-accurate out-of-order CPU model with cycle-accurate DRAM model. Lower graph is modern non-cycle-accurate CPU model. The DRAM simulators are the same in both graph. . . . 62
4.2 Absolute simulation time breakdown of Timing CPU with 1, 2, and 4 channels of cycle-accurate DDR4. The bottom component of each bar represents the CPU simulation time and the top component is the DRAM simulation time. . . . 69
4.3 DRAM latency and overall latency reported by Gem5 and ZSim. . . . 70
4.4 ZSim 2-phase memory model timeline diagram compared with real hardware/cycle accurate model. Three back-to-back memory requests (0, 1, 2) are issued to the memory model. . . . 71
4.5 Varying ZSim "minimum latency" parameter changes the benchmark reported latency, but has little to none effect on DRAM simulator. . . . 74
4.6 CPI differences of an event based model in percentage comparing to its cycle-accurate counterpart. DDR4 and HBM protocols are evaluated. . . . 76

5.1 Parallel DRAM simulator architecture. . . . 80
5.2 Simulation time using 1, 2, 4, and 8 threads. . . . 81
5.3 Cycle-accurate model (upper) vs MegaTick model (lower) . . . 82
5.4 Simulation time using MegaTick synchronization. Random(left) and stream(right) traces. . . . 84

5.5 MegaTick relative simulation time to serial cycle-accurate model ("Baseline-all") with relative CPU simulation time for each benchmark("Baseline-CPU"). 8-channel HBM. 8 threads for parallel setups. 85
5.6 CPI error comparison of MegaTick model. Cycle-accurate results are the comparison baseline(0%). 88
5.7 LLC average miss latency percentage difference comparing to cycle-accurate model. 89
5.8 Inter-arrival latency distributions density of *bwaves_r* benchmark. Mega2(top left), Mega4(top right), Mega8(bottom left), and Mega16(bottom right). 91
5.9 MegaTick and its accuracy mitigation schemes. Balanced Return will return some requests before the next MegaTick (middle graph). Proactive Return will return all requests before the next MegaTick (lower graph). 93
5.10 CPI difference comparing to cycle-accurate model using Balanced Return mitigation. Absolute average CPI errors are shown in the legend. 94
5.11 CPI difference comparing to cycle-accurate model using Proactive Return mitigation. Absolute average CPI errors are shown in the legend. 95
5.12 Balanced Return model LLC average miss latency percentage difference comparing to cycle-accurate model. 96
5.13 Proactive Return model LLC average miss latency percentage difference comparing to cycle-accurate model. 97
5.14 Inter-arrival latency distributions density of *bwaves_r* benchmark with Proactive Return mitigation. Mega2(top left), Mega4(top right), Mega8(bottom left), and Mega16(bottom right). 98

6.1 Latency density histogram for each benchmark obtained by Gem5 O3 CPU and 1-channel DDR4 DRAM. X-axis of each graph is cut off at 99 percentile latency point, the average and 90-percentile point are marked in each graph for reference. 102
6.2 Feature extraction diagram. We use one request as an example to show how the features are extracted. 108
6.3 Model Training Flow Diagram 109
6.4 Model Inference Flow Diagram 113
6.5 Feature importance in percentage for decision tree and random forest 116
6.6 Classification accuracy and average latency accuracy for decision tree model on various benchmarks. 118
6.7 Classification accuracy and average latency accuracy for random forest model on various benchmarks. 119
6.8 Simulation speed relative to cycle accurate model, y-axis is log scale. 120
6.9 Simulation speed vs number of memory requests per simulation. . . . 121
6.10 Classification accuracy and average latency accuracy for randomly mixed multi-workloads. 123

6.11 Request percentage breakdown of latency classes and their associated contention classes for randomly mixed multi-workloads. "+" classes are the contention classes apart from their base classes. 124
6.12 Classification accuracy vs average latency accuracy of an early prototype of a decision tree model. 126

7.1 A comparison of max theoretical performance, and real scores on Linpack (HPL) and Conjugate Gradients (HPCG). Source: Jack Dongarra 131
7.2 Torus 135
7.3 3-level Fattree 136
7.4 Dragonfly 137
7.5 A diameter-2 graph ($n = 10, k' = 3$) 138
7.6 Scalability of different topologies studied in this work 139
7.7 Each node, via its set of nearest neighbors, defines a unique subset of nodes that lies at a maximum of 1 hop from all other nodes in the graph. In other words, it only takes 1 hop from anywhere in the graph to reach one of the nodes in the subset. Nearest-neighbor subsets are shown in a Petersen graph for six of the graph's nodes. 140
7.8 Angelfish (bottom) and Angelfish Lite (top) networks based on a Petersen graph. 141
7.9 Example of how inappropriate adaptive routing in Fishnet-lite will cause congestion. Green tiles means low buffer usage while red tiles means high buffer usage. 144
7.10 Overview of Simulation Setup 148
7.11 Graphic illustrations of MPI workloads used in this study. Upper row: Halo(left), AllPingPong(right); Lower row: AllReduce(left), Random(right). 150
7.12 Simulations of network topologies under constant load; the MMS2 graphs are the 2-hop Moore graphs based on MMS techniques that were used to construct the Angelfish networks. 152
7.13 AllReduce workload comparison for all topology-routing combinations 155
7.14 AllPingpong workload comparison for all topology-routing combinations 156
7.15 Halo workload comparison for all topology-routing combinations . . . 157
7.16 Random workload comparison for all topology-routing combinations . 158
7.17 Averaged scaling efficiency from 50k-node to 100k-node 161
7.18 Execution slowdown of different topologies under increasing workload 162

List of Abbreviations

CPI	Cycles Per Instruction
DRAM	Dynamic Random Access Memory
DDR	Double Data Rate
DIMM	Dual In-line Memory Module
Gbps	Gigabits per second
GDDR	Graphics Double Data Rate
HBM	High Bandwidth Memory
HMC	Hybrid Memory Cube
IPC	Instructions Per Cycle
JEDEC	Joint Electron Device Engineering Council
LPDDR	Low Power Double Data Rate
Mbps	Megabits per second
PCB	Print Circuit Board
SDRAM	Synchronous Dynamic Random Access Memory
TSV	Through Silicon Via

Chapter 1: Overview

Memory systems today exhibit more complexity than ever. On one hand, main memory technology has a much more diverse portfolio. Other than the main stream DDR DRAMs, LPDDR, GDDR, and stacked DRAMs such as HBM and HMC have been proliferating not only in their specific domains, but also emerge with cross domain applications. Non-volatile memories also have commodity products in the market: Intel's Optane (previously 3DXPoint) are available in the form of DDR4 compatible DIMMs. This introduces more heterogeneity to the main memory media. On the other hand, the scale of computer systems, from personal computers, server computers, to high performance computing systems, has been increasing. The memory systems have to be able to keep scaling in order not to bottleneck the whole system. However, current memory simulation works cannot accurately or efficiently model these developments, making it hard for researchers and developers to evaluate or to optimize designs for memory systems.

In this work we address these issues from multiple angles.

First, to provide an accurate modeling tool for the diverse range of main memory technologies, we develope a fast and extendable cycle accurate main memory simulator, DRAMsim3, that can accurately model almost all existing DRAM protocols

and some NVM protocols. We extensively validated our simulator against various hardware models and measurements to ensure the simulation accuracy. DRAMsim3 also has state-of-the-art performance and features that no currently available simulators can offer. It can be easily extended to support upcoming protocols as well. We showcase this simulator's capability by conducting a thorough characterization of various existing DRAM protocols and provide insights on modern memory system designs. We introduce how we designed, implemented and validated the simulator and the discovery we made from the memory characterization study in detail in Chapter 2 and Chapter 3.

While our cycle accurate simulator offers the best performance and accuracy of its kind, due to the fundamental limit of cycle accurate model, the simulation performance still struggles to scale with the increasing channel-level parallelism of modern DRAM. To address this issue, we explore the feasibility of bring parallel simulation into the memory simulation world to gain speed. We proposed and implemented the first practical parallel memory simulator, along with a lax synchronization technique that we call MegaTick to boost parallel performance. In our simulation experiments we show our parallel simulator can run up to 3x faster than our cycle accurate simulator when simulating a 8-channel memory, with an average of 1% loss in overall accuracy. We will expand this part in Chapter 5.

Moreover, to further push the boundary of memory simulation, and to overcome the inherent limitation of cycle accurate simulation models, we explore alternative modeling techniques. We introduce the novel idea of modeling DRAM timing simulation as a classification problem, and hence solve it with a statistical

and machine learning model. We prototyped a machine learning model that can dynamically extract features from memory address streams, and it is trained with the ground truth provided by a cycle accurate simulator. The model only needs to be trained once before being used in any kind of workload, and thanks to its dynamic feature extraction, it can be trained within seconds. This model runs up to 200 times faster than a cycle accurate simulator and offers 97% accuracy in terms of memory latency on average. We will further introduce this model in Chapter 6 with more details.

Finally, for larger scale systems like high performance computing systems, where the performance is often dictated by data movement, we propose a new set of high bisection bandwidth, low latency interconnect topologies to improve the performance of data movement. Simulating large scale systems and our proposed topology network requires distributed simulation tools, and therefore we implement proposed topologies into a distributed parallel discrete event simulator. We then run large scale simulations up to more than 100,000 nodes for both existing and proposed topologies, and characterizing other factors that can be critical to system performance such as routing and flow control, interface technology, and physical link properties (latency, bandwidth). Detailed results and analysis can be found in Chapter 7.

In brief, the contributions of this dissertation can be summarized as follows:

- We develop a state-of-the-art cycle accurate DRAM simulator that has the best simulation performance and features among existing DRAM simulators.

It is validated by hardware model and supports thermal simulation for stacked DRAM as well.

- We conduct a thorough memory characterization over popular existing DRAM protocols using cycle accurate simulations. Through the experiments we identify the performance bottleneck of memory intensive workloads and how modern DRAM protocols help reduce the performance overhead with increased parallelism.

- We propose and build the first practical parallel DRAM simulator, coupled with a relaxed synchronization scheme called MegaTick that helps boost the parallel performance. We comprehensively evaluate the idea and show MegaTick can deliver effective performance gain with modest accuracy loss for multi-channel DRAM simulations.

- We discuss the limitations of cycle accurate DRAM simulation models, and quantitatively demonstrate how cycle accurate models are holding back overall simulation performance. We also showcase how cycle accurate models are incompatible with modern architecture simulation frameworks.

- We propose and prototype the first machine learning based DRAM simulation model. We convert the DRAM modeling problem into a multi-class classification problem for DRAM latencies, and develop a novel dynamic feature extraction method that saves training time and improves model accuracy.

- Our machine learning prototype model runs up to about 300 times faster than

cycle accurate model, predicts 97% memory request latencies accurately, and it can be easily integrated into modern architecture simulation frameworks. It opens up a completely new pathway to future DRAM modeling.

- We propose efficient interconnect topologies for large scale memory systems. We implement our proposed topologies into a discrete parallel event simulation framework, and evaluate with existing topologies through simulation. Our results show the proposed topologies outperforms traditional topologies at large network scale and workloads.

Chapter 2: Cycle Accurate Main Memory Simulation

In this chapter we introduce the main memory technology background, its modeling technique, and how we design and develop our cycle accurate main memory simulator.

2.1 Memory Technology Background

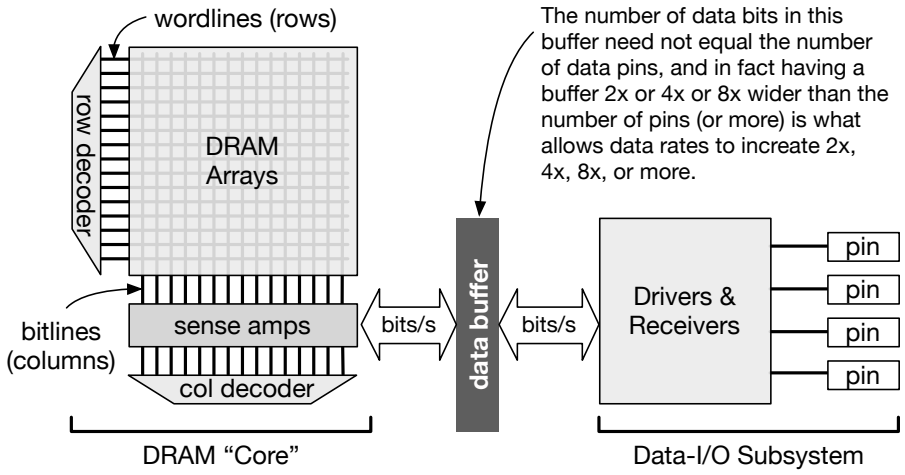


Figure 2.1: *Stylized DRAM internals, showing the importance of the data buffer between DRAM core and I/O subsystem. Increasing the size of this buffer, i.e., the fetch width to/from the core, has enabled speed increases in the I/O subsystem that do not require commensurate speedups in the core.*

Dynamic Random Access Memory (DRAM) uses a single transistor-capacitor

pair to store each bit. A simplified internal organization is shown in Figure2.1, which indicates the arrangement of rows and columns within the DRAM arrays and the internal core's connection to the external data pins through the I/O subsystem. The use of capacitors as data cells has led to a relatively complex protocol for reading and writing the data, as illustrated in Figure2.2. The main operations include *precharging* the bitlines of an array, *activating* an entire row of the array (which involves *discharging* the row's capacitors onto their bitlines and *sensing* the voltage changes on each), and then *reading/writing* the bits of a particular subset (a *column*) of that row [2].

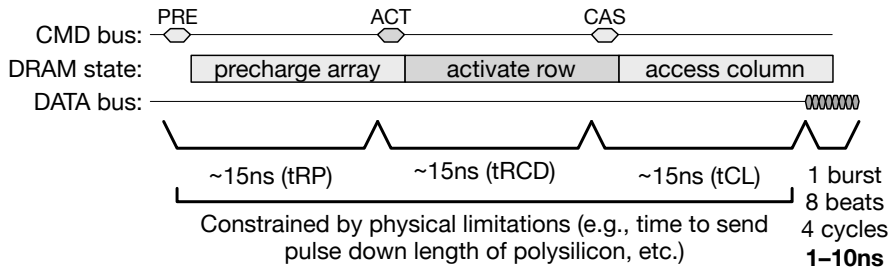


Figure 2.2: *DRAM read timing, with values typical for today. The burst delivery time is not drawn to scale: it can be a very small fraction of the overall latency. Note: though precharge is shown as the first step, in practice it is performed at the end of each request to hide its overhead as much as possible, leaving the array in a precharged state for the next request.*

Previous studies indicate that increasing DRAM bandwidth is far easier than decreasing DRAM latency [2–6], and this is because the determiners of DRAM latency(e.g., precharge, activation, and column operations) are tied to physical constants such as the resistivity of the materials involved and the capacitance of the

storage cells and bitlines. Consequently, the timing parameters for these operations are measured in nanoseconds and not clock cycles; they are independent of the DRAM's external command and data transmission speeds; and they have only decreased by relatively small factors since DRAM was developed (the values have always been in the tens of nanoseconds).

The most significant changes to the DRAM architecture have come in the data interface, where it is easier to speed things up by designing low-swing signaling systems that are separate from the DRAM's inner core [7]. The result is the modern DDR architecture prevalent in today's memory systems, in which the interface and internal core are decoupled to allow the interface to run at speeds much higher than the internal array-access circuitry. The organization first appeared in JEDEC DRAMs at the first DDR generation, which introduced a *2n prefetch* design that allowed the internal and external bandwidths to remain the same, though their effective clock speeds differed by a factor of two, by "prefetching" twice the number of bits out of the array as the number of data pins on the package. The DDR2 generation then doubled the prefetch bits to 4x; the DDR3 generation doubled it to 8x, and so on. This is illustrated in Figure2.1, which shows the decoupling data buffer that lies between the core and I/O subsystem. The left side of this buffer (the core side) runs slow and wide; the right side (the I/O side) runs fast and narrow; the two bandwidths are equal.

This decoupling has allowed the DRAM industry to focus heavily on improving interface speeds over the past two decades.As shown in Figure2.2, the time to transmit one burst of data across the bus between controller and DRAM is measured in

cycles and not nanoseconds, and, unlike the various operations on the internal core, the absolute time for transmission *has* changed significantly in recent years. For instance, asynchronous DRAMs as recent as the 1990s had bus speeds in the range of single-digit Mbps per pin; DDR SDRAM appeared in the late 1990s at speeds of 200 Mbps per pin, two orders of magnitude faster; and today's GDDR5X SGRAM speeds, at 12 Gbps per pin, are another two orders of magnitude faster than that. Note that every doubling of the bus speed reduces the burst time by a factor of two, thereby exacerbating the already asymmetric relationship between the data-access protocol (operations on the left) and the data-delivery time (the short burst on the right).

The result is that system designers have been scrambling for years to hide and amortize the data-access overhead, and the problem is never solved, as every doubling of the data-bus speed renders the access overhead effectively twice as large. This has put pressure in two places:

- **The controller design**. The controller determines how well one can separate requests out to use different resources (e.g., channels and banks) that can run independently, and also how well one can gather together multiple requests to be satisfied by a single resource (e.g., bank) during a single period of activation.

- **Parallelism in the back-end DRAM system**. The back-end DRAM system is exposed as a limitation when it fails to provide sufficient concurrency to support the controller. This concurrency comes in the form of parallel channels, each with multiple ranks and banks.

It is important for system design to balance application needs with resources available in the memory technology. As previous research has shown, not all applications can make use of the bandwidth that a memory system can provide [8], and even when an application *can* make use of significant bandwidth, allocating that resource without requisite parallelism renders the additional bandwidth useless [6]. In simpler terms, *more* does not immediately translate to *better*. This chapter studies which DRAM architectures provide more, and which architectures do it better. The following sections describe the DRAM architectures under study in terms of their support for concurrency and parallel access.

DDR DRAM protocols have evolved over the past couple of decades with each successive generation more or less doubling the maximum supported theoretical pin bandwidth of the previous generation. This has primarily been achieved either by making the DRAM pin interface wider or by increasing the frequency at which the data is transmitted across the DDR interface. Each of these methods have their own limitations in terms of feasibility and power budget. For most of the modern systems with on die memory controller and off-package DDR DIMMs, the maximum width of the DDR interface is fundamentally limited by how many I/O pins the CPU die allocates for interconnection with the DDR DIMMs. Since the number of I/O pins on the CPU die is a scarce resource, the DDR bus width has remained the same for the DDR2, DDR3, and DDR4 class of memories with the extra I/O pins allocated utilized to increase the number of DDR channels supported by the CPU. GDDR memories have traditionally had a wider DRAM-device interface to support higher bandwidths at relatively low capacities. On-package memories such

as High Bandwidth Memory (HBM) aren't constrained by the CPU I/O pins and hence tend to utilize extremely wide buses to support high bandwidth requirements. Increasing the frequency of data transmission to support higher bandwidths comes at cost an increase in power consumption and a decrease in reliability. The doubling of maximum supported pin bandwidth across DDR2, DDR3, DDR4 generation has primarily been achieved through an increase in the data transmission frequency. While a higher supported maximum theoretical pin bandwidth for a given protocol has the potential to increase the overall application performance, it only tells part of the story. This maximum theoretical pin bandwidth is only achievable if there is data always available at the I/O buffers to be transmitted across the DDR interface.

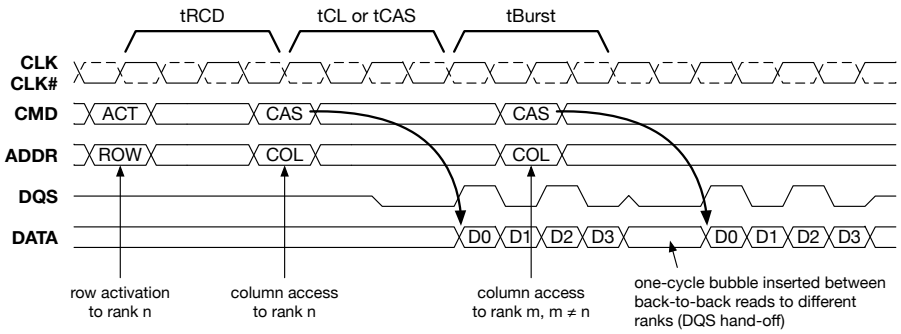


Figure 2.3: *DDR SDRAM Read timing.*

2.1.1 DDRx SDRAM

As mentioned above, the modern DDRx SDRAM protocol has become widespread and is based on the organization shown in Figure 2.1, which decouples the I/O interface speed from the core speed, requiring only that the two bandwidths on either side of the internal data buffer match. One of the distinguishing features of

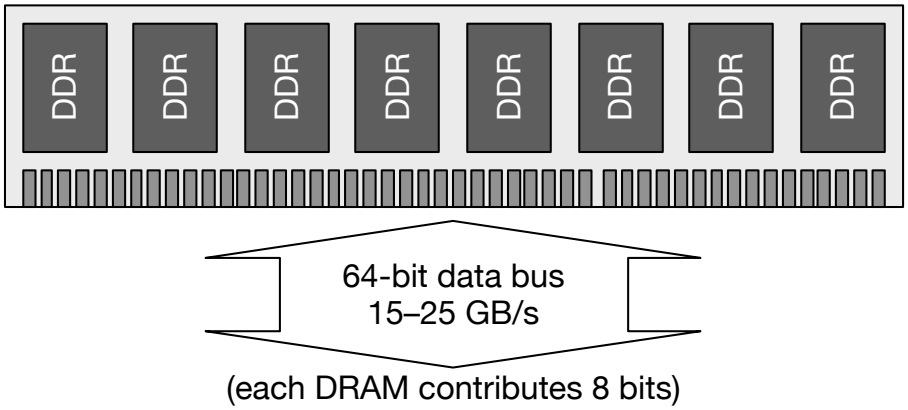


Figure 2.4: *DDRx DIMM Bus*

DDRx is its data transmission, which occurs on both edges of a data clock (double data rate, thus the name), the data clock named the *DQS* data-strobe signal. DQS is *source-synchronous*, i.e, it is generated by whomever is driving the data bus, and the signal travels in the same direction as the data. The signal is shown in Figure2.3, which presents the timing for a read operation.

In our simulations, we use a DIMM organization as shown in Figure2.4: a 64-bit data bus comprising eight x8 DRAMs.

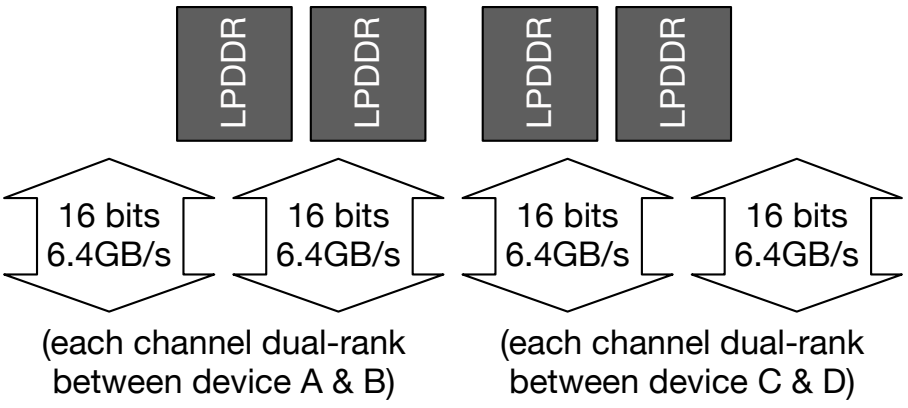


Figure 2.5: *LPDDR Bus*

2.1.2 LPDDRx SDRAM

Low Power DDR SDRAM makes numerous optimizations to achieve the same bandwidth as DDRx, in the same multi-drop organizations (e.g. multi-rank DIMMs), but at a significantly reduced power cost. Optimizations include removing the DLL, strict usage of the DQS strobe, and improved refresh. Since the beginning, DDRx SDRAM has included a delay-locked loop (DLL), to align the DQS and data output of the DRAM more closely with the external clock [2]. Providing the DLL allowed system designers to forgo using the source-synchronous data strobe DQS and instead use the system's global clock signal for capturing data, thereby making a system simpler. The downside is that the DLL is one of the most power-hungry components in a DDRx SDRAM. However, some designers were perfectly content to design more complex systems that would use the DQS strobe and eliminate the DLL, resulting in lower power dissipation. Thus the LPDDR standard was born: among other power optimizations, Low Power DDR SDRAM eliminated the DLL and required that system designers use the DQS data strobe for the capture of data, because the output of the DRAM would no longer be tightly aligned with the system clock. Another optimization for LPDDR4 is that each device is not only internally multi-banked, it is internally *multi-channel* [9]. Each device has two control/address buses, two data buses, and the specification describes the Quad-Die Quad-Channel Package: it has four dies and four separate buses, each 16 bits wide, with 16 bits coming from each of two devices in an overlapped, dual-rank configuration. This is an incredible amount of parallelism in a small, low-power package and approaches the parallelism

(if not the bandwidth) of HBM.

In our simulations, we model LPDDR4 in two different ways that are common: first, we use a DIMM like that in Figure2.4. Second, we simulate the the Quad-Die, Quad-Channel Package shown in Figure2.5.

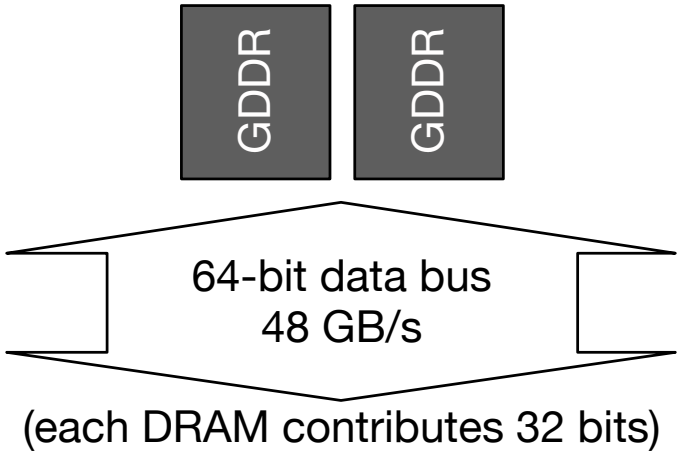


Figure 2.6: *GDDR Bus*

2.1.3 GDDR5 SGRAM

The DDRx standards have provided high bandwidth and high capacity to commodity systems (laptops, desktops), and the LPDDRx standards have offered similar bandwidths at lower power. These serve embedded systems as well as super-computers and data centers that require high performance and high capacity but have strict power budgets.

The GDDRx SGRAM standards have been designed for graphics subsystems and have focused on even higher bandwidths than DDRx and LPDDRx, sacrificing channel capacity. SGRAMs are not specified to be packaged in DIMMs like the

DDRx and LPDDRx SDRAMs. Each SGRAM is packaged as a wide-output device, typically coming in x16 or x32 datawidths, and they often require significant innovations in the interface to reach their aggressive speeds.

For example, GDDR5 runs up to 6Gbps per pin, GDDR5X is available at twice that, and the protocols require a new clock domain not seen in DDRx and LPDDRx standards: Addresses are sent at double-data-rate on the system clock, and the data strobe now runs a higher frequency than the system clock, as well as no longer being bi-directional. This has the beneficial effect of eliminating the dead bus cycle shown in Figure2.3 as the "DQS hand-off," as the strobe line need not idle if it is never turned around. Instead of being source-synchronous, the data strobe is unidirectional and used by the DRAM for capturing data. For capturing data at the controller side during data-read operations, the controller trains each GDDR5 SGRAM separately to adjust its data timing at a fine granularity relative to its internal clock signal, so that the data for each SGRAM arrives at the controller in sync with the controller's internal data clock.

In our simulations, we use an organization as shown in Figure2.6: a 64-bit data bus made from four x16 GDDR5 chips placed side-by-side.

2.1.4 High Bandwidth Memory (HBM)

JEDEC's High Bandwidth Memory uses 3D integration to package a set of DRAMs; it is similar to the DIMM package shown in Figure2.7 in that it gathers together eight separate DRAM devices into a single parallel bus. The difference is

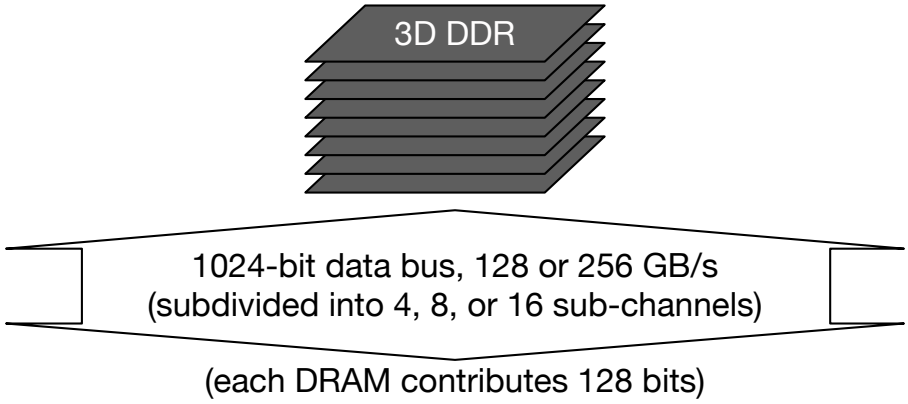


Figure 2.7: *HBM Interface*

that HBM uses through-silicon vias (TSVs) as internal communication busses, which enables *far* wider interfaces. Whereas a DDRx-based DIMM like that in Figure2.4 gangs together eight x8 parts (each part has 8 data pins), creating a bus totaling 64 bits wide, HBM gangs together eight x128 parts, creating a bus totaling 1024 bits wide. This tremendous width is enabled by running the external communications over a silicon *interposer*, which supports wire spacing far denser than PCBs. This approach uses dollars to solve a bandwidth problem, which is always a good trade-off. JEDEC calls this form of packaging "2.5D integration."

The 8 channels of HBM can operate individually or cooperatively. HBM2 standard also introduced pseudo-channel, which further divide one channel into two pseudo channels.

2.1.5 Hybrid Memory Cube (HMC)

Hybrid Memory Cube is unique in that, unlike all the other DRAM architectures studied herein, the DRAM interface is not exported; instead, HMC packages

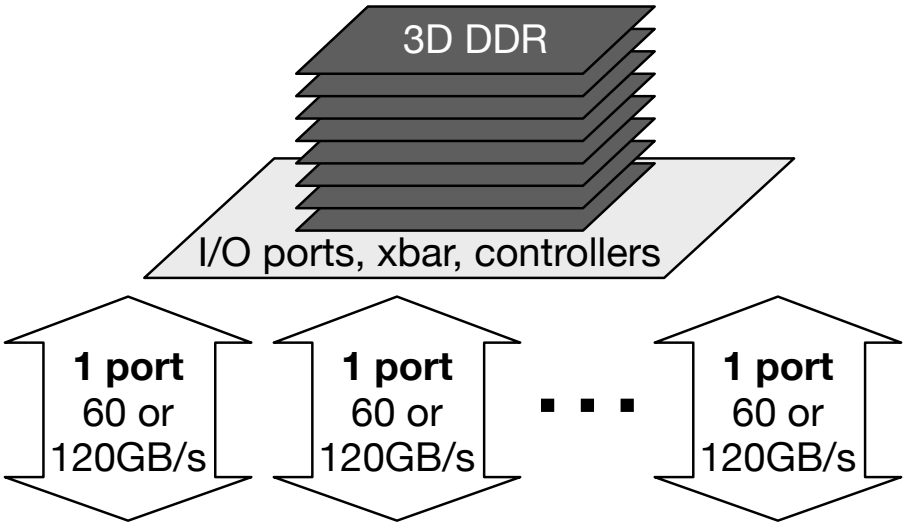


Figure 2.8: *HMC Interface*

internally its own DRAM controllers. As shown in Figure2.8, it includes a 3D-integrated stack of DRAMs just like HBM, but it also has a non-DRAM logic die at the bottom of the stack that contains three important things:

1. A set of memory controllers that control the DRAM. HMC1 has 16 internal controllers; HMC2 has up to 32.

2. The interface to the external world: a set of two or four high-speed ports that are independent of each other and transmit a generic protocol, so that the external world need not use the DRAM protocol shown in Figure2.2. Link speed and width can be chosen based on needs.

3. An interconnect that connects the I/O ports to the controllers. Communication is symmetric: requests on any link can be directed to any controller, and back.

2.1.6 DRAM Modeling

As introduced in Section 2.1, there is a set of timing constraints on how DRAM commands can be issued, and the new memory protocols only increase such constraints. The DRAM controller has the sole responsibility of bookkeeping these commands and the constraints to ensure timing correctness and no bus conflicts. On top of this, to maximize the performance and fairness, the controller also has the responsibility of scheduling the requests efficiently. Studies have shown properly designed scheduling algorithms can lead to huge performance gains [10]. Therefore, an accurate modeling of a DRAM controller should take into account both correctness and scheduling performance, and this is why we not only need to design our simulator to be generalized to simulate all DRAM protocols with correct DRAM timings, but also need to be specific to fully model the features that differentiate the protocols.

2.2 Simulator Design & Capability

In this section we introduce the design and features of our new cycle accurate DRAM simulator, DRAMsim3, and how we bridge the architecture simulation with thermal modeling.

2.2.1 Simulator Design & Features

We build the simulator in a modular way that it not only supports almost every major DRAM technologies existing today, but it also supports a variety of features

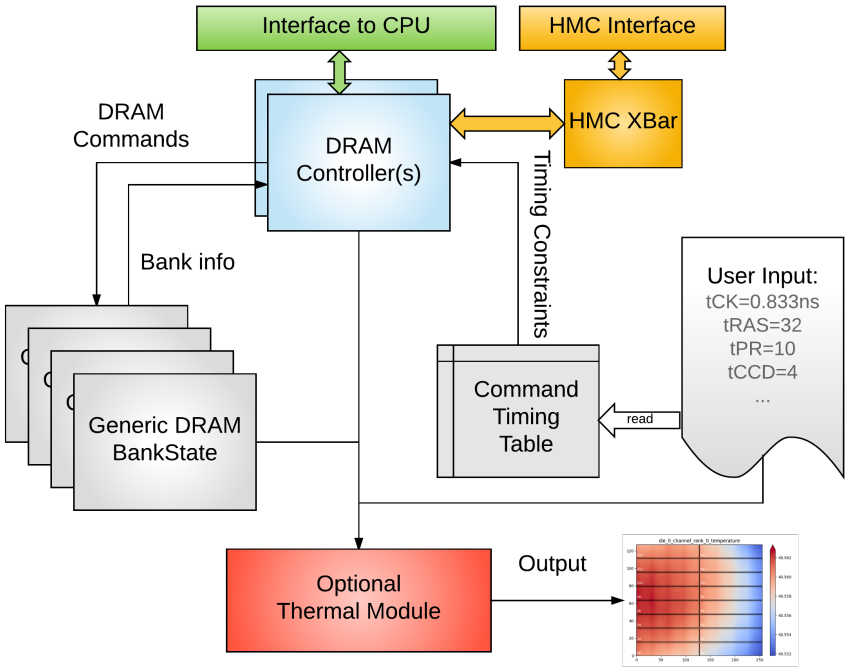


Figure 2.9: *Software Architecture of DRAMsim3*

that comes along with these technologies. The idea is first to build a generic parameterized DRAM bank model which takes DRAM timing and organization inputs, such as number of rows and columns, the values of *tCK, CL, tRCD*, etc. Then we build DRAM controllers that initialize banks and bankgroups according to which DRAM protocol it is simulating, and enable controller features only available on that DRAM protocol. For example, dual-command issue is only enabled when simulating an HBM system, while *t32AW* enforcement is only enabled when simulating a GDDR5 system. On top of the controller models, we build the system-level interfaces to interact with a CPU simulator or a trace frontend. This interface can also be extended to add additional functionality, and we add a cycle accurate crossbar and arbitration logic specifically for HMC to faithfully simulate its internals. The

system diagram is shown as Figure 2.9.

This modular hierarchical design allows us to add basic support for new protocols as simple as adding a text configuration file without compiling the code. It also enables us to customize protocol-specific features modularly without affecting other protocols. In our code repository, we ship more than 80 configuration files for various DRAM protocols.

As for the controller design, we made the following design choices:

- **Scheduling and Page Policy**: A First-Ready–First-Come-First-Served (FR-FCFS) [10] scheduling policy. FR-FCFS can reduce the latency and improve throughput by scheduling overlapped DRAM commands. We also use open-page policy with row-buffer starvation prevention improves fairness. We apply this scheme to all memory controllers except for HMC, because HMC only operates in strict close-page mode.

- **DRAM Address Mapping**: Our address mapping offers great flexibility, and users can specify the bit fields in arbitrary order. As for default address mapping schemes, to reduce row buffer conflicts and exploit parallelism among DRAM banks, we interleaved the DRAM addresses in the pattern of *row-bank-bankgroup-column* (from MSB to LSB). For configurations with multiple channels or ranks, we also interleaved the channel/rank bits in between the row-address bits and bank-address bits. Note that DDR3 has no bank group, and so this is ignored. Another exception: HMC enforces a close-page policy that does not take advantage of previously opened pages, and thus

putting column address bits on the LSB side would not be beneficial. Therefore we adopt the address-mapping scheme as default recommended by the HMC specification, which is *row-column-bank-channel* (from MSB to LSB).

- **HMC Interface**: Different from all other DRAM protocols, HMC uses high-speed links that transmit a generic protocol between the CPU and HMC's internal vault controllers. To accurately simulate this behavior, we modeled a crossbar interconnect between the high-speed links and the memory controllers. The packets are broken down to flits to be sent across the internal crossbar, which has two layers: one for requests and another for responses, to avoid deadlocks. We use FIFO arbitration policy for the crossbar control.

- **Refresh Policy**: We have a refresh generator that can raise refresh request based on per-rank refresh or per-bank refresh, depending on the user's input. The refresh policy is independent of how the controller handles the refresh, and therefore new refresh policies can be easily integrated as well.

DRAMsim3 uses Micron's DRAM power model [11] to calculate the power consumption on the fly, or it can generate a command trace that can be used as inputs for DRAMPower [12]. The power data can also be fed into an optional thermal model running side-by-side or standalone; we will further illustrate it in Section 2.2.2.

The software architecture of the simulator is shown in Figure 2.9 and the protocols and features supported are listed in Table 2.1.

DRAMsim3 can be built as a shared library that can be integrated into popular

Table 2.1: *Supported Protocols & Features of DRAMsim3*

Protocols	Features
DDR3 DDR4 DDR5 GDDR5 GDDR5X LPDDR3 LPDDR4 STT-MRAM HBM HMC	Bankgroup timings Self-refresh timings (GDDR5) t32AW (GDDR5X) QDR mode Bank-level refresh (HBM) Dual-command issue (HMC) High-speed link simulated (HMC) Internal X-bar simulated Fine-grained flexible address mapping

CPU simulators or simulation frameworks such as SST [13], ZSim [14] and Gem5 [15] as their backend memory simulator. We will open-source the code repository, along with the glue code to work with above stated simulators.

2.2.2 Bridging Architecture and Thermal Modeling

Fine-grained thermal simulation can be time consuming due to the amount of calculations need to be done. Therefore, we offer the freedom to adjust the granularity in both spatial and temporal domains so that the user can chose accordingly and balance the simulation speed versus accuracy. For spatial granularity, each DRAM

die is divided into smaller grids that in reality would correspond to DRAM subarrays (shown as Figure 2.11 (c)). By default it's 512×512 cells but users can also use larger grids to speed up simulation with less accuracy. For temporal granularity, the transient thermal calculation is done once per epoch, and the epoch length can be configured as an arbitrary number of DRAM cycles.

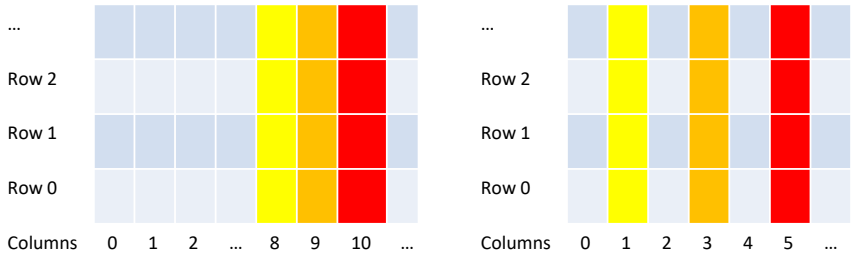
During each thermal epoch, the thermal module needs to know A) how much energy is consumed on that die, and B) what is the energy distribution (in physical location). We use Micron's DDR power model to calculate power, and given the time in cycles, we can calculate energy. The energy can be broken down into per-command energy (e.g. activation, precharge, read and write) and background energy. We assign those per-command energy values only to those locations that the command concerns, for instance, we only distribute the activation energy to wherever the activated row is on the die. Then we distribute the background energy across the whole die evenly.

To know exactly the location to map the per-command energy, the physical layout of the DRAM circuit needs be known. Unfortunately, most of the DRAM circuit designs and layouts are proprietary information that is not publicly available. According to the reverse-engineered results shown in a recent research [16], DRAM manufacturers obfuscate DRAM cell locations by remapping the address bits. I.e. the DRAM address sent by the controller is remapped internally in the DRAM circuitry, and as a result, the row and column in the controller's view may end up in a different physical row and column on the DRAM die. For example, if, like [16] discovered, the column address sent by controller is internally decoded as:

$$C_{10}...C_3C_2C_1C_0 \rightarrow C_{10}...C_4C_2C_1C_0C_3$$

where C_i is the ith bit of column address, the controller's view of columns 8, 9 and 10 would actually be physical columns 1, 3, and 5. Note that this rearranging is transparent to DRAM controller and works independently from the address mapping that controller has to perform.

To accurately model this, we implement a location mapping function which allows users to input any arbitrary address bits location remapping schemes. e.g. If an DRAM part has 4 bank address bits, 15 row address bits, and 10 column address bits, the total number of allowed location mapping schemes is $(4+15+10)! \approx 8.84^{30}$. Therefore, while we provide a default mapping scheme, the users can always change the mapping scheme to meet a specific circuit design.



(a) Column 8-10 in DRAM Controller's view (b) Actual physical columns after mapping

Figure 2.10: *An example of how DRAM internal structures can be rearranged. (a) shows column 8–10 in DRAM controller's view (b) show the corresponding physical columns internally in DRAM subarrays.*

2.2.3 Thermal Models

Given that our functional model can simulate a variety of DRAM protocols including both stacked and planar designs, the thermal models differentiate for each case in order to achieve more accuracy. For 3D DRAMs (HMCs, HBMs) as illustrated in Figure 2.11(a), the temperature of each stacked die is estimated. For 2D DRAMs, however, since a memory module comprises several DRAM devices which are separated from each other in distance (Figure 2.11 (b)), we assume all the DRAM devices in a rank have the same thermal condition, and they are independent when calculating the temperature. Therefore, DRAMsim3 only estimates the temperature for a single DRAM device per rank even though it simulates the function for the whole memory module. We assume each DRAM die (or DRAM device) comprises three layers: active layer, metal layer and dielectric layer. The power is generated from the active layer and is dissipated to the ambient through a silicon substrate (as illustrated in Figure 2.11(c)). We assume other surfaces of the device are adiabatic. In the following, we will introduce the thermal modeling method in detail.

2.2.3.1 Transient Model

We follow the **energy balance method** [17] to model the temperature. In this technique, the dies are divided into small volume elements (called thermal grids) as illustrated in Figure 2.11(c). Then each thermal grid is modeled as a nodal point and the heat conduction in the DRAM circuit is modeled as shown in Figure 2.12.

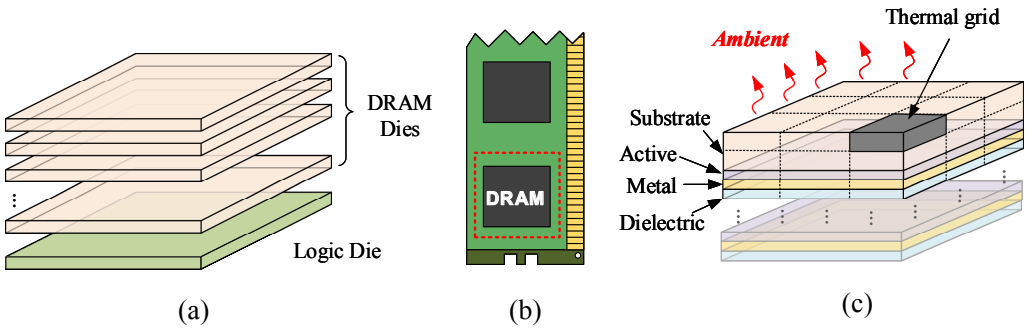


Figure 2.11: *Illustration of (a) the 3D DRAM, (b) memory module with 2D DRAM devices and (c) layers constituting one DRAM die*

Each pair of adjacent nodal points is connected with a *thermal resistor* (R_{vert}, R_{lat}) which indicates a heat conduction path between the two nodes. The thermal resistance is calculated according to the material's thermal conductivity (k) and the geometrical dimension of the related thermal grids. As shown in Figure 2.12, $R_{lat}^{1,2} = \frac{\Delta X/2}{k_1 \Delta Y \Delta Z} + \frac{\Delta X/2}{k_2 \Delta Y \Delta Z}$. R_{vert} is calculated similarly. For the node that connects to the ambient, the corresponding resistance is calculated as $R_{amb} = \frac{\Delta Z/2}{k_3 \Delta X \Delta Z}$. Besides the thermal resistor, each nodal point is connected with a *thermal capacitor* (C) which represents the ability of the thermal grid to store the thermal energy. Given the specific heat capacity (C_h) of the material of a thermal grid, the related capacitance is calculated as $C = \rho C_h \times \Delta X \Delta Y \Delta Z$ (where ρ is the density of the material within the thermal grid). For each thermal grid on the active layer, there is a *heat source* (q_s) connected to the nodal point. q_s represents the heat generation rate within the thermal grid and is calculated based on the power dissipated in that grid. Given the above information, we can estimate the temperature of a node, which represents the average temperature within the corresponding thermal grid.

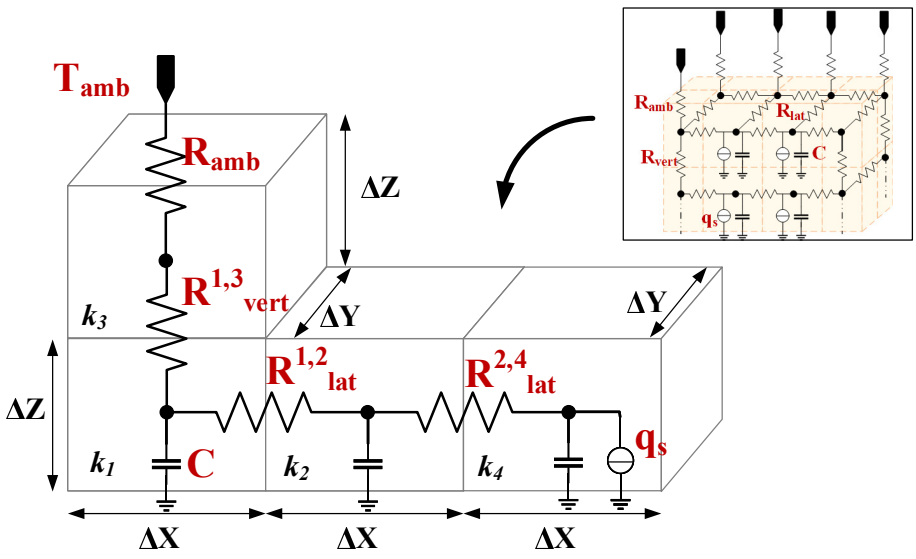


Figure 2.12: *Illustration of the thermal model*

Suppose there are totally N thermal grids. Let $\boldsymbol{P} \in \mathbb{R}^N$ and $\boldsymbol{T} \in \mathbb{R}^N$ represent the power and temperature for all grids, respectively; $\boldsymbol{G} \in \mathbb{R}^{N \times N}$ represents the matrix of thermal conductance which is calculated using the thermal resistance; $\boldsymbol{C} \in \mathbb{R}^{N \times N}$ is a diagonal matrix with each element in the diagonal representing the thermal capacitance of the grid. Then the temperature at time t can be calculated by solving the following equation:

$$\boldsymbol{GT} + \boldsymbol{P} = \boldsymbol{C}\frac{d\boldsymbol{T}}{dt}. \tag{2.1}$$

In practice, the transient temperature profile is calculated every power sampling epoch which is defined by the user. At the end of each epoch, we estimate the average power profile (*i.e.* $\boldsymbol{P}$) during this epoch. This $\boldsymbol{P}$, together with the temperature at the end of previous epoch ($\boldsymbol{T}_{t-1}$), is used to calculate the current temperature ($\boldsymbol{T}_t$). In DRAMsim3, we use *explicit method* [17] to get the solution. This method subdivides the epoch into small time steps (Δt) and calculates the

temperature for each Δt iteratively. $\boldsymbol{T}_t$ is calculated at the last time step. In order to guarantee the convergence, this method requires the time step to be small enough:

$$\Delta t \leq \frac{C_{i,i}}{G_{i,i}} \qquad \forall i = 0, 1, 2, ..., N-1 \tag{2.2}$$

In our simulator, users can specify the thermal parameters (including the thermal conductivity, thermal capacitance *etc.*), the dimension of each layer in the DRAM, the size of a thermal grid and the length of a power sampling epoch. Given the above information, $\boldsymbol{G}$ and $\boldsymbol{C}$ will be fixed. Therefore, we only need to calculate $\boldsymbol{G}$, $\boldsymbol{C}$ and Δt (*i.e.* the proper time step) for one time at the beginning of the simulation.

2.2.3.2 Steady State Model

At the end of simulation, DRAMsim3 also estimates the steady-state temperature profile using the average power profile during the period of simulation. The steady-state thermal model only contains the resistors; hence Equation 2.1 is reduced to:

$$\boldsymbol{GT} + \boldsymbol{P} = 0. \tag{2.3}$$

Note that Equation 2.3 is a linear equation set, and $\boldsymbol{G}$ is a sparse matrix [17]. This equation is solved using SuperLU [18], which provides a library to solve large sparse linear equations.

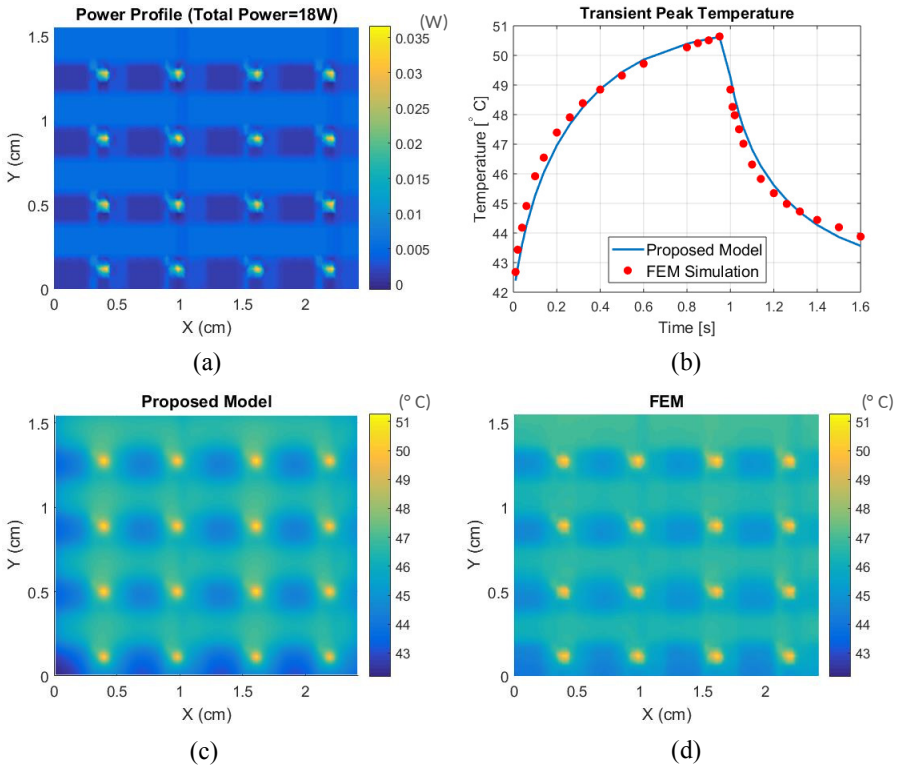


Figure 2.13: *(a) The original power profile, (b) the transient result for for the peak temperature, (c) the temperature profile at* 1*s calculated using our thermal model and (d) the temperature profile at* 1*s calculated using the FEM method*

2.2.3.3 Thermal Model Validation

The proposed thermal model is validated against the Finite Element Method (FEM) results. We use ANSYS to perform the FEM simulation. We use the thermal model to estimate the temperature for a multi-core processor die. The power profile of the multi-core processor is generated based on [19] and is illustrated in Figure 2.13(a) (Total power equals to 18W). Note that, although the processor power is used to validate the thermal model, this model is applicable to the DRAM power. This processor die contains three layers as illustrated in Figure 2.11(c). The sim-

ulation is taken for 1.6s. Before 1s, the processor power stays constant as shown in Figure 2.13(a). After 1s, the processor power is reduced by 75%. Figure 2.13(b) shows the transient peak temperature using our model and the FEM simulation. Figure 2.13(c) and (d) represent the temperature profile at 1s acquired using our model and the FEM method, respectively. According to the figure, the result of our model accurately matches the FEM result.

2.3 Evaluation

2.3.1 Simulator Validation

Other than the thermal model validation described in Section 2.2.3.3, we also validated our DRAM timings against Micron Verilog models. We take a similar approach as [20], that is, feeding request traces into DRAMsim3, we output DRAM command traces and convert them into the format that fits into Micron's Verilog workbench. We ran the Verilog workbench through ModelSim Verilog Simulator and no DRAM timing errors were produced. We not only validated the DDR3 model as previous works did, but we also validated the DDR4 model as well. DRAMsim3 is the first DRAM simulator to be validated by both models, to our knowledge.

We also use DRAMsim3 to conduct a thorough memory characterization study of various memory protocols; the results can be found later in Chapter 3.

2.3.2 Comparison with Existing DRAM Simulators

We compare DRAMsim3 with existing DRAM simulators including DRAMSim2 [20], Ramulator [21], USIMM [22] and DrSim [23]. These are open sourced DRAM simulators that can run as standalone packages with trace inputs, making it viable for us to conduct a fair and reproducible comparison.

Each simulator is compiled by clang-6.0 with O3 optimization on their latest publicly released source code (except for USIMM where we use officially distributed binary). We use randomly generated memory request traces for all these simulators. The requests are exactly the same for each simulator, while only the trace format is adjusted to work with each specific simulator. The read to write request ratio is 2:1. Since DDR3 is the only protocol all tested simulators support, we run each simulator with a single channel, dual rank DDR3-1600 configuration and each has the exact same DRAM structures and timing parameters. We also made sure each simulator has comparable system parameters such as queue depth.

We measure the host simulation time for each simulator to finish processing 10 million requests from the trace, to quantify simulation performance. The results are shown in Figure 2.14. In terms of simulation speed, DRAMsim3 offers the best simulation performance among the contestants: it is on average 20% faster than DRAMSim2, the next fast DRAM simulator, and more than twice as fast as the other simulators in both random and stream request patterns.

We also examine how many simulated cycles it takes for each simulator to finish 10 million random and stream requests. This is an indicator of the throughput

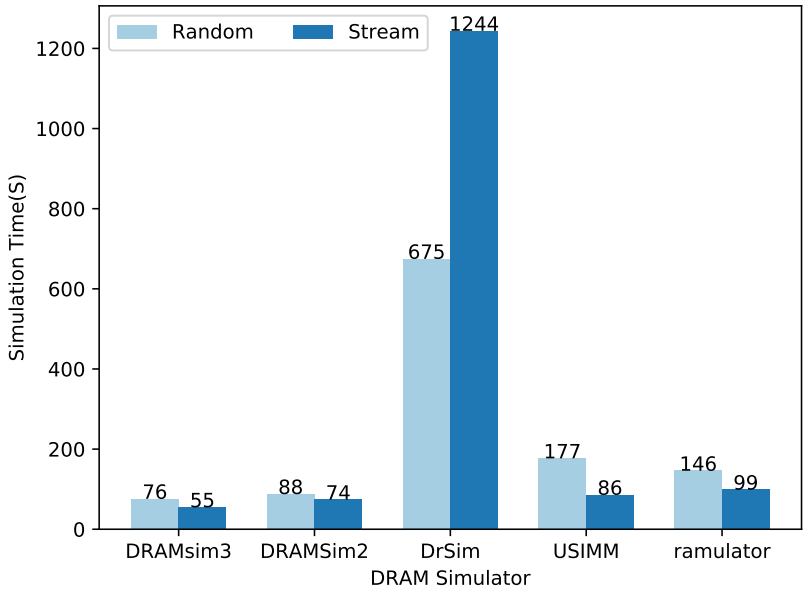


Figure 2.14: *Simulation time comparison for 10 million random & stream requests of different DRAM simulators*

or bandwidth provided by the memory controllers simulated by each simulator. For instance, the fewer simulated cycles it takes for a simulator to simulate 10 million requests, the higher the bandwidth its simulated controller will provide. The results are shown in Figure 2.15. DRAMsim3 is on par with other simulators in this measurement, indicating that the scheduler and controller design in DRAMsim3 is as efficient as the controller design in other simulators.

2.4 Conclusion

In this chapter we present DRAMsim3, a fast, validated, thermal-capable DRAM simulator. We introduced the architectural and thermal modeling capabilities of DRAMsim3. Through the evaluations we demonstrated the validation of

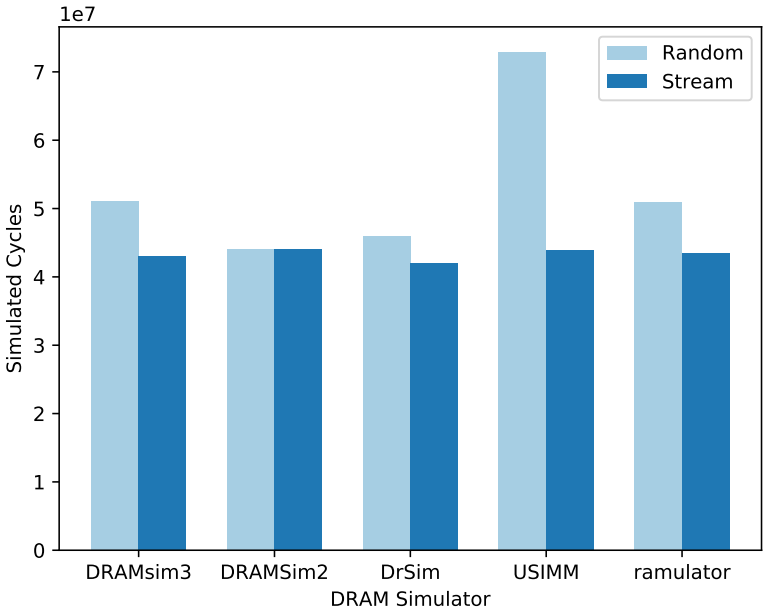


Figure 2.15: *Simulated cycles comparison for 10 million random & stream requests of different DRAM simulators*

the simulator, and showcased the unparalleled simulation performance of DRAMsim3 with uncompromising simulator design.

Chapter 3: Performance and Power Comparison of Modern DRAM Architectures

3.1 Introduction

In response to the still-growing gap between memory access time and the rate at which processors can generate memory requests [24, 25], and especially in response to the growing number of on-chip cores (which only exacerbates the problem), manufacturers have created several new DRAM architectures that give today's system designers a wide range of memory-system options from low power, to high bandwidth, to high capacity. Many are multi-channel internally. In this chapter, we present a simulation-based characterization of the most common DRAMs in use today, evaluating each in terms of its effect on total execution time and power dissipation.

We use DRAMsim3 to simulate nine modern DRAM architectures: DDR3 [26], DDR4 [27], LPDDR3 [28], and LPDDR4 SDRAM [9]; GDDR5 SGRAM [29]; High Bandwidth Memory (both HBM1 [30] and HBM2 [31]); and Hybrid Memory Cube (both HMC1 [32] and HMC2 [33]). The DRAM command timings are validated, and the tool provides power and energy estimates for each architecture. To obtain ac-

curate memory-request timing for a contemporary multicore out-of-order processor, we integrate our code into *gem5* and use its DerivO3 CPU model [15]. To highlight the differences inherent to the various DRAM protocols, we study single-channel (and single-package, for those that are multi-channeled within package) DRAM systems. Doing so exposes the fundamental behaviors of the different DRAM protocols & architectures that might otherwise be obscured in, for example, extremely large, parallel systems like Buffer-on-Board [34] or Fully Buffered DIMM [35] systems.

This chapter asks and answers the following questions:

- Previous DRAM studies have shown that the memory overhead can be well over 50% of total execution time (e.g., [5, 6, 36]); what is the overhead today, and how well do the recent DRAM architectures combat it? In particular, how well do they address the memory-latency and memory-bandwidth problems?

  As our results show, main memory overheads today, for single-rank organizations, are still 42–75% for nearly all applications, even given the relatively modest 4-core system that we study. However, when sufficient parallelism is added to the memory system to support the bandwidth, which can be as simple as using a dual-rank organization, this overhead drops significantly. In particular, the latest high-bandwidth 3D stacked architectures (HBM and HMC) do well for nearly all applications: these architectures reduce the memory-stall time significantly over single-rank DDRx and LPDDR4 architectures, reducing 42–75% overhead down to less than 30% of total execution time. These architectures combine into a single package all forms of parallelism in

the memory system: multiple channels, each with multiple ranks/banks. The most important effect of these and other highly parallel architectures is to turn many memory-bound applications to compute-bound applications, and the total execution time for some applications can be cut by factors of 2–3x.

- Where is time and power spent in the DRAM system?

  For all architectures but HBM and HMC, the majority of time is spent waiting in the controller's queues; this is true even though the workloads represent only a small number of cores. Larger systems with dozens or hundreds of cores would tend to exacerbate this problem, and this very phenomenon is seen, for example, in measured results of physical KNL systems [37]. For HBM and HMC systems, the time is more evenly distributed over queuing delays and internal DRAM operations such as row activation and column access.

  Power breakdowns are universal across the DRAM architectures studied: for each, the majority of the power is spent in the I/O interface, driving bits over the bus. This is an extremely good thing, because everything else is overhead, in terms of power; this result means that one pays for the bandwidth one needs, and the DRAM operations come along essentially for free. The most recent DRAMs, HMC especially, have been optimized internally to the point where the DRAM-specific operations are quite low, and in HMC represent only a minor fraction of the total. In terms of power, DRAM, at least at these capacities, has become a pay-for-bandwidth technology.

- How much locality is there in the address stream that reaches the primary

memory system?

The stream of addresses that miss the L2 cache contains a significant amount of locality, as measured by the hit rates in the DRAM row buffers. The hit rates for the applications studied range 0–90% and average 39%, for a last-level cache with 2MB per core. (This does not include hits to the row buffers when making multiple DRAM requests to read one cache line.) This relatively high hit rate is why optimized close-page scheduling policies, in which a page is kept open if matching requests are already in the controller's request queue (e.g., [10, 38]), are so effective.

In addition, we make several observations. First, "memory latency" and "DRAM latency" are two completely different things. *Memory latency* corresponds to the delay software experiences from issuing a load instruction to getting the result back. *DRAM latency* is often a small fraction of that: average memory latencies for DDRx and LPDDRx systems are in the 80–100ns range, whereas typical DRAM latencies are in the 15–30ns range. The difference is in arbitration delays, resource management, and whether sufficient parallelism exists in the memory system to support the memory traffic of the desired workload. Insufficient parallelism leads to long queuing delays, with requests sitting in the controller's request buffers for tens to hundreds of cycles. If your memory latency is bad, it is likely not due to DRAM latency.

This is not a new concept. As has been shown before [6], more bandwidth is not always better, especially when it is allocated without enough concurrency in

the memory system to maintain it. Execution time is reduced 21% when moving from single-rank DDR3 channels to dual-rank channels. Execution time is reduced 22% when moving from a single-channel LPDDR4 organization to a quad-channel organization. Execution time is reduced 25% for some apps when moving from a 4-channel organization of HBM to an 8-channel organization. And when one looks at the reason for the reduction, it is due to reduced time spent in queues waiting for memory resources to become free. Though it may sound counter-intuitive, average *latencies* decrease when one allocates enough *parallelism* in the memory system to handle the incoming request stream. Otherwise, requests back up, and queuing delays determine the average latency, as we see in DDRx, LPDDR4, and GDDR5 based systems. Consequently, if one's software is slow due to latency issues, consider improving your NoC, or increasing the number of controllers or channels to solve the problem.

Second, bandwidth is a critical and expensive resource, so its allocation is important. As mentioned above, having enough bandwidth with parallelism to support it can reduce execution time by 2–3x and turn some previously memory-bound apps into compute-bound apps. This is a welcome result: one can bring value to advanced processor-architecture design by simply spending *money* on the memory system. Critical rule of thumb to note: multicore/manycore architectures require at a minimum ∼1GB/s of sustained memory bandwidth per core, otherwise the extra cores sit idle [1]. Given the result mentioned above that bandwidth is the key factor in total power dissipation, this makes bandwidth allocation a non-trivial exercise. Note that our results indicate that even if an application's bandwidth

usage is not extreme, providing it with sufficient bandwidth *and* parallelism can cut execution time in half or more.

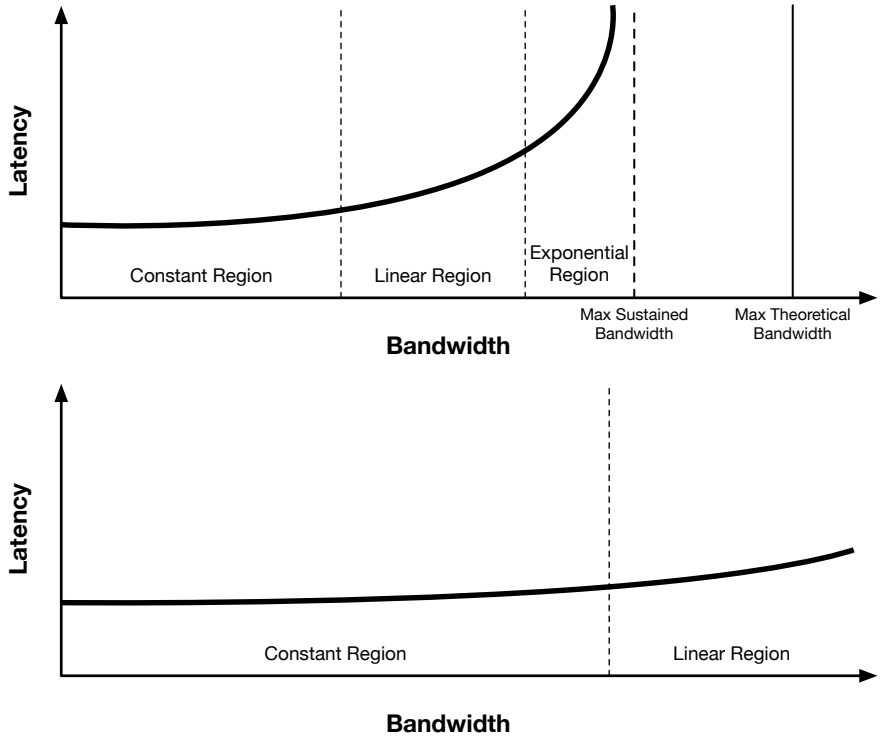


Figure 3.1: ***Top**: as observed by Srinivasan [1], when plotting system behavior as latency per request vs. actual bandwidth usage (or requests per unit time).*

Third, for real-time systems, Hybrid Memory Cube is quite interesting, as it provides *highly* deterministic latencies. This characteristic of HMC has been noted before [25] and is due in part to the architecture's extremely high bandwidth, which pushes the exponential latency region out as far as possible (see Figure 3.1 for an illustration). However, high bandwidth alone does not provide such determinism, otherwise we would see similar deterministic latencies in HBM systems, which we do not. The effect is due not only to bandwidth but also to the internal scheduling algorithms of HMC, which use a close-page policy that does not opportunistically

seek to keep a page open longer than required for the immediate request. While this may sacrifice some amount of performance, it provides predictable latencies and keeps the internal DRAM power down to a level below that of all other DRAM architectures studied, including power-optimized LPDDR4.

The following sections provide more background on the topic, describe our experimental setup, and compare & contrast the various DRAM architectures.

3.2 Experiment Setup

We run DRAMsim3 within the gem5 simulator. The following sections elaborate.

Table 3.1: *Gem5 Simulation Setup*

CPU	Gem5 DerivO3 CPU model, x86 architecture, 4-core
Core	4GHz, Out-of-order, 8-fetch, 8-issue, 192 reorder buffer entries
L1 I-Cache	per-core, 32KB, 2-way associative, 64 Byte cache line, LRU
L1 D-Cache	per-core, 64KB, 2-way associative, 64 Byte cache line, LRU
L2 Cache	shared, MOESI protocol, 8MB, 8-way associative, 64 Byte cache line, LRU
Workloads	bzip2, gcc, GemsFDTD, lbm, mcf, milc, soplex, STREAM, GUPS, HPCG

3.2.1 Simulation Setup

We configure gem5 to simulate an average desktop processor: x86-based, 4-core, out-of-order. The detailed configuration is in Table 3.1. From several

Table 3.2: *DRAM Parameters*

DRAM Type	Density	Device Width	Page Size	# of Banks (per rank)	Pin Speed	Max. Bandwidth [3]	tRCD (ns)	tRAS (ns)	tRP (ns)	CL/CWL (ns)
DDR3	8Gb	8 bits	2KB	8	1.866Gbps	14.9GB/s	14	34	14	14/10
DDR4	8Gb	8 bits	1KB	16	3.2Gbps	25.6GB/s	14	33	14	14/10
LPDDR4	6Gb	16 bits	2KB	8	3.2Gbps	25.6GB/s	-[5]	-[5]	-[5]	-[5]
GDDR5	8Gb	16 bits	2KB	16	6Gbps	48GB/s	14/12[4]	28	12	16/5
HBM[1]	4Gbx8	128 bits	2KB	16	1Gbps	128GB/s	14	34	14	14/4
HBM2[1]	4Gbx8	128 bits	2KB	16	2Gbps	256GB/s	14	34	14	14/4
HMC[1]	2Gbx16	32 bits	256 Bytes	16	2.5Gbps[2]	120GB/s	14	27	14	14/14
HMC2[1]	2Gbx32	32 bits	256 Bytes	16	2.5Gbps[2]	320GB/s	14	27	14	14/14

[1] HBM and HMC have multiple channels per package, therefore the format here is channel density x channels.

[2] The speed here is HMC DRAM speed, simulated as 2.5Gbps according to [8]. HMC link speed can be 10–30Gbps.

[3] Bandwidths for DDR3/4, LPDDR4 and GDDR5 are based on 64-bit bus design; HBM and HBM2 are 8×128 bits wide; Bandwidth of HMC and HMC2 are maximum link bandwidth of all 4 links. We use 2 links 120GB/s in most simulations.

[4] GDDR5 has different values of tRCD for read and write commands.

[5] We are using numbers from a proprietary datasheet, and they are not publishable.

suites, we select benchmarks to exercise the memory system, including those from SPEC2006 [39] that are memory-bound according to [40]. These benchmarks have CPIs ranging from 2 to 14, representing moderate to relatively intensive memory workloads. We also simulate STREAM and GUPS from the HPCC benchmark suite [41]. STREAM tests the sustained bandwidth, while GUPS exercises the memory's ability to handle random requests. Finally we use HPCG [42], *high-performance conjugate gradients*, which represents memory-intensive scientific computing workloads. We ran four copies of each workload, one on each core of the simulated processor. Gem5 is configured to run in system-emulation mode, and all the benchmarks are fast-forwarded over the initialization phase of the program, and then simulated with the DerivO3 Gem5 CPU model for 2 billion instructions (500 million per core).

DRAM Parameters: Several of the most important parameters are listed in Table 3.2, including tRCD, tRAS, tRP, and tCL/CWL. Most of the parameters are based on existing product datasheets or official specifications. Some parameters, however, are not publicly available—for example, some timing parameters of HBM and HMC are not specified in publicly available documentation. Previous studies [8, 43] have established reasonable estimations of such parameters, and so we adopt the values given in these studies.

3.3 Results

The following sections present our results and analysis.

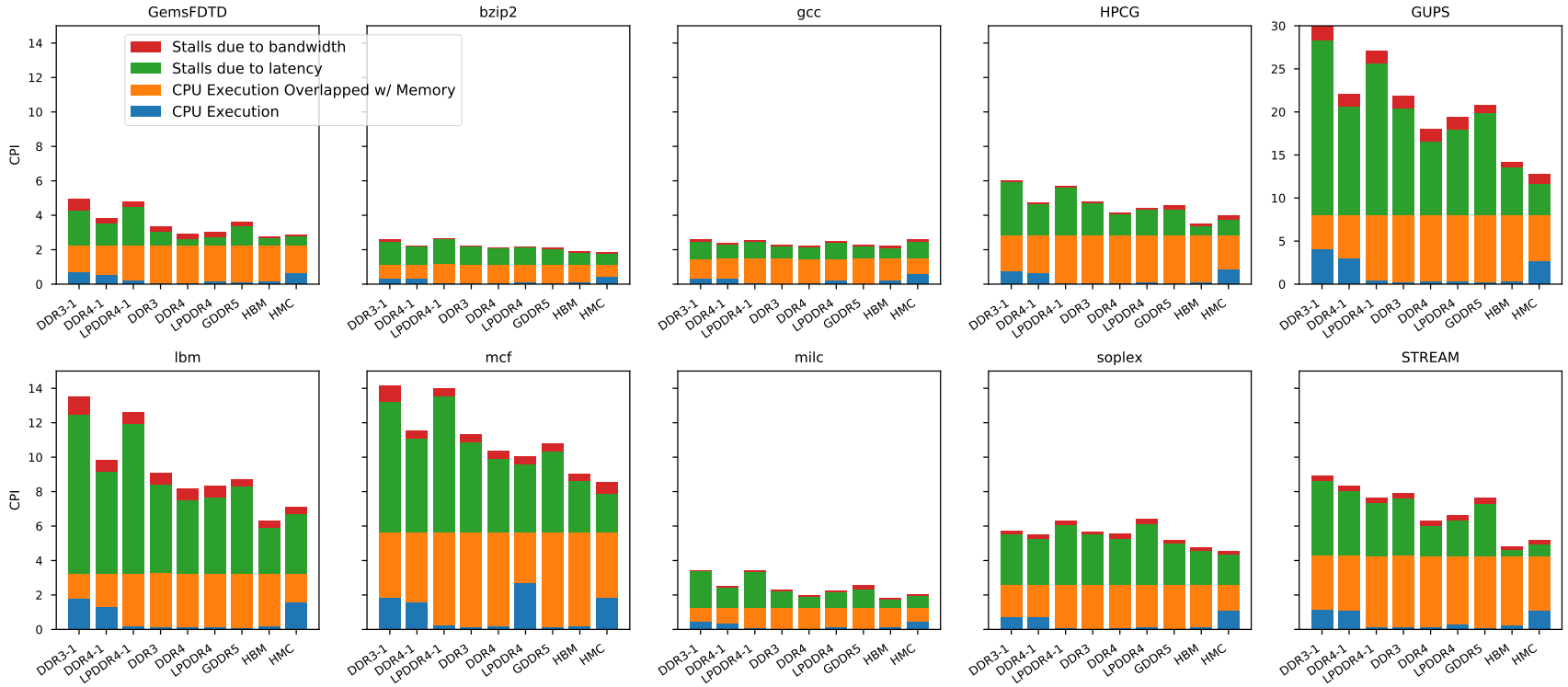


Figure 3.2: *CPI breakdown for each benchmark. Note that we use a different y-axis scale for GUPS. Each stack from top to bottom are stalls due to bandwidth, stalls due to latency, CPU execution overlapped with memory, and CPU execution.*

3.3.1 Overall Performance Comparisons

Figure 3.2 shows performance results for the DRAM architectures across the applications studied, as average CPI. To understand the causes for the differences, e.g. whether from improved latency or improved bandwidth, we follow [5] and [3]: we run multiple simulations to distinguish between true execution time and memory overhead and distinguish between memory stalls that can be eliminated by simply increasing bandwidth and those that cannot.

The tops of the orange bars indicate the ideal CPI obtained with a perfect primary memory (zero latency, infinite bandwidth). The remaining portion above the orange bars is the overhead brought by primary memory, further broken down into stalls due to lack of bandwidth (red bar) and stalls due to latency (green bar). For example, the best CPI that could be obtained from a perfect memory for STREAM, as shown in Figure 3.2, is 4.3. With DDR3, the DRAM memory contributes another 4.6 cycles to the execution time, making the total CPI 8.9. Among these 4.6 cycles added by DDR3, only 0.3 cycles are stalls due to lack of memory bandwidth; the remaining 4.3 cycles are due to memory latency.

The first thing to note is that the CPI values are all quite high. Cores that should be able to retire 8 instructions per cycle are seeing on average one instruction retire every two cycles (bzip2), to 30 cycles (GUPS). The graphs are clear: more than half of the total overhead is memory.

As a group, the highest CPI values are single-rank (DDR3-1, DDR4-1) or single channel (LPDDR4-1) configurations . Single rank configurations expose the

tFAW protocol limitations [2,44], because all requests must be satisfied by the same set of devices. Having only one rank to schedule into, the controller cannot move to another rank when the active one reaches the maximum activation window; thus the controller must idle the requisite time before continuing to the next request when this happens. The effect is seen when comparing DDR3-1 to DDR3, an average 21% improvement from simply using a dual-rank organization; or when comparing DDR4-1 to DDR4, an average 14% improvement from simply moving to dual-rank from single-rank.

LPDDR4-1 and LPDDR4 have the same bandwidths, but different configurations (64×1 vs 16×4 buses). There is a 22% improvement when using the quad-channel configuration, indicating that using more parallelism to hide longer data burst time works well in this case.

From there, the comparison is DDR3 to LPDDR4 to DDR4, and the improvement goes in that direction: LPDDR4 improves on DDR3 performance by an average of 8%, and DDR4 improves on LPDDR4 by an average of 6%. This comes from an increased number of internal banks (DDR4 has 16 per rank; LPDDR4 has 8 per rank but more channel/ranks, DDR3 has only 8 per rank), as well as increased bandwidth. The reason why LPDDR4, having more banks, does not outperform DDR4 is its slower DRAM timings, which were optimized for power but not performance.

Next in the graphs is GDDR5, which has almost twice the bandwidth of DDR4 and LPDDR4, but because it is a single-rank design (GDDR5 does not allow multi-drop bus configurations), it behaves like the other single-rank configurations: DDR3-1, DDR4-1, and LPDDR4-1, which is to say that it does not live up to its potential

under our testing setup. graphics

The best-performing DRAM architectures are HBM and HMC: the workloads are split roughly evenly on which DRAM is "best." One may not be impressed by the performance improvement here: though HBM and HMC have maximum bandwidths of 128GB/s and 120GB/s, roughly 8 and 13 times more than single-rank DDR3, they only achieve improvements of 2–3x over DDR3. Indeed, the performance improvement is less than the bandwidth increase; however, the total memory overhead decreases by a more significant amount, from being much more than half of the total CPI to accounting for less than 30% of the total CPI in many cases.

The net result is that the most advanced DRAMs, HBM and HMC, which combine all of the techniques previously shown to be important (multiple channels, multiple ranks and/or banks per channel, and extremely high bandwidths), outperform all other DRAM architectures, often by a factor of two. The difference comes from virtually eliminating DRAM overhead, and the result is that half of the benchmarks go from being memory-bound to being compute-bound.

Lastly, it is clear from Figure 3.2 that the performance improvement brought by HBM and HMC is due to the significant reduction of latency stalls. In the following section, we break down the latency component to understand better.

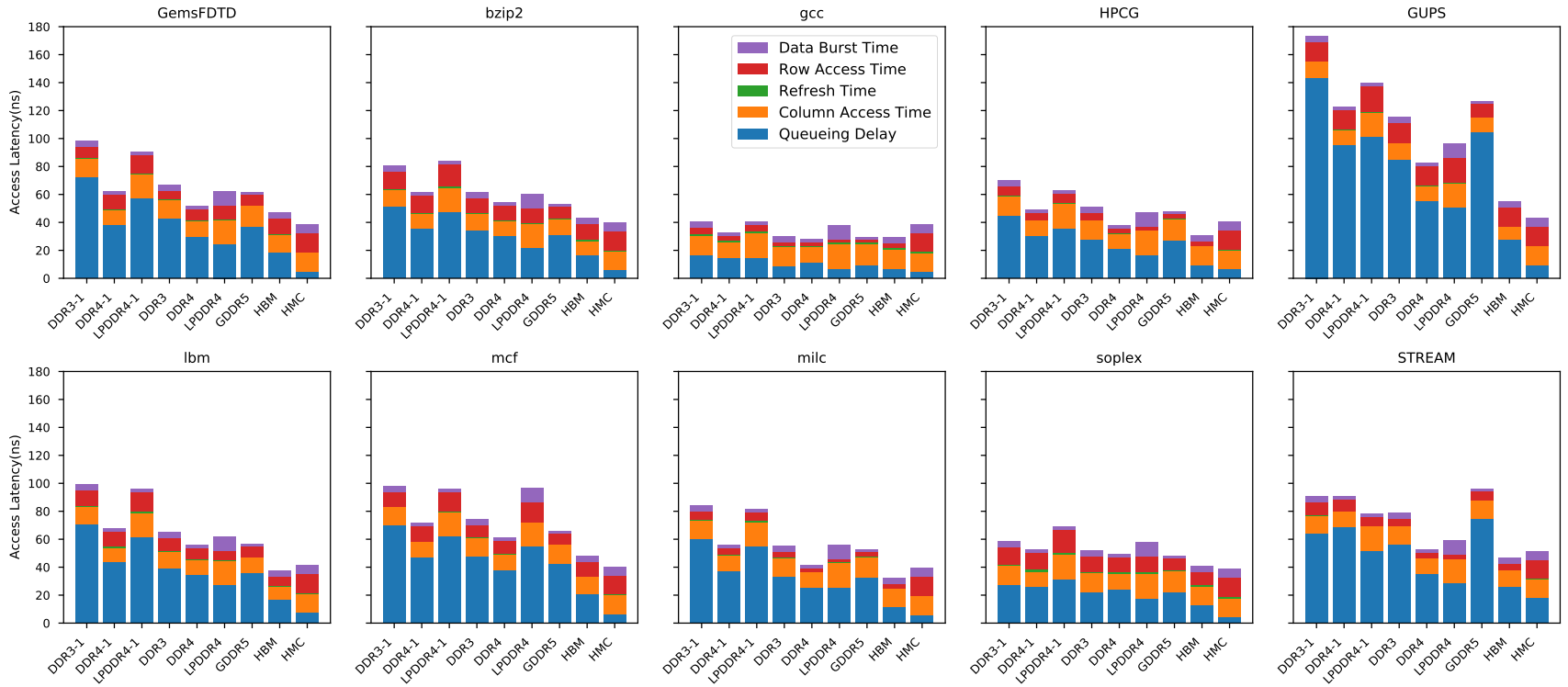


Figure 3.3: *Average access latency breakdown. Each stack from top to bottom are Data Burst Time, Row Access Time, Refresh Time, Column Access Time and Queuing Delay.*

3.3.2 Access Latency Analysis

Average memory latency is broken down in Figure 3.3, indicating the various operations that cause an operation not to proceed immediately. Note that row access time varies with the different row buffer hit rates. In the worst case, where there is no row buffer hit, or the controller uses a close page policy (such as HMC), the row access time would reach its upper bound tRCD. Note also that the highly parallel, multi-channel DRAMs not only overlap DRAM-request latency with CPU execution but with other DRAM requests as well; therefore, these averages are tallied over individual requests.

At first glance, the reason HBM and HMC reduce the average access latency in Figure 3.2 is that they both tend to have shorter queuing delays than the other DRAMs. The reduced queuing delay comes from several sources, the most important of which is the degree of parallelism: HMC has 16 controllers internally; and HBM is configured with eight channels. This matches previous work [6], which shows that high bandwidth must be accompanied by high degrees of parallelism, and we also see it when comparing LPDDR as a DIMM (single-channel) with LPDDR in the quad-channel format: both have exactly the same bandwidth, but the increased parallelism reduces execution time significantly.

HBM and HMC also have additional parallelism from integrating many more banks than DDR3 and DDR4: they have 128 and 256 banks per package respectively, which is 8/16 times more than a DDR3 DIMM. Thus, potentially 8 times more requests can be served simultaneously, and the queuing delay for requests to each

bank is reduced.

Further latency reduction in HMC is due to the controllers attached to it. In contrast with HMC, HBM can exploit open pages. In benchmarks such as *milc*, *HPCG*, *gcc*, *STREAM* and *lbm*, the row access time for HMC is reduced significantly, while HMC has the same constant row access time. Compared to DDR3, DDR4 and GDDR5, which also utilize open pages, HBM has more banks, meaning more potentially opened pages and thus higher row buffer hit rates, which further reduces the access latency. This happens in *STREAM* and *lbm*, where HBM has noticeably lower row access time than DDR3, DDR4 and GDDR5. This can be verified by looking at the row buffer hit rates shown in Figure 3.7.

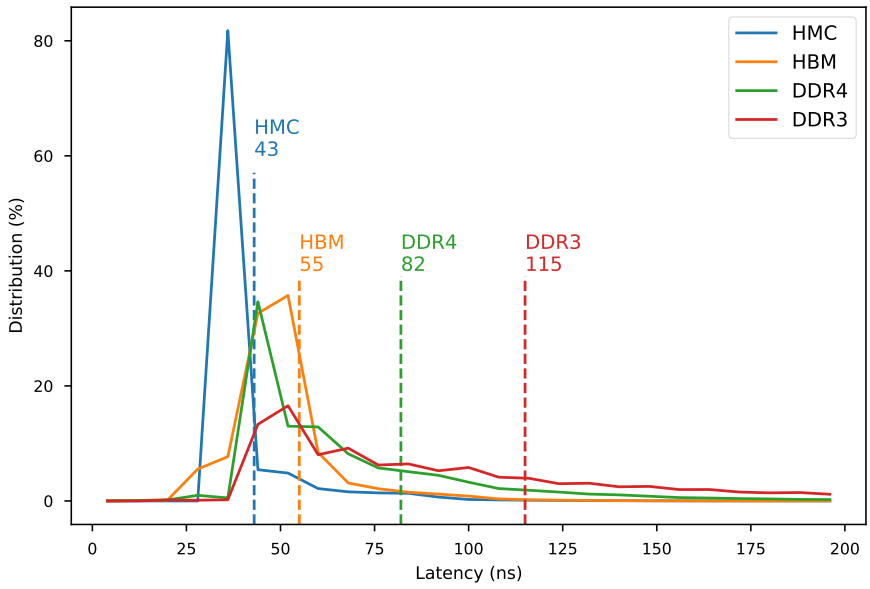


Figure 3.4: *Access latency distribution for GUPS. Dashed lines and annotation show the average access latency.*

Note that HMC exhibits a stable average access latency. From Figure 3.3

we see its average latency around 40ns, ranging from $39ns$ for *soplex* to $52ns$ for *STREAM*. The average is $41ns$ with a standard deviation of $3.9ns$. HBM has the same average latency of $41ns$, but a higher standard deviation of $8.6ns$. This implies the behavior of HMC is more predictable in access latency, and it can be potentially useful in real-time systems, due to the deterministic nature of its close-page scheduling policy. Also note that the average latency of HMC is nearly pushed to its lower limit, as the row access and column access times contribute most of the latency. Queuing time can be improved by increasing parallelism, but improving row and column access time requires speeding up the internal DRAM arrays.

For further insight, we look at the access-latency distributions for several DRAMs, to give an idea of the quality of service and fairness of the DRAM protocols and DRAM-controller schedulers. Figure 3.4 shows probability densities for GUPS running on HMC, HBM, DDR4, and DDR3. The x-axis gives latency values; the y-axis gives the probability for each; and the area under each curve is the same. Thus, HMC, which spikes around 35ns, has an average that is near the spike, and the other DRAMs have average latencies that can be much higher.

3.3.3 Power, Energy, and Cost-Performance

There are two major parts of a DRAM device's power: DRAM core power and I/O power. We use Micron's DRAM power model [45] to calculate the core power; we estimate I/O power based on various sources [46–48] and assume the I/O power for each DRAM is a constant while driving the bus.

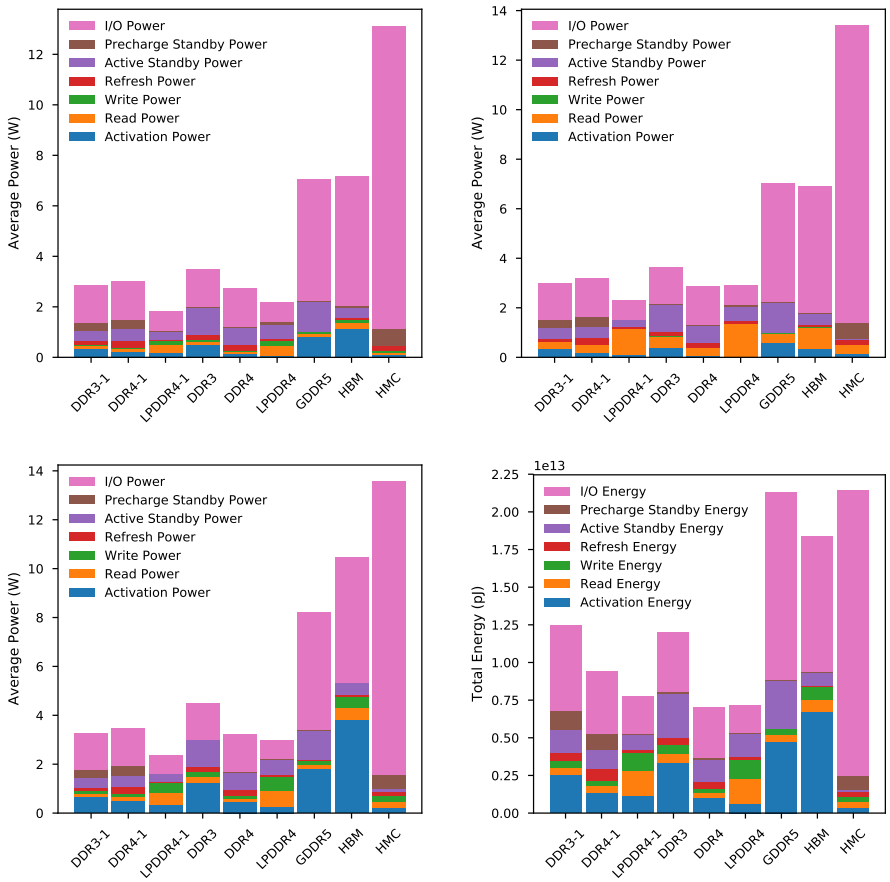


Figure 3.5: *Average power and energy. The top 2 figure and the lower left one show the average power breakdown of 3 benchmarks. The lower right one shows the energy breakdown of GUPS benchmark.*

Figure 3.5 shows the power estimation for a representative group of the benchmarks. I/O power tends to dominate all other fields. The high-speed interfaces of HBM and HMC are particularly power-hungry, driving the overall power dissipation of DRAM system upwards of 10W. HMC has the highest power dissipation, though its DRAM core only dissipates a small portion of its power. GDDR5 also has very high I/O power dissipation, considering its pin bandwidth is less than half that of HBM (48GB/s vs 128GB/s). DRAM-core power varies from application to appli-

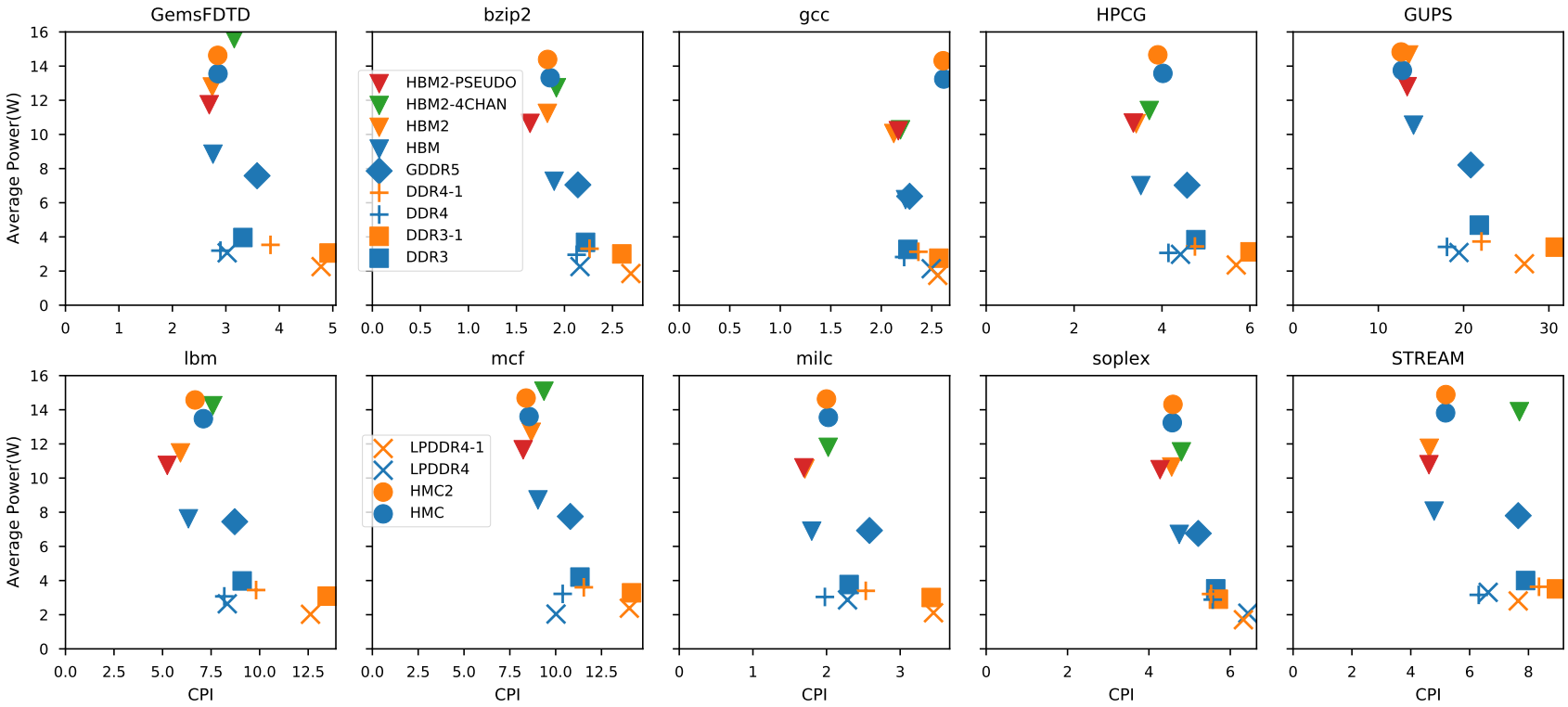


Figure 3.6: *Average power vs CPI. Y-axis in each row has the same scale. Legends are split into 2 sets but apply to all sub-graphs.*

cation. HMC is still very steady, whereas others that adopt open-page policies see varying DRAM-core power. For instance, the core power of HBM can vary from 2 Watts to 5 Watts, with activation power the most significant variable.

Energy. Power is not the only story: energy-to-solution is another valuable metric, and we give the energy for GUPS in the far right graph. While the power numbers range from min to max over a factor of 7x, the energy numbers range only 3x from min to max, because the energy numbers represent not only power but also execution time, which we have already seen is 2–3x faster for the highest-performance and hottest DRAMs. Here we also see the effect of the single-rank vs. dual-rank systems: The single-rank configurations (DDR3-1, DDR4-1) have lower power numbers than the corresponding dual-rank configurations (DDR3, DDR4), because they have fewer DIMMs and thus fewer DRAMs dissipating power. However, the dual-rank configurations require lower energy-to-solution because they are significantly faster.

Power-Performance. Combining power with CPI values from earlier, we obtain a Pareto analysis of performance vs power, as shown in Figure 3.6. We add a few more simulation configurations that were not presented in previous sections: HBM2 running in pseudo channel mode (16 channels), labeled *HBM2-PSEUDO*; HBM2 configured as 4 channels, each 256 bits, labeled *HBM2-4CHAN*; and HMC2 which has 32 internal channels. Note also that, while HMC dissipates twice the power of HBM, HMC2 dissipates little more than HMC since they're configured using same links, whereas HBM2 dissipates noticeably more power than HBM.

In the graphs, the x-axis is CPI, and the y-axis is average power dissipation.

By Pareto analysis, the optimal designs lie along a wavefront comprising all points closest to the origin (any design that is above, right of, or both above and to the right of another is "dominated" by that other point and is thus a worse design). The LPDDR4 designs are Pareto-optimal for nearly all applications, because no other design dissipates less power; and the HBM2-PSEUDO design is Pareto-optimal for nearly all applications, because it almost always has the lowest CPI value. HBM2-PSEUDO is a design from the HBM2 specification in which the 8-channel HBM2 is divided into 2 pseudo channels each; as we have been discussing, this significant increase in parallelism is the type that one might expect to make HBM2 perform even better, and these results show that, indeed, it does.

Some interesting points to note: The improvements in design from 4-channel HBM2 to 8-channel to 16-channel almost always lie on a diagonal line, indicating that the 16-channel design dominates the others. HBM1 is almost always directly below HBM2, indicating that it has roughly the same performance but dissipates less power—this suggests that the 4-core processor is not fully exercising this DRAM, which we show to be the case in the following section. HMC1 and HMC2 have a similar vertical alignment, which suggests precisely the same thing. GDDR5 is almost always dominated by other designs (it is above and further to the right than many other DRAM designs), which indicates that its cost-performance is not as good as, for example, HBM1 (which exceeds it in performance) or DDR4 (which is more power-efficient). The relationship between DDR3 and DDR4 organizations is almost always a horizontal line, indicating that DDR4 improves performance significantly over DDR3, and a modest decrease in power dissipation.

While the power ranges from 2 Watts to 14 Watts for each benchmark, the changes in CPI are usually less significant (discussed in Section 3.3.1). Thus, to deliver the significant degree of performance improvement one would expect of high performance DRAMs like HBM and HMC, a disproportional amount of power is paid. Only in extremely memory-intensive cases, like GUPS and STREAM and manycore CPUs with high core counts (as we estimate in the next section), is the power-performance justified by using stacked memories like HBM and HMC. On the other hand, switching from DDR3 to DDR4 always seems to be beneficial in terms of both power and performance. LPDDR4 not surprisingly has the lowest power consumption and, in its quad-channel configuration, comparable performance to DDR4.

3.3.4 Row Buffer Hit Rate

The row buffer is the set of sense amps holding the data from an opened DRAM row. Accessing the data in a row buffer is much faster than accessing a closed row; therefore, most DRAM controller designers exploit this to improve performance. The only exception in the DRAMs studied in this work is HMC, which embraces a close-page design.

Figure 3.7 shows the row buffer hit rate for each application. GUPS has a near-zero row-buffer hit rate due to its completely random memory access patterns. Other than GUPS, the row buffer hit rates range from 13% to 90%, averaging 43% (39% if GUPS included). HBM and HBM2 have higher row hit rates than other

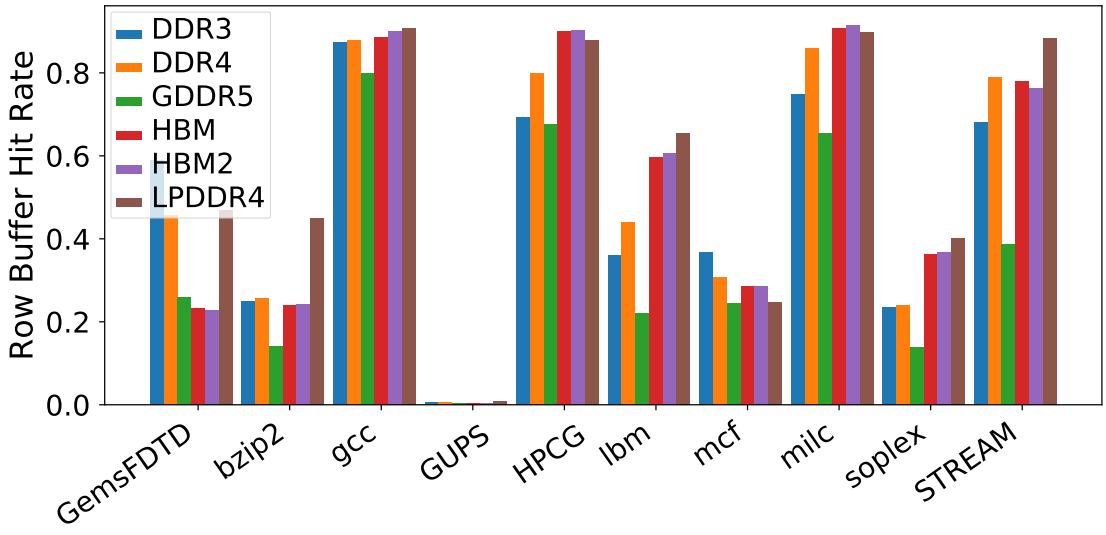


Figure 3.7: *Row buffer hit rate. HMC is not shown here because it uses close page policy. GUPS has very few "lucky" row buffer hits.*

DRAMs in most cases, because they have many more available rows. Note that high row buffer hit rate alone does not guarantee better performance. For example, DDR3 has highest row buffer hit rate in the *GemsFDTD* benchmark, and while the high row buffer hits reduce the row access latency (which can be seen in Figure 3.3), but it also has much higher queuing delay as a result of having fewer banks than other DRAM types.

3.3.5 High Bandwidth Stress Testing

Our studies so far are limited by the scale of the simulated 4-core system and only extract a fraction of the designed bandwidth out of the high-end memories like HBM and HMC. When we observe average inter-arrival times for the benchmarks (average time interval between each successive memory request), they are more than

$10ns$ for most benchmarks—i.e., on average, only one memory request is sent to DRAM every $10ns$. This does not come close to saturating these high-performance main memory systems.

Therefore, to explore more fully the potentials of the DRAM architectures, we run two contrived benchmarks designed to stress the memory systems as much as possible:

1. We modify STREAM to use strides equal to the cache block size, which guarantees that every request is a cache miss.

2. We use Cray's *tabletoy* benchmark, which generates random memory requests as fast as possible and stresses the memory system significantly more than GUPS, which is limited to pointer-chasing. If the average latency for the DRAM is 30ns, GUPS only issues an average of one request per 30ns. Tabletoy issues requests continuously.

3. Lastly, we scale the cycle time of the processor to generate arbitrarily high memory-request rates.

In addition, we use higher-bandwidth HMC configurations. In the previous studies, we used 2-link configurations for HMC and HMC2 at 120GB/s, because 4-link configurations would have been overkill. For the stress-test study, we upgrade links for both HMC and HMC2 for a maximum of 240GB/s and 320GB/s, respectively.

This should present two extremes of high-bandwidth request rates to the DRAM system: one sequential, one random, and as the request rates increase,

these should give one a sense of the traffic that high-core-count manycore CPUs generate.

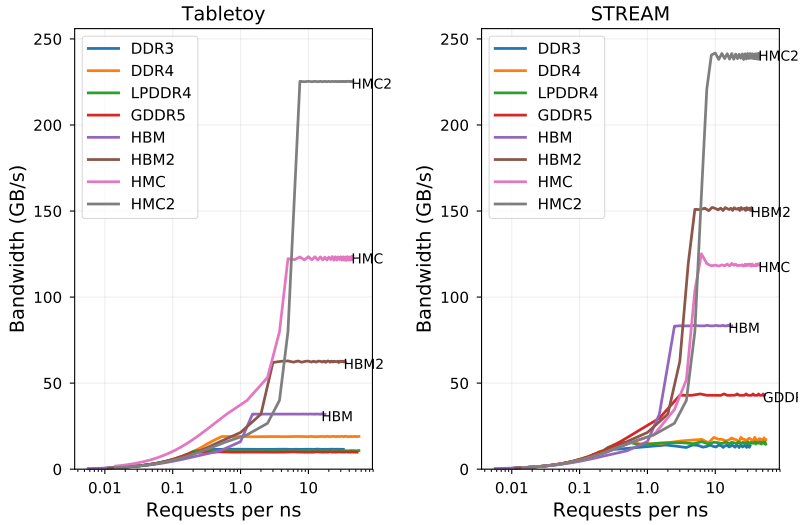


Figure 3.8: *Tabletoy (random), left; STREAM (sequential), right. 64 bytes per request.*

Figure 3.8 shows the results. The left-hand graph shows a random request stream; the right-hand graph shows a sequential stream. The x-axis is the frequency of requests being sent to the memory in log scale. Each request is for a 64-byte cache block. The request rate ranges from $100ns$ per request to more than 10 requests per ns. In the left-hand graph, one can see that all the traditional DDR memories are saturated by $3ns$ per request. HBM and HBM2 reach their peak bandwidths at around 1 to 2 requests per ns, getting 32GB/s and 63GB/s respectively. HMC and HMC2 reach their peak bandwidth at the rate of 4 to 5 requests per ns, reaching 121GB/s and 225GB/s. The difference between HMC and HMC2 here is the number of vaults (channels) they have, 16 vs 32. The effective pin bandwidth of each vault is 10GB/s, meaning that both HMC and HMC2 reach about 75% of the peak internal

bandwidths.

Looking at the sequential results in the right-hand graph, the high-speed DRAMs, e.g. GDDR5, HBM and HBM2, gain significantly more bandwidth than they do with the random stream. HMC and HMC2 only changes slightly, once again shows its steady performance regarding different types of workloads.

The stress tests explain the bandwidth/latency relationship explained at the beginning of the chapter in Figure 3.1. As the request rate is increased, the DRAMs go through the constant region into the linear region (where the curves start to increase noticeably; note that the x-axis is logarithmic, not linear). Where the stress test's bandwidth curves top out, the DRAM has reached its exponential region: it outputs its maximum bandwidth, no matter what the input request rate, and the higher the request rate, the longer that requests sit in the queue waiting to be serviced.

The stress-test results show that any processing node with numerous cores is going to do extremely well with the high-end, multi-channel, high-bandwidth DRAMs.

3.4 Conclusion and Future Work

The commodity-DRAM space today has a wide range of options from low power, to low cost and large capacity, to high cost and high performance. For the single-channel (or single package) system sizes that we study, we see that modern DRAMs offer performance at whatever bandwidth one is willing to pay the power

cost for, as the interface power dominates all other aspects of operation. Note that this would not be the case for extremely large systems: at large capacities, refresh power may also be very significant and dominate other activities. However, at the capacities we study, it is the transmission of bits that dominates power; thus, this provides an important first metric for system design: determine the bandwidth required, and get it.

Our studies show that bandwidth determines one's execution time, even for the modest 4-core CPU studied herein, as the higher bandwidths and, more importantly, the parallelism provided in the high-performance packages, assure that queuing delays are minimized. High-bandwidth designs such as HMC and HBM can reduce end-to-end application execution time by 2–3x over DDRx and LPDDR4 architectures. This translates to reducing the memory overhead from over half of the total execution time to less than 30% of total execution time. The net result: previously memory-bound problems are turned into compute-bound problems, bringing the focus back to architectural mechanisms in the processor that can improve CPI.

Chapter 4: Limitations of Cycle Accurate Models

4.1 Introduction

While working on the characterization study using cycle accurate simulators in Chapter 3, we experienced significantly long simulation times while running the simulations, some tasks can easily take days. This drives us to explore modern simulation techniques and opportunities outside of cycle accurate models.

Long been the prevalent main memory media, the accuracy of DRAM simulation is crucial to the overall accuracy of the simulated system. Like CPU simulators used to be, DRAM simulators are dominantly cycle-accurate models. Often times cycle-accurate DRAM simulators are integrated with CPU simulators to provide accurate memory timings. With CPU simulators moving away from cycle-accurate models so that simulation runs much faster, DRAM simulation speed starts to bottleneck the overall simulation speed. To demonstrate how much time is spent in the DRAM simulators, we run a set of benchmarks with two types of CPU models using the same DRAM simulator, and breakdown the simulation time based on the wall timers we planted in our code. Detailed simulation configuration will be described in Section 4.3. As shown in Figure 4.1, with cycle-accurate out-of-order (O3) CPU model, the DRAM simulator only accounts for 10% to 30% of the overall simulation

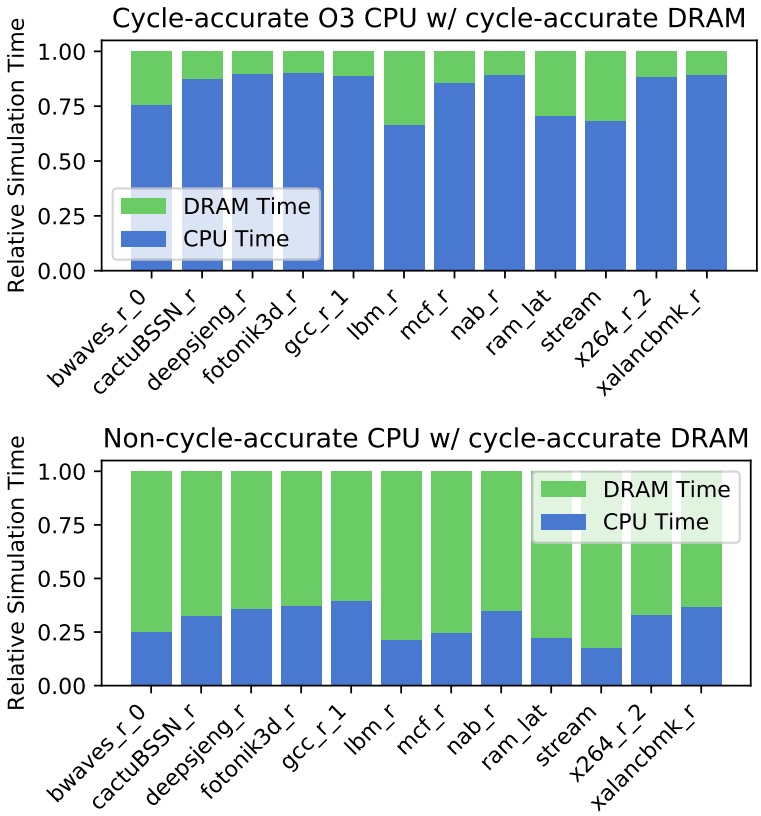


Figure 4.1: *Simulation time breakdown, CPU vs DRAM. Upper graph is cycle-accurate out-of-order CPU model with cycle-accurate DRAM model. Lower graph is modern non-cycle-accurate CPU model. The DRAM simulators are the same in both graph.*

time. But as we switch to a faster CPU model, the DRAM simulation time bloats to 70% to 80% of overall simulation time. Note that the DRAM simulator we use here is already the fastest cycle-accurate DRAM simulator available, which signifies this is a fundamental issue of the cycle-accurate model instead of specific implementation. Also, these results are not limited to specific CPU simulator implementations, because a CPU simulator running at a similar speed will produce similar amount of memory requests in the same time frame, and therefore the DRAM simulator will be

under the same amount of workload and will take the same amount of time to run. Performance aside, some non cycle accurate CPU simulators still manage to work with cycle-accurate DRAM simulators. But the incompatibility causes accuracy issues, which we will further discuss in Section 4.3.3.

So, we believe it is time to review cycle accurate DRAM simulation, discuss its limitations, and explore the alternative modeling techniques.

4.2 Background

To better understand the landscape of architecture simulation techniques, we survey existing alternative modeling techniques of both DRAM and CPU. Note that while this work primarily focuses on DRAM modeling, we will also look into how CPU modeling technique is developing and what can we learn and apply to DRAM modeling. In addition, because a DRAM simulator is often integrated as an interactive memory backend of a CPU simulator, understanding CPU simulator helps create compatible DRAM simulator and thus avoid problems described in Section 4.3.3.

4.2.1 CPU Simulation Techniques

Traditionally, to achieve simulation fidelity, CPU simulators are designed to be cycle-accurate, meaning that just like real processors, the simulator state changes cycle by cycle, and during each cycle, the microarchitecture of CPU (and cache) is faithfully simulated. Other simulation components such as DRAM simulators or

storage simulators also synchronize with the CPU simulator every cycle. While simulating all the microarchitecture details reaches really good accuracy, the downside of this approach is that simulation speed is very slow, especially when CPUs are including more and more cores and a deeper cache hierarchy. Simulations can easily take days, sometimes even weeks, to finish.

A lot of techniques are explored to accelerate cycle-accurate simulations, for instance, checkpointing, which saves the simulator and program state at certain point to a file and allows the simulator to recover from that checkpoint later with the exact same state. This is mostly used to skip the warmup period and make sure simulations start at the same state. Similarly, some CPU simulators use a simpler, non-cycle-accurate model to fastforward the simulation to a warmed-up state and then switch to cycle-accurate model for further simulation.

Some researchers such as [49] take a statistical approach, which instruments and samples the simulated workload, uses statistical methods to identify distinctive program segments, and then extracts these distinctive segments for future simulation. The extracted segments, which are typically called simulation points, can be then simulated with a cycle-accurate simulator. This way, the simulation time is cut short by simulating fewer instructions, instead of improving the simulator.

More recently, CPU researchers are moving away from the cycle-accurate model due to its scalability issues. Several approximation models are proposed and implemented. For example, SST, Graphite [50] and Gem5 Timing CPU Model employs One-IPC model, meaning that every instruction is one cycle in the pipeline. Sniper [51] and ZSim [14] use approximation models for IPC which allows them to

simulate out-of-order pipelines with relatively faster speed. Another benefit of applying this approximation model is that CPU cores and caches can be efficiently simulated in parallel, which allows multi-core, even many-core, CPU simulation applicable with decent scaling efficiency.

4.2.2 DRAM Simulation Techniques

Before cycle-accurate simulators were adopted en masse, researchers applied very simplistic models for DRAM simulations. For example, the fixed-latency model assumes all DRAM requests take the same amount of time to finish, which completely ignores scheduling and queueing contentions that may cause significantly longer latency. There are also queued models that account for the queueing delay, but they fails to comply with various DRAM timing constraints. Previous studies such as [52] have shown that such simplistic models suffer from low accuracy compared to cycle-accurate DRAM models.

Then came along cycle accurate DRAM models, such as [20, 22, 23, 53, 54] and DRAMsim3. These cycle-accurate DRAM simulators improved DRAM simulation accuracy, some are also validated by hardware models, but as we have shown, they started to lag the simulation performance.

Other than cycle-accurate models, there are also event based models such as [55, 56]. Event based models do not strictly enforce DRAM timing constraints, and can accelerate the simulation if the events are not frequent. But just as [56] pointed out, when memory workloads gets more intensive, memory events will be

as frequent as every cycle, and therefore will undermine the advantage of the event-based approach.

Finally, there are analytical DRAM models such as [57, 58]. [57] presents a DRAM timing parameter analysis but does not provide a simulation model. The model in [58] provides predictions on DRAM efficiency instead of per-access timing information. These analytical models provide insights on the timing parameters and high level interpretations, but have limited usage by design.

4.3 Empirical Study

In this section we setup our simulation framework to quantitatively evaluate DRAM models on simulation speed and accuracy. Table 4.1 shows our simulation setup.

Table 4.1: *Simulation Setup*

CPU	Gem5 Timing CPU model, x86 architecture
Core	4GHz, IPC=1
L1 I-Cache	per-core, 32KB, 4-way associative, 64 Byte cache line, LRU
L1 D-Cache	per-core, 64KB, 4-way associative, 64 Byte cache line, LRU
L2 Cache	private, MOESI protocol, 256KB, 8-way associative, 64 Byte cache line, LRU
L3 Cache	shared, MOESI protocol, 2MB, 16-way associative, 64 Byte cache line, LRU
Main Memory	Dual-rank DDR4: 1, 2, 4 channels. 8-channel HBM2.
Workloads	A subset of SPEC CPU2017 benchmarks, STREAM, and memory latency benchmark (*ram_lat*).

We choose Gem5 not only because of its reputation for accuracy, but also

because it supports multiple CPU models and DRAM models and can be easily swapped. This allows us to directly compare two different models, whether they're CPU models or DRAM models, while keeping all other components of the simulation the same. And therefore we can fairly compare and evaluate different models.

We have two CPU model choices here. First is out-of-order (O3, or DerivO3) CPU, that faithfully simulates the details of the core architecture, but only simulates at the rate of tens of thousands instructions per second on the host machine. The other is Timing CPU model; this is a One-IPC core model, which does not offer core microarchitecture simulation, but runs more than 10 times faster than the O3 CPU model. Note we only use this Timing CPU model for simulation speed experiments, in which case it represents other CPU simulators running at similar rate. For all accuracy evaluations, we use O3 CPU as it is the most accurate and reliable option we have.

For DRAM models, we use DRAMsim3 as the representative cycle-accurate simulator, because it offers the best simulation speed, and it is also hardware validated. For event based model, we choose [55], because it is conveniently integrated into Gem5 and offers similar DRAM protocols to DRAMsim3 that allows us to directly compare against. The DDR4 configuration in both models is single channel, dual rank, and has the same timing parameters. The HBM configuration in both models is 8 channel and 128 bits wide each.

To test a wide range of memory characteristics, we use a subset of SPEC CPU2017 benchmarks that are most representative according to [59]. We also include $STREAM$, which is very bandwidth sensitive, and ram_lat, an LMBench-like

memory benchmark that is latency sensitive. These benchmarks will show us the full spectrum of memory characteristics and behaviors.

4.3.1 Quantifying DRAM Simulation Time

First we experiment how much simulation time is spent in the DRAM simulator versus the CPU simulator. The two graphs in Figure 4.1 were obtained by using O3 CPU model and Timing CPU model respectively and have the same DRAMsim3 HBM backend.

Note that because HBM has 8 channels, and each channel has an independent DRAM controller, and it therefore takes more time to simulate HBM than a regular 1 channel DRAM. To quantify how the number of channels affects simulation time, we sweep 1, 2, 4 channels of DDR4 with Timing CPU and show the absolute simulation time in Figure 4.2.

It can be seen that even with only one channel of DDR4, the cycle-accurate DRAM simulator still accounts for an average 40% of overall simulation time with a minimum of 30% and a maximum of 56%. For two channel DDR4, DRAM simulation time ranges from 46% to 69% with an average of 53%. For 4 channels, the min, max and average number are 62%, 81% and 68% respectively. While these numbers are produced with a single simulated core, modern CPU simulators such as [13, 14, 50, 51] can utilize multiple host cores to simulate multiple simulated cores, making the core simulation time scalable. Therefore, we can still conclude that DRAM cycle-accurate simulation does not scale with regards to the number of

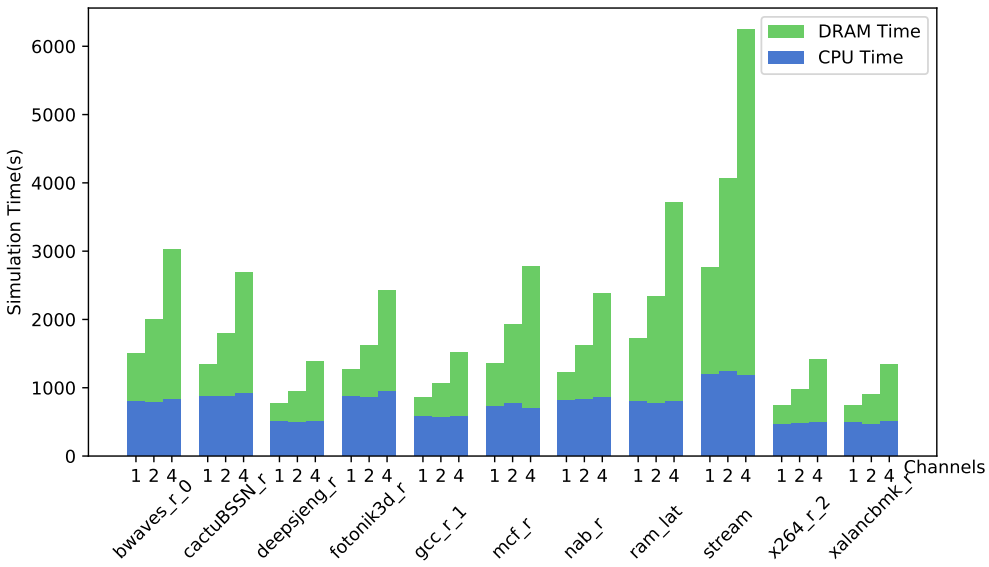


Figure 4.2: *Absolute simulation time breakdown of Timing CPU with 1, 2, and 4 channels of cycle-accurate DDR4. The bottom component of each bar represents the CPU simulation time and the top component is the DRAM simulation time.*

channels, and it takes a significant proportion of simulation time even with only 1 DRAM channel.

4.3.2 Synchronization Overhead

The nature of cycle accurate simulation requires A) the CPU simulator to synchronize with the DRAM simulator every DRAM cycle and B) when there are multiple channels within the DRAM, they have to synchronize with each other every cycle. As we will see later in Chapter 5, these are huge issues when moving to parallel simulation. Moreover, another aspect of synchronization problem is the performance cost when integrated into other parallel simulation framework such as SST. As a simulation framework, SST can integrate individual component simulators (e.g.

DRAMSim2) and provide an interface for each component to communicate with each other. Doing so allows SST to distribute simulated components to different cores or machines and simulate them in parallel. The implementation of the wrapper interface for DRAMSim2, for instance, treats each cycle of DRAM as an event. This means the simulation framework, when a cycle accurate DRAM simulator is present, has to synchronize with the DRAM simulator every single cycle, even if the synchronization event could be a costly MPI call over the wire. At this point it is hard to justify running the DRAM simulator in a separate thread or process in such a simulation framework.

4.3.3 Compatibility: A Case Study of ZSim

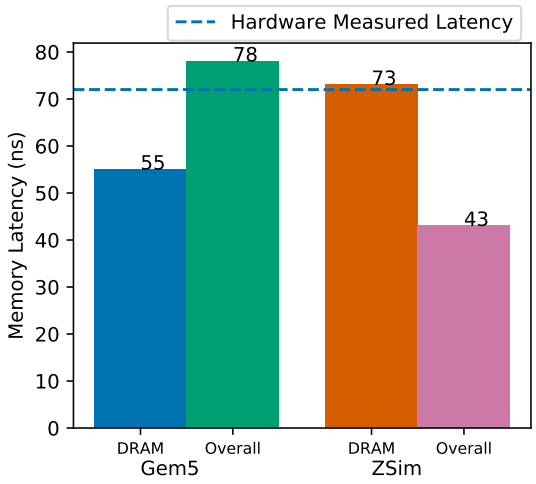


Figure 4.3: *DRAM latency and overall latency reported by Gem5 and ZSim.*

Besides the poor simulation performance, cycle accurate DRAM simulation also poses compatibility issues when integrating with modern CPU simulators or

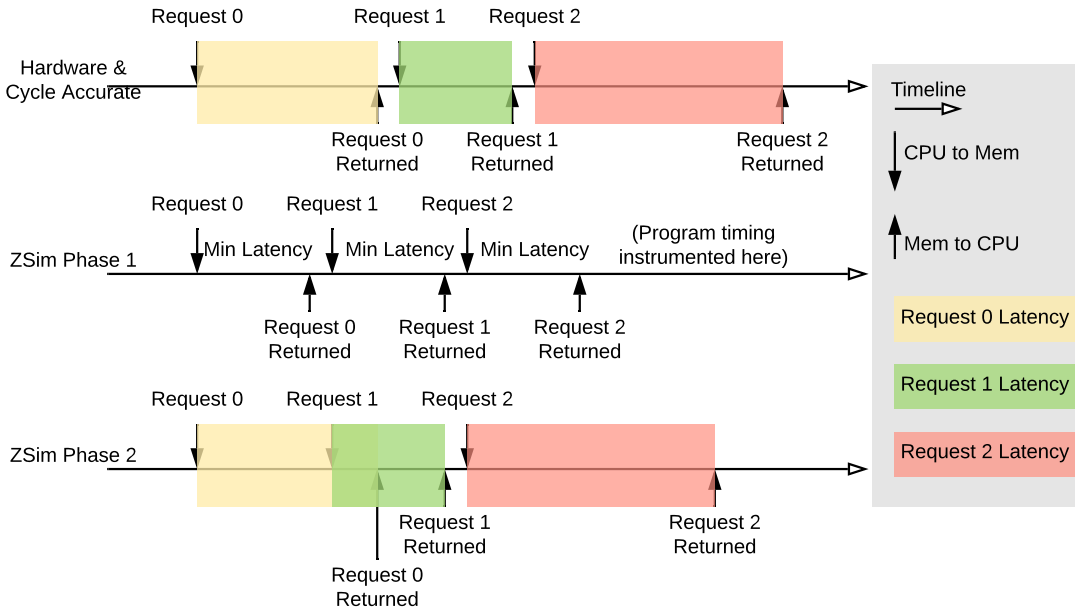


Figure 4.4: *ZSim 2-phase memory model timeline diagram compared with real hardware/cycle accurate model. Three back-to-back memory requests (0, 1, 2) are issued to the memory model.*

frameworks, especially those that rely on parallel simulation for speed, as cycle accurate models require synchronization every cycle, which will create huge overhead for parallel performance. For example, [50, 51] do not include a cycle accurate main memory backend at all. [14] supports a cycle accurate memory backend, but as we will see soon, it has its issues when integrating a cycle accurate memory backend.

The problem was first discovered by [60], who observed a memory latency error of about 20ns when they tested a memory latency benchmark. But [60] did not answer where this 20ns missing latency comes from, as they suspected it came from the cycle accurate DRAM simulator they were using. We will analyze this situation and provide a conclusive answer to this question.

To replicate the issue independently, we developed a simplified version of LMBench(*ram_lat* we referred in Table 4.1) that randomly traverses a huge array, and measured the average latency of each access. When the array is too large to fit in the cache and most accesses go to DRAM, the average access latency will include the DRAM latency. The benchmark inserts time stamps before and after the memory traversal, and it uses them to determine the overall latency of a certain number of memory requests, dividing by the number of requests to obtain average memory latency. This average memory latency consists of cache latency and DRAM latency, and thus we use the term ***overall latency*** in the following discussion.

Like [60], we ran this benchmark natively on our machine to obtain "hardware measured" latency(72ns), then ran it in ZSim along with DRAMSim2 as the DRAM backend, and we were able to reproduce similar results as [60]. That is, the *overall latency* (43ns) is 29ns lower than hardware measurement (72ns). To determine whether this is a ZSim specific issue or DRAM simulator issue, we ran the same benchmark in Gem5 with the same cache and DRAM parameters, and this time, the *overall latency* is 78ns, much closer to our hardware measurement. So we conclude this is a ZSim specific issue not a DRAM simulator issue. We then further looked into the simulator statistics, and found that the DRAM latency reported by the DRAM simulator in Gem5 is 55ns, which makes sense as the *overall latency* (78ns) should be a combination of DRAM latency (55ns) and cache latency (23ns). However, in ZSim, the DRAM latency reported by the DRAM simulator is 73ns, much higher than *overall latency*, which makes no sense. Figure 4.3 shows these results. This again confirms that the issue lies within the ZSim memory model.

The way ZSim memory model works is, it has two phases of memory models, the first phase is a fixed latency model that assumes a fixed "minimum latency" for all memory events. The purpose is to simulate instructions as fast as possible, and generate a trace of memory events. After the memory event trace is generated, the second phase kicks in, and that's when the cycle accurate DRAM simulator actually works, the cycle accurate simulation uses the event trace as input and updates latency timings associated with these events.

For instance, Figure 4.4 demonstrates how ZSim memory model handles memory requests differently from hardware/cycle accurate models. Suppose there are 3 back-to-back memory requests(each relies on the finishing of previous one). In real hardware or a cycle accurate model, each memory request's latency may vary, and the next request cannot be issued until the previous request is returned. In ZSim Phase 1, all requests are assumed to be finished with "minimum latency", and therefore finish earlier than they should. Then in ZSim Phase 2, cycle accurate simulation is performed, more accurate latency timing is produced by cycle accurate simulator and all 3 requests update their timings. But even if all memory requests obtain correct timings in Phase 2, unfortunately, when the simulated program, like our benchmark, has instrumenting instructions such as reading the system clock, it will obtain the timing numbers during Phase 1, which can be substantially smaller. This is why the *overall latency* is much smaller than DRAM latency.

So in other words, the "minimum latency" ZSim parameter will dictate the latency observed by the simulated program. To verify this claim, we run the same simulation with different "minimum latency" parameters, and plot them against the

benchmark reported latency and DRAM simulator reported latency altogether, as in Figure 4.5.

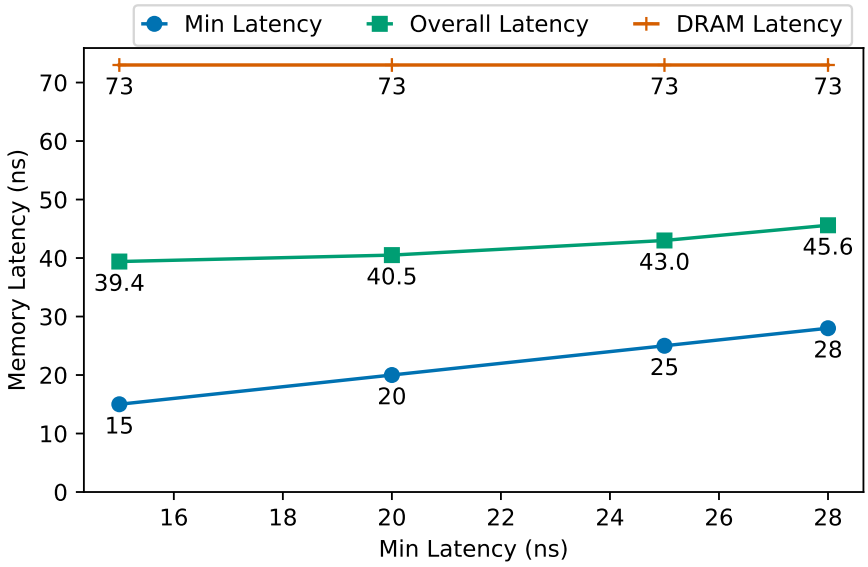


Figure 4.5: *Varying ZSim "minimum latency" parameter changes the benchmark reported latency, but has little to none effect on DRAM simulator.*

It can be seen in Figure 4.5 that, while we increase the "minimum latency" parameter, the overall latency pronounced by benchmark increases correspondingly, while the DRAM simulator reported latency keeps steady.

The reason that ZSim has to use a two-phase memory model is that it has to have a memory model that can give a latency upon first sight so that it can generate an event trace during an interval. The only model that is able to do so is a fixed latency model, but apparently it is not accurate enough and cannot handle dynamic contention, and therefore ZSim requires a second, cycle accurate phase to correct the timings. In addition to this self-instrumenting error, the broader issue is that during the second phase, the memory requests received by the memory controller

will have an inaccurate inter-arrival timing produced by Phase 1, which may alter the results of cycle accurate simulation results. In other words, the inaccuracy in Phase 1 can lead to further inaccuracy of Phase 2 memory simulation.

The root cause for the convoluted memory model of ZSim, and other fast simulators that do not support cycle accurate DRAM simulators is, cycle accurate DRAM simulator is no longer compatible with these fast abstract simulation models, and there are as yet no good alternatives that work with these abstract models.

4.3.4 Event-based Model

As we stated earlier, even based DRAM models typically offer better simulation performance than cycle-accurate models. But a general concern is the accuracy implication. To obtain a comprehension of event based model accuracy, we compare the event based DRAM model [55] included in Gem5 with DRAMsim3. Both simulators are integrated into the same Gem5 build so that we can conduct a fair comparison of the same CPU, cache, and benchmark with only the DRAM model being different. For both DRAM models, we run all the benchmarks with a DDR4 profile and an HBM profile. The DDR4/HBM timing parameters are configured to be the same in both DRAMsim3 and the event based model. The CPU model we use to evaluate accuracy is the Gem5 O3 CPU model, which provides deterministic, reproducible results. We use the CPI numbers obtained by DRAMsim3 backed simulations as baseline, and plot the relative CPI of event based simulations in percentage, shown in Figure 4.6.

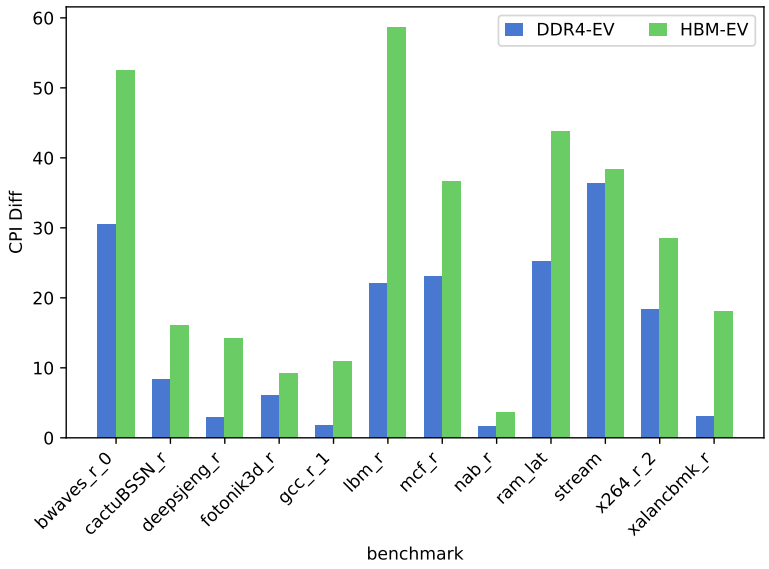


Figure 4.6: *CPI differences of an event based model in percentage comparing to its cycle-accurate counterpart. DDR4 and HBM protocols are evaluated.*

The CPI difference ranges from 3% to almost 60% across all benchmarks. In general, less memory-intensive benchmarks tend to have lower CPI differences. The DDR4 event based model averages a 15% CPI difference, and the HBM event based model averages a 28% CPI difference from their cycle accurate counterparts. While we cannot conclusively say the difference in CPI translates to inaccuracy as the event based model implements a different scheduling policy for the controller, the CPI difference is much higher than those between cycle accurate models. So even though event based models can be several times faster than cycle accurate models, one has to make sure the accuracy is acceptable for the kind of workload he or she wants to simulate.

4.4 Conclusion

In this chapter we empirically discussed the limitations of cycle accurate DRAM simulation models. We showed that while still being the most accurate model, cycle accurate DRAM models cannot keep up with the trend of architecture simulator development in terms of simulation performance and model compatibility. As for alternative modeling solutions, the event based model we evaluated here does not have convincing accuracy. To overcome these limitations of cycle accurate models, we propose our solutions in Chapter 5 and Chapter 6.

Chapter 5: Parallel Main Memory Simulation

To address the scalability issue of multi-channel DRAM simulation raised in Chapter 4, we explore the possibility and pathways of simulating DRAM in parallel.

From the architecture's point of view, in a multi-channel DRAM system, each DRAM channel can operate on its own and is independent from other channels. So theoretically the DRAM channels can be naturally simulated in parallel without having to synchronize with each other. Although in practice, we still need an interface to handle inputs and outputs such as taking requests from traces or a front-end simulator, mapping DRAM addresses to channels, aggregating statistics across channels, and so on.

5.1 Naive Parallel Memory Simulation

5.1.1 Implementation

We first start a naive parallel implementation. Based on DRAMsim3, we developed a multi-threaded DRAM simulator using OpenMP [61]. OpenMP is an industry standard parallel programming diagram, it allows non-intrusive implementation of multi-threading without changing the source code of a program. However,

to avoid data races, synchronization overhead, and parallelization overhead, we do need to optimize the code for best parallel performance.

We implement the following optimizations to the original serial DRAMsim3 code:

- We eliminate shared writable data structures across channels inside the parallel region of the program, so that there will be no locking or synchronization mechanisms needed when executing the parallel code. This helps us minimizes the parallelization overhead.

- The program will automatically choose the number of threads based on the user's setting or number of channels to be simulated, whichever is smaller, to make sure that no more threads than the number of channels are spawned.

- We use *static* scheduling of OpenMP as it has low overhead when dealing with small amount of data and similar amount of work for each thread.

Figure 5.1 is a system diagram of how we partition the simulator into serial and parallel regions. In this naive implementation, we have a unified input interface that takes requests and map each to its corresponding channel; then we start the parallel simulation, and each channel simulates one cycle. This includes further address translation, scheduling, updating bank states and timings, and update of statistics of that channel. Finally after each cycle we enters another serial region, which returns any completed requests from each channel, and optionally aggregates statistics from all channels.

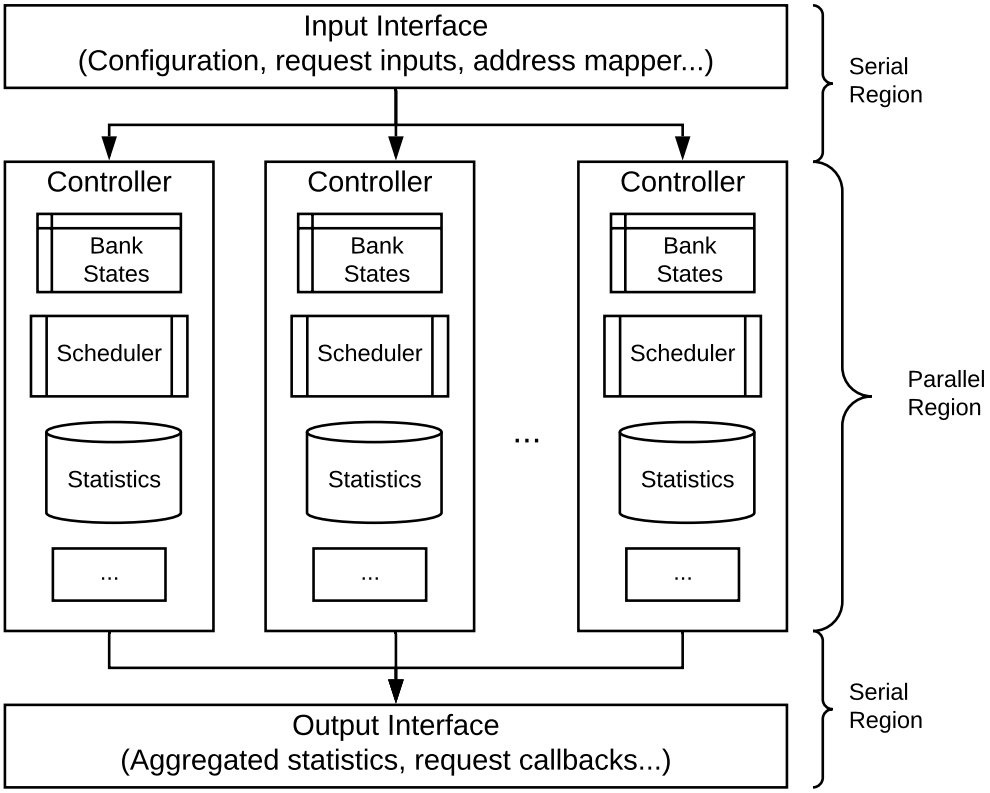


Figure 5.1: *Parallel DRAM simulator architecture.*

5.1.2 Evaluation

First, we examine the execution speedup of the parallel DRAM simulator over trace inputs. The trace frontend contributes little overhead to the overall simulation time. We can also load traces to keep the DRAM simulator busy all the time to maximize the simulation time spent in the DRAM simulator. Therefore it is the ideal scenario for testing parallelization speedup.

We run two types of trace, stream and random, for 10 million cycles on a 8-channel HBM configuration. We compare the simulation time of running the simulation in 1, 2, 4, and 8 threads respectively.

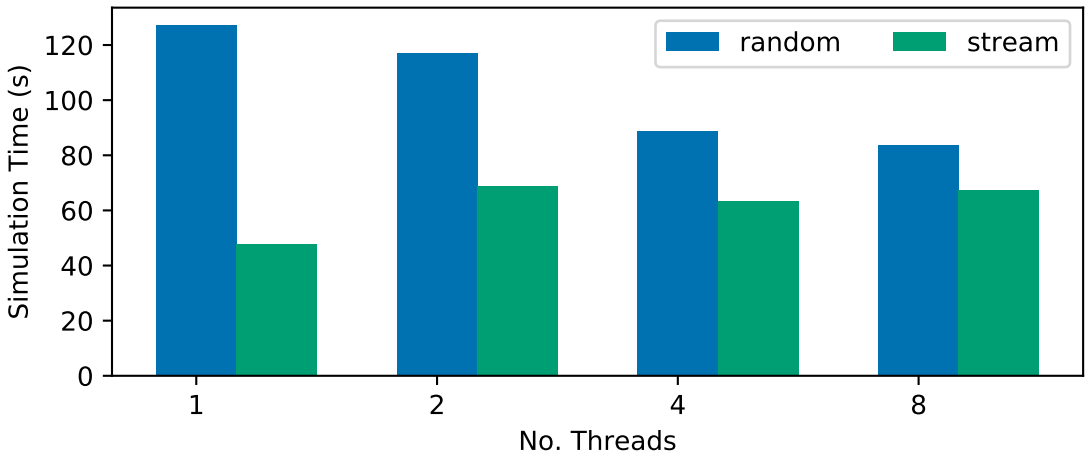


Figure 5.2: *Simulation time using 1, 2, 4, and 8 threads.*

The results are intriguing: as shown in Figure 5.2, the random trace simulations get slightly speed up when using more and more cores, and it eventually settles for 60% of single thread simulation time at 8 threads. The stream trace simulations, however, take a performance hit when using multiple threads, and the simulation usually runs more than 1.2 times slower when using multiple threads than the single thread version.

For perspective, the single thread stream trace simulation only takes 48 seconds while the single thread random trace simulation takes 127 seconds for the same number of simulated cycles. This makes sense given that when simulating the stream trace, it is easier for the scheduler to find the next available request in the request queue whereas for the random trace the scheduler typically has to search through the entire queue. It means there is much more "work" to be done in each simulated cycle for a random trace. This is important for the multi-thread performance

Regardless of the reasoning, running the simulation 40% faster using 8 threads

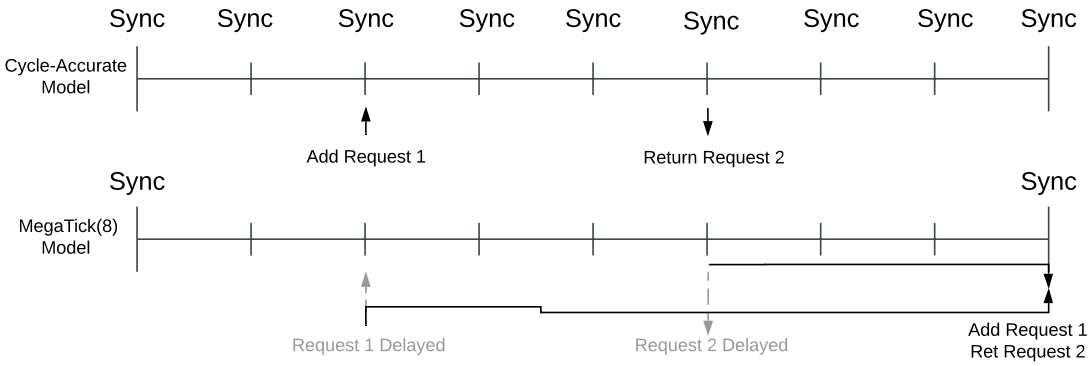


Figure 5.3: *Cycle-accurate model (upper) vs MegaTick model (lower)*

at best is underwhelming, and running slower on more threads is disappointing. In the next section, we propose a new scheme to break this performance barrier.

5.2 Multi-cycle Synchronization: MegaTick

The problem we encountered in Section 5.1.2 is that the benefit of multi-threading is overshadowed by the overhead. The core issue is, a cycle-accurate DRAM simulator has to synchronize every DRAM cycle, creating very frequently reoccurring synchronization overhead each scheduled cycle, which eventually consumes all the speedup gained from multi-threading.

To address this problem, we propose a semi-accurate approach, that is, to synchronize the DRAM simulator and the serial interface every few DRAM cycles, instead of every DRAM cycle, and in between two synchronization points the DRAM is still simulated cycle-accurately. We call each synchronization period a MegaTick. Figure 5.3 shows the difference of the MegaTick model versus a cycle-accurate model using 8-cycle MegaTick as an example. In a cycle-accurate model, requests can be

issued to and returned from DRAM every DRAM cycle. whereas in the MegaTick model, these events can only happen on a MegaTick boundary, meaning that these events will have to be delayed until the next synchronization point.

Using MegaTick allows more DRAM cycles to be simulated in parallel continuously, which may increase the threading efficiency and thus accelerate the simulation. On the other hand, if paired with a CPU simulator, from the CPU's perspective, it means the CPU has to hold outstanding requests longer to issue to the DRAM, and receive the response from DRAM later than it should. This could mean accuracy loss. So the question becomes, how many DRAM cycles should there be in a MegaTick, what is the accuracy implication when using MegaTick, and how much speedup can we gain using this technique?

To answer these questions, we set up simulations in similar configurations as Section 5.1.2 but change the CPU-DRAM simulator interface so that they now use MegaTick synchronization. We vary the number of MegaTicks from 2, 4, 8 to 16 DRAM cycles, and compare the simulation speed and several accuracy metrics against cycle-accurate simulations as shown in the following section.

5.2.1 Performance Evaluation

We again run trace simulations with the same setup as Section 5.1.2 except this time we use MegaTick synchronization. We test MegaTick values from 2, 4, 8 to 16 and measure the simulation time.

The results can be seen in Figure 5.4. For the random trace, previously we were

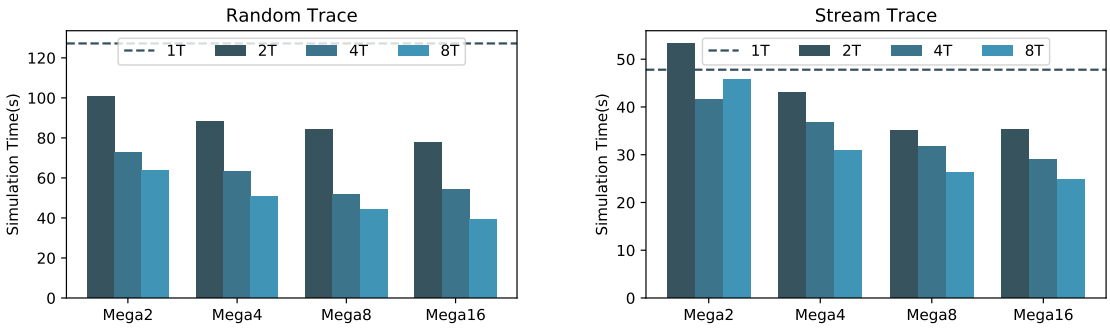


Figure 5.4: *Simulation time using MegaTick synchronization. Random(left) and stream(right) traces.*

only able to get 40% faster simulation time at best comparing to single-thread baseline. Using MegaTick we are able to run the simulation 3.3 times faster. Although we do start to see a diminishing return after 8 cycles per MegaTick. Similarly for the stream trace, we were not able to run the simulation faster, because each parallel region is too short. With MegaTick we are able to run twice as fast compared to the single thread version.

While trace simulations prove that MegaTick synchronization does indeed improves simulation speed, it is time to further validate whether MegaTick can improve simulation speed when using DRAM simulator that is integrated with a CPU simulator. We integrate the MegaTick DRAMsim3 with Gem5 TimingCPU Model with 3 levels of cache (same parameters as in Table 4.1), and use an 8-channel HBM as memory backend. We run all the parallel simulations with 8 threads and compare the simulation time of using various of MegaTick values. We normalize the simulation time against the single-thread cycle-accurate setup. Figure 5.5 shows the results.

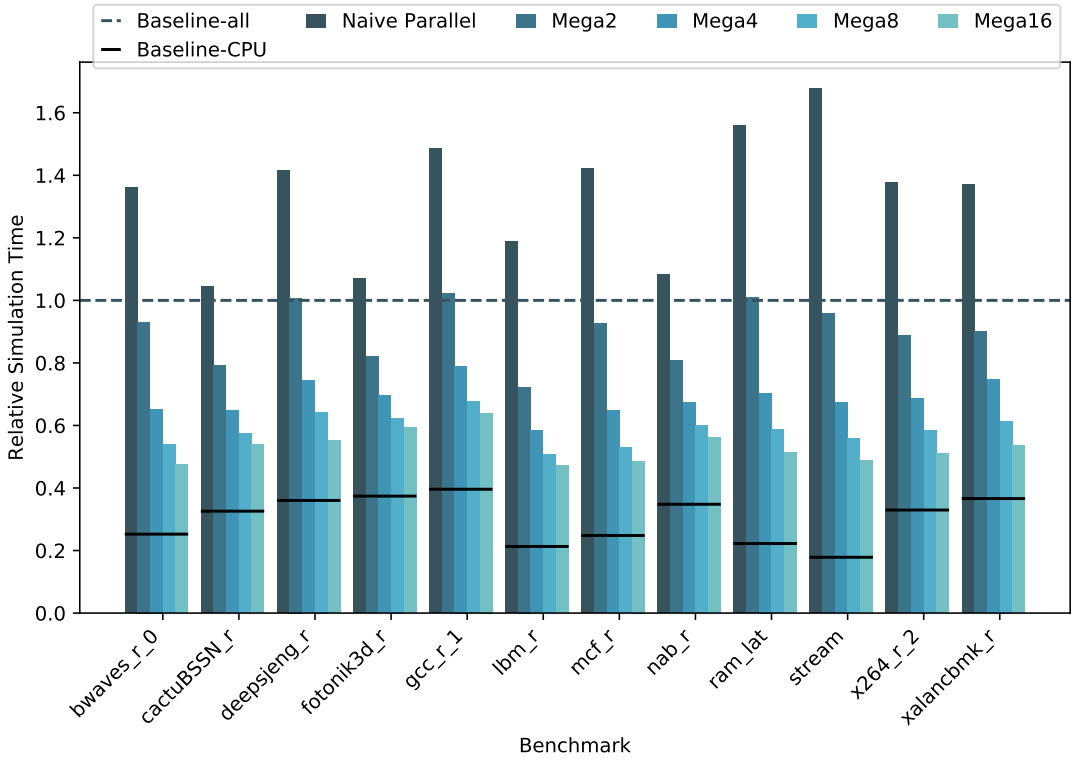


Figure 5.5: *MegaTick relative simulation time to serial cycle-accurate model ("Baseline-all") with relative CPU simulation time for each benchmark("Baseline-CPU"). 8-channel HBM. 8 threads for parallel setups.*

Similar to trace simulations, cycle-accurate multi-thread simulations are always slower than single-thread simulations, in this case they average a 34% slowdown. "Mega2" (synchronizing every 2 cycles) seems to be a break-even point for using parallel simulations in this case, and though some benchmarks are slightly slower than the single-thread version, the overall average speedup is 10% over the single-thread simulations. "Mega4" setup sees the largest performance jump relative to the baseline, where we observe and average of a 31% performance boost. "Mega8" and "Mega16" further improves the overall simulation speed by 41% and

47% respectively, with several cases shortening the overall simulation time by half. Again, we are seeing a diminishing return after "Mega16" and therefore we did not go beyond it. In addition, comparing to the CPU simulation time, MegaTick reduces the DRAM simulation time to less or equal to CPU simulation time for most benchmarks. This makes the multi-channel DRAM simulation time no longer the dominant component of the overall simulation time, accomplishing part of the

5.2.2 Accuracy Evaluation

While MegaTick does not require any changes to the internals of the cycle-accurate memory controller model, as we previously pointed out in Section 5.2, the latency of incoming and outgoing memory requests may be increased because of the longer synchronization period with the CPU.

Because the main memory system is at the bottom of the memory hierarchy, any changes we make can affect the performance of everything up the chain like caches and processors. Besides, even though MegaTick does not change the internals of the cycle-accurate memory controller, the interface changes may or may not have impact on the behaviors of the controller. Therefore, we use a comprehensive set of metrics on different components to examine the accuracy impact of the MegaTick model.

For overall system accuracy and core accuracy impact evaluation, we use CPI(cycles per instruction). For cache impact, we mainly rely on last level cache (LLC) statistics, as it directly interacts with main memory. In our simulation setup

we have a 3 level cache, and therefore we choose the L3 miss latency as the metric. For memory controllers, the interface timing change is directly perceived as the changes on inter-arrival timings. The inter-arrival timings may alter scheduler behavior, as some of the scheduling decisions are based on the timings of the requests, and it ultimately reflects on the bandwidth or throughput of the memory system.

To best evaluate these metrics, we use the deterministic, cycle-accurate out-of-order CPU model provided by Gem5 (DerivO3 CPU model), along with a 3-level cache hierarchy, and swap only the main memory models for each run. We compare our proposed MegaTick models against cycle-accurate model (DRAMSim3). We configure these 2 models with the same parameters wherever applicable, and use the cycle-accurate model as the baseline for comparison. The benchmarks and memory model configurations are the same as Section 5.2.1.

5.2.2.1 CPI Accuracy Evaluation

First, we compare the overall system accuracy in terms of CPI. We use the CPI numbers from cycle-accurate simulation results as baseline, and calculate the percentage error of CPI from other configurations. The results are shown in Figure 5.6.

As can be seen in Figure 5.6, The absolute average CPI error for MegaTick models are 1.5%(Mega2), 2.7%(Mega4), 4.7%(Mega8) and 9.5%(Mega16). The worst cases are Mega16 results which exceed 20% error, which is not ideal. But Mega4 and Mega8 both have acceptable accuracy, especially for the benchmarks

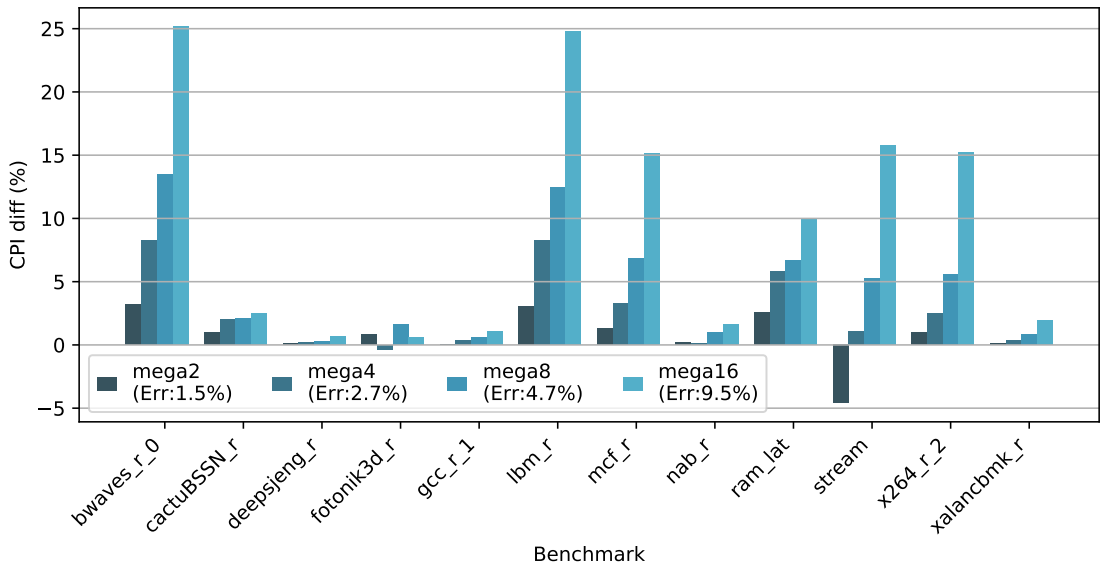


Figure 5.6: *CPI error comparison of MegaTick model. Cycle-accurate results are the comparison baseline(0%).*

that are not very memory intensive.

Most of the MegaTick models have higher CPIs than cycle-accurate model because of the increasing memory latency. But there is one exception of the STREAM benchmark with the Mega2 model, which has a lower CPI than the cycle-accurate mode. To verify this outlier result, we discovered that, like other MegaTick models, this configuration also has higher average memory latency, but also has a higher average memory bandwidth than the cycle-accurate model. The possible reason could be this particular synchronization mode accidentally favors the controller scheduling and thus produces higher bandwidth. STREAM is a more bandwidth sensitive benchmark and thus has a lower CPI in this case.

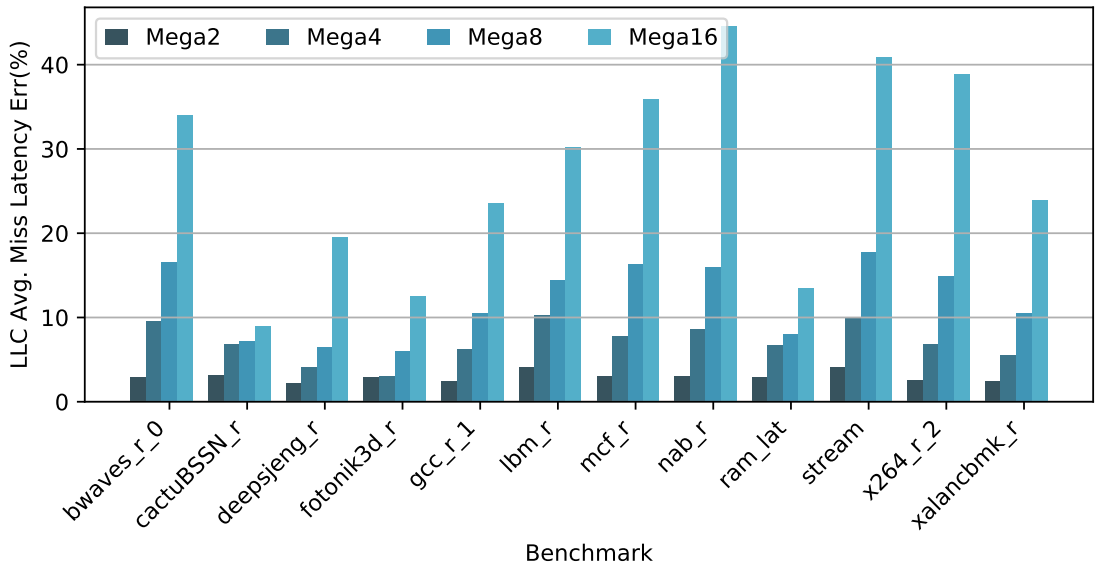


Figure 5.7: *LLC average miss latency percentage difference comparing to cycle-accurate model.*

5.2.2.2 LLC Accuracy Evaluation

Unlike the CPI errors we see in Section 5.2.2.1, the LLC average miss latency errors are more observable. Mega2 and Mega4 still have relatively low errors of 3.0% and 7.1% on average. Mega8 and Mega16 sees an increasing 12% and 27% error. This is expected because the average LLC miss latency is already low, typically $20ns$ to $30ns$, and a few nanoseconds of delay per memory request is going to be a much more observable portion of the overall miss latency. For the very same reason, *ram_lat* benchmark has lowest average percentage error among the benchmarks because the memory access pattern of this benchmark is highly random, making it hard to prefetch and creating row misses in the DRAM backend. Therefore the absolute miss latency is already high enough that having a few nanoseconds of extra

delay translates to a smaller percentage error.

5.2.2.3 Memory Accuracy Evaluation

From the memory controller's perspective, the MegaTick model directly alters the request inter-arrival timings, and it works both ways: The incoming request may be delayed if it misses a synchronization cycle, therefore increasing the inter-arrival latency. The returning request may also be delayed for the same reason, and the next incoming request that depends on the returning request is therefore also delayed.

To better visually present the changes of the inter-arrival timings, we plot histogram density in Figure 5.8. We overlap the distribution of cycle-accurate results with MegaTick results in each graph for better comparison. It can be clearly seen that, from Mega2 to Mega16, the "shifting" of the distribution away from its cycle-accurate base becomes more and more clear. To quantitatively describe this "shifting", we calculate the intersection [62] value between the two histograms.

The intersection of two histograms can be understood as the scale of overlapping area of two histograms, and the histogram intersection of two histograms, H_1 and H_2, can be calculated as follows:

$$intersection(H_1, H_2) = \sum_i min(H_1(i), H_2(i))$$

We can obtain the normalized intersection (to H_1) by dividing the intersection value to $\sum_i H_1(i)$. This normalized intersection value ranges from 0 to 1, with 0

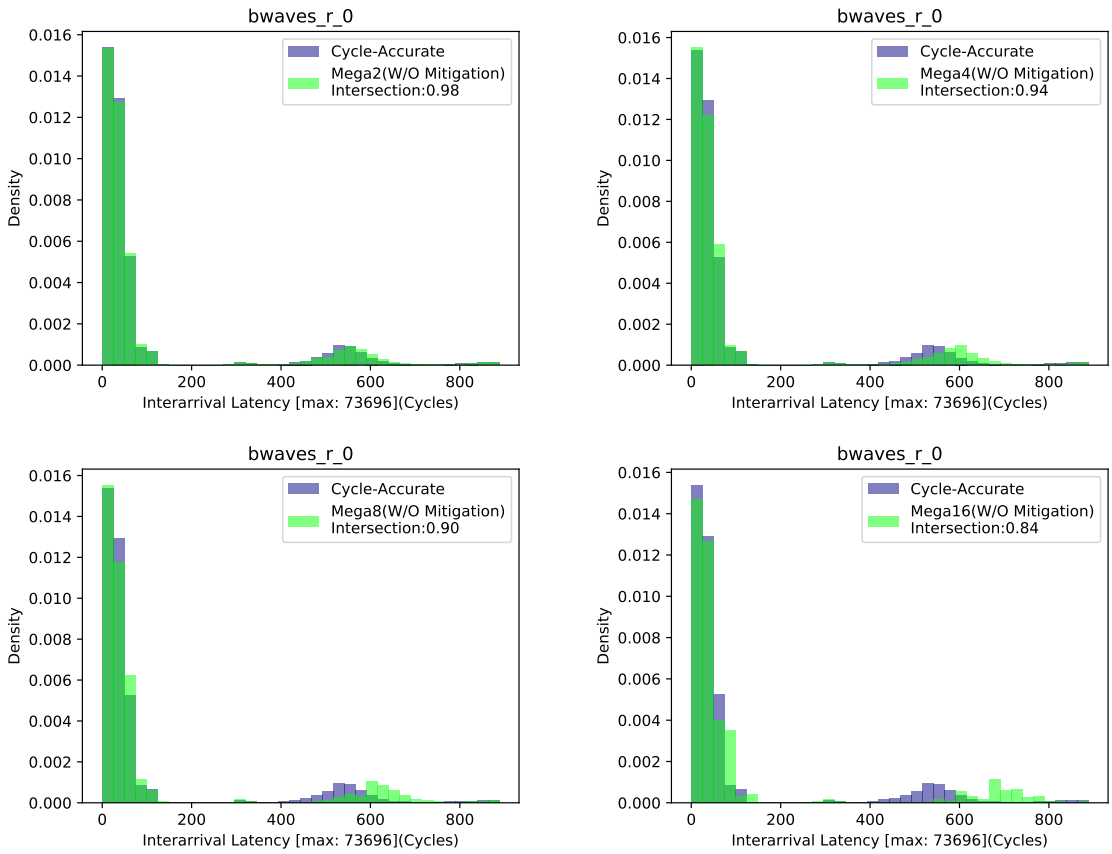


Figure 5.8: *Inter-arrival latency distributions density of bwaves_r benchmark. Mega2(top left), Mega4(top right), Mega8(bottom left), and Mega16(bottom right).*

meaning H_2 has no overlap with H_1 and 1 meaning H_2 fully overlaps with H_1. In the graphs we refer to this normalized intersection value as "intersection" for simplicity.

As can be seen in Figure 5.8, as we increase MegaTick value, the shifting of MegaTick distribution widens, and the intersection with cycle-accurate distribution decreases. We only show $bwaves_r$ benchmark in consideration of space.

5.2.2.4 Summary

As we demonstrated in this section, the MegaTick models show promising accuracy results, especially for Mega4 and Mega8 cases. However, the changes in LLC miss latency is not negligible. In the next section, we propose mitigation schemes that further significantly improve the overall CPI accuracy and LLC miss latency accuracy.

5.3 Accuracy Mitigation

The root cause of inaccuracy brought by MegaTick is due to increased memory latency of memory requests in between synchronization points(as shown in Figure 5.3). To be more specific, there are two aspects of this problem: memory requests are held by the processor longer to wait for the next synchronization cycle; and returning memory requests are held by the memory controller longer to wait for the next synchronization cycle. Some of these requests are closer to the previous synchronization cycle than the next synchronization cycle. So if we can issue or return these requests in the previous synchronization cycle instead of waiting for the next synchronization cycle, we can get a more accurate latency by balancing out the errors.

While from the memory controller's perspective, there is no way that it can predict when the next memory request will arrive, the memory controller does certainly know when the next memory request will return: when the memory controller issues a *READ* or *WRITE* DRAM command, it takes $tCL + t_{burst}$ cycles to finish

this request, which is typically more than a dozen DRAM cycles. Therefore, the memory controller knows ahead of the time when a memory request will complete in a future cycle, and at each synchronization cycle, the memory controller can simply look ahead in the returning queue and determine which request to return earlier.

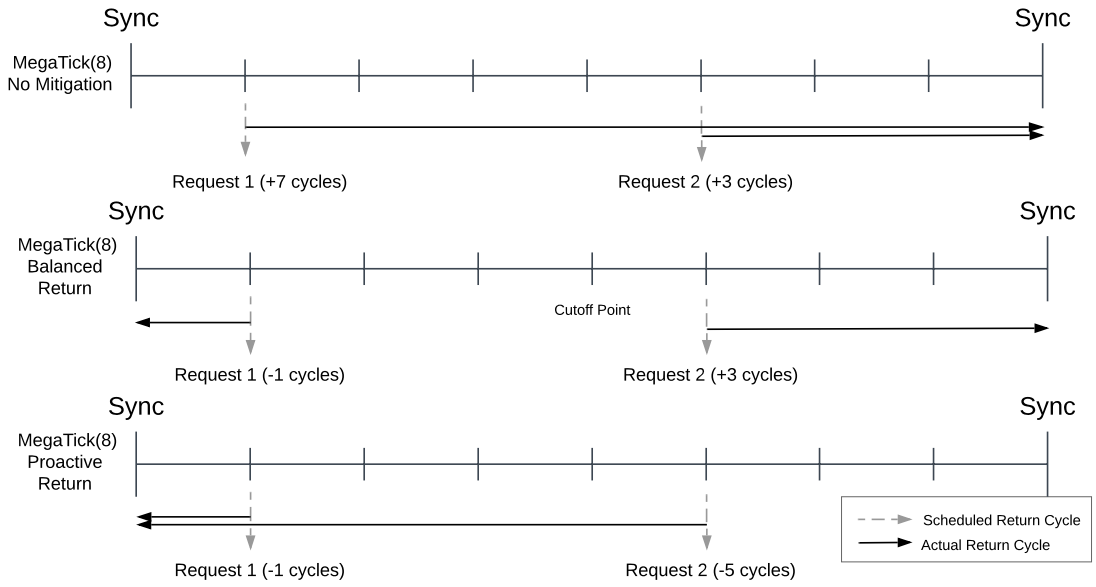


Figure 5.9: *MegaTick and its accuracy mitigation schemes. Balanced Return will return some requests before the next MegaTick (middle graph). Proactive Return will return all requests before the next MegaTick (lower graph).*

Based on this advantage, we propose two mitigation schemes: Balanced Return (BR) and Proactive Return (PR).

Balanced Return (Figure 5.9, middle) draws the line at the middle point between 2 MegaTicks, returns requests before the midpoint at the previous MegaTick and returns requests after the midpoint at the next MegaTick. This will create an average memory latency close to the cycle-accurate model.

Proactive Return (Figure 5.9, lower) returns all requests before the next MegaTick at current MegaTick. This will result in lower average memory latency comparing to cycle-accurate model, but it will also balance the effects of increased incoming request latencies, which the memory controller cannot control.

We implement these mitigation schemes into our simulation framework, and run the same experiments with mitigation. The results are shown as the following.

5.3.1 CPI Errors after Mitigation

First we measure the CPI of each configuration and plot the percentage difference compared to the corresponding cycle-accurate model. The results are shown in Figure 5.10 and Figure 5.11.

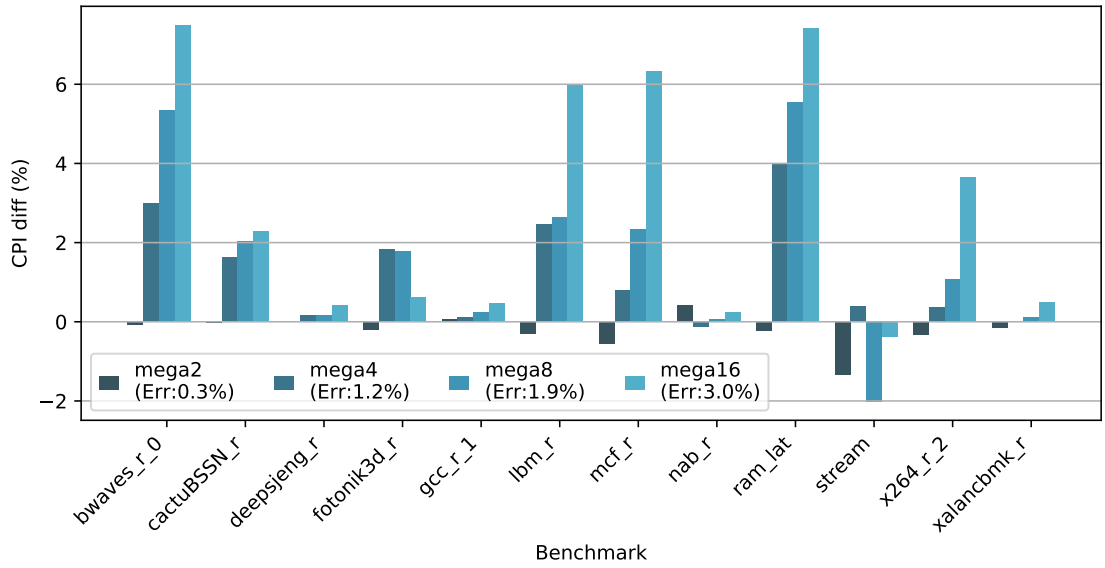


Figure 5.10: *CPI difference comparing to cycle-accurate model using Balanced Return mitigation. Absolute average CPI errors are shown in the legend.*

For the Balanced Return scheme, although the error improves a lot comparing

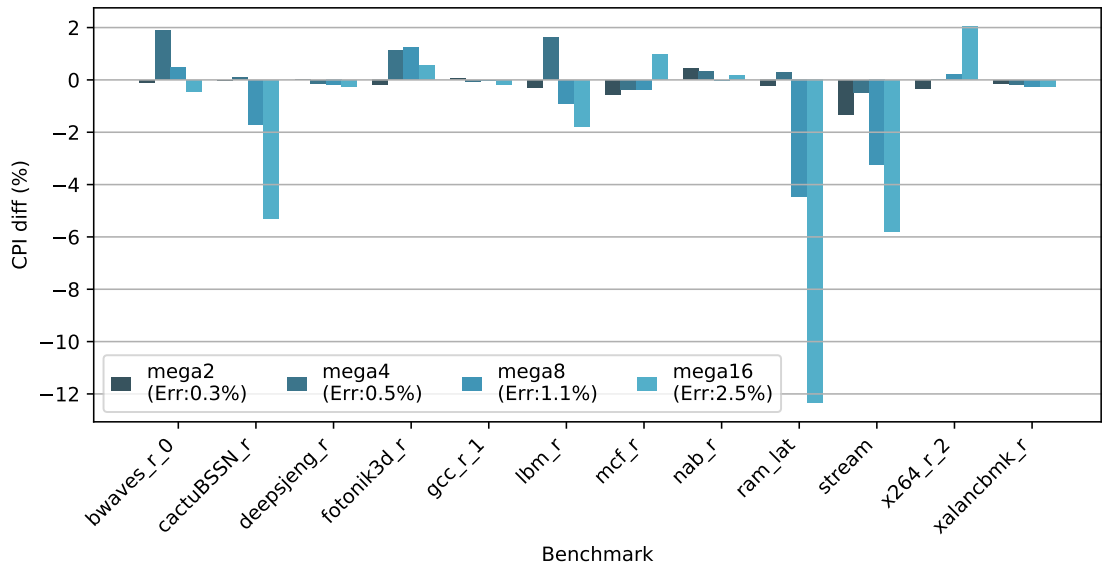


Figure 5.11: *CPI difference comparing to cycle-accurate model using Proactive Return mitigation. Absolute average CPI errors are shown in the legend.*

to MegaTick without mitigation, the majority of benchmarks still shows a higher CPI, indicating that compensating the returning memory latency is still not enough for the cycles lost in the incoming latency. The average CPI error is a mere 1.2% for Mega4, with a worst case of 4% error. Mega8 averages a 1.9% CPI error but worse cases reaches 5%. Mega16 further brings down the worst case error to 7% with an average of 3%.

For the Proactive Return scheme, the CPI accuracy further improves for almost all settings comparing to Balanced Return scheme. Mega4 averages 0.5% error without exceeding 2% error in any benchmark. Mega8 improves to 1.1% error with only 1 benchmark slightly over -4%. Mega16 sees the worst error here with -12% error in *ram_lat* benchmark, the most latency sensitive benchmark in the selection, but still manages to reduce the overall average error down to 2.5%.

5.3.2 LLC Errors after Mitigation

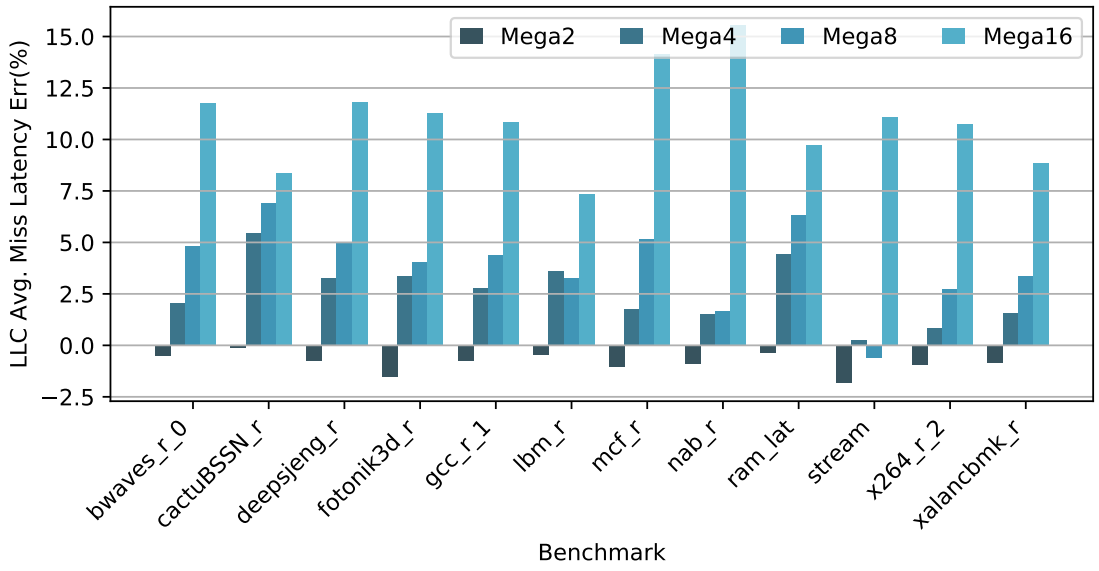


Figure 5.12: *Balanced Return model LLC average miss latency percentage difference comparing to cycle-accurate model.*

As for LLC average miss latency, we see a similar trend as with CPI accuracy: For Mega4 and Mega8 configurations, Balanced Return reduces the average LLC miss latency error from 7.1% and 12% to 2.6% and 4.0%, respectively. Worst case errors in Mega16 also significantly reduced by one third. The majority of configurations still show an increase in LLC miss latency because even though the returning requests are "balanced", the incoming request delay is still there.

Proactive Return further reduces the Mega4 and Mega8 errors to 0.85% and 0.95% respectively. For Mega16, even though the absolute error decreases to 7.5% from 11% of Balanced Return, we see a swinging distribution of the errors around zero, meaning that even with aggressively proactive returning, some cache misses

still take longer than before. This is due to the long synchronization window that causes the distribution of latencies to be compensated in one direction.

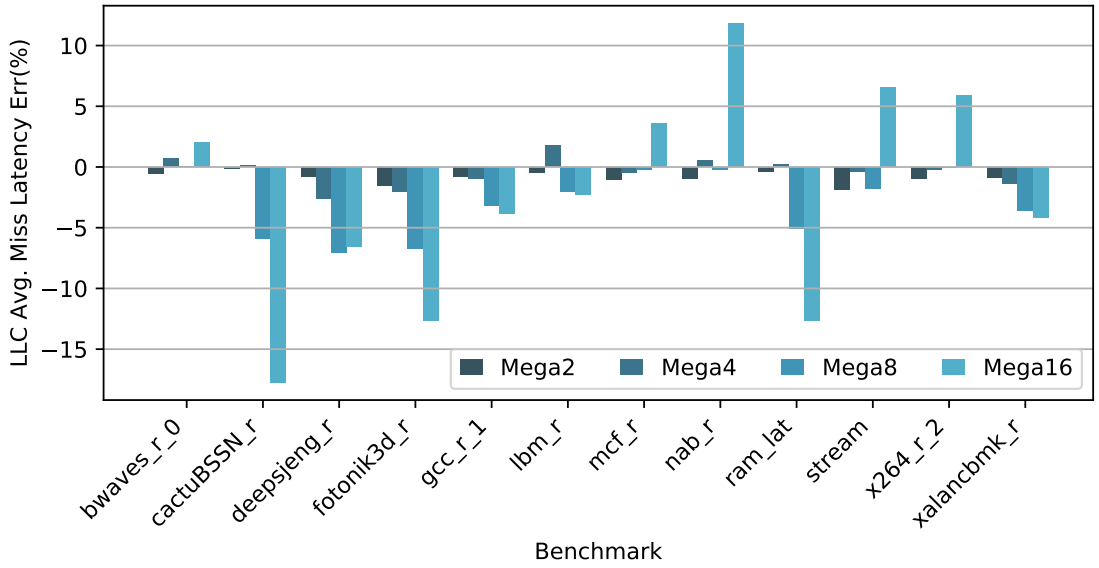


Figure 5.13: *Proactive Return model LLC average miss latency percentage difference comparing to cycle-accurate model.*

5.3.3 Memory Impact after Mitigation

Finally, on the memory controller side, the mitigation schemes effectively correct the "shifting" inter-arrival latency distribution. As the example shown in Figure 5.14, with Proactive Return mitigation, the intersection of Mega4 and Mega8 increases to 0.98 and 0.97 respectively, and Mega16 also jumps 0.07 to 0.87. While inter-arrival latency distributions cannot entirely represent the internals of a DRAM simulator, they demonstrate how the mitigation can minimize the effects of MegaTick.

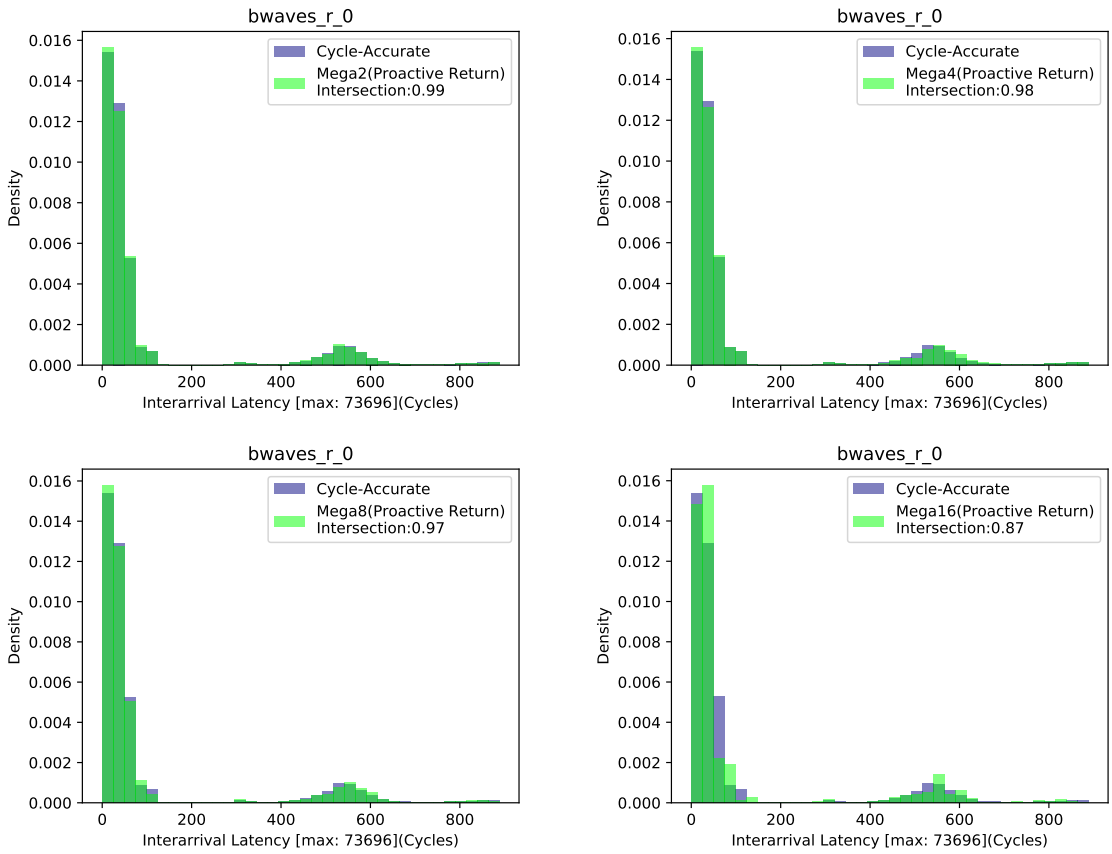


Figure 5.14: *Inter-arrival latency distributions density of bwaves_r benchmark with Proactive Return mitigation. Mega2(top left), Mega4(top right), Mega8(bottom left), and Mega16(bottom right).*

5.3.4 Summary

Comparing to MegaTick without mitigation, we are able to reduce the average CPI error from 2.7% to 0.5% for Mega4 model, and from 4.7% to 1.1% for Mega8 model with worst case errors also cut down by more than half. Similarly for LLC miss latency, Proactive Return mitigation is able to bring down the LLC miss latency error to low single digit percentage numbers. This proves the effectiveness of these mitigation schemes. Moreover, these mitigation schemes add no additional overhead

to the simulation time because, from the memory controller's point of view it only needs to change the timing threshold of returning requests from the current cycle to a future cycle, and no additional operations are needed.

5.4 Discussion

After the extensive testing, we can recommend some the accuracy versus performance tradeoffs when using the MegaTick technique. As we have already shown, the biggest performance jump is from Mega2 to Mega4, whereas the performance only gains marginally from Mega4 to Mega8. Combining with the accuracy results, We recommend Mega8/4 for best performance/accuracy results.

As for the number of threads used in simulations, generally matching numbers of threads to number of channels is good practice using MegaTicks. 2 channels or less is still better off using single thread.

5.5 Conclusion

In this chapter, we developed the first parallel DRAM simulator, and further proposed a lax synchronization scheme to reduce the parallelization overhead and to achieve practical speedup over single thread simulations. We also applied accuracy mitigation mechanisms to improve accuracy loss from the lax synchronization, making it a fast and accurate alternative to single thread DRAM simulators.

Chapter 6: Statistical DRAM Model

While we successfully implemented a practical parallel simulator that tackles multi-channel DRAM simulation scalability, it is still cycle-based, and thus the single channel performance and the interface compatibility issues are still unaddressed. These challenges require a fundamental change in simulation model, and hence we present our proposed statistical/machine learning DRAM simulation model.

6.1 Propositions

Different from analytic models that provide a high level analysis, which we discussed in Section 2.1.6, the statistical models here mean to provide an on-the-fly DRAM timing per request based on a "trained" statistical or machine learning model.

The foundation of why such a statistical model would work on DRAM is that:

- DRAM banks only have a finite number of states.
- The timing of each DRAM request has already been largely dictated by the DRAM states when it arrives at the controller.
- Our observation shows most DRAM request latencies fall into a very few la-

tency buckets, indicating that this behavior is likely the result of the previous two points.

And we will expand each of the claims one by one as follows.

DRAM banks only have a finite number of states: a DRAM bank can be modeled as a state machine: it can be in idle, open, refreshing, or low power states. Although there are typically thousands of rows that can be opened or closed, what matters to a specific request to a bank is whether the row of that request is open or not, so it will reduce to 2 states in this regard. Similarly, while there can be multiple banks in a rank and even multiple ranks in a channel, but for each request there is only a subset of these states that really matter to the timing of that request. Also, the queuing status when a new request arrives can also be accounted as states.

The timing of each DRAM request has already been largely dictated by the DRAM states when it arrives at the controller: intuitively speaking, when a request arrives at the DRAM controller, there are very limited actions that the controller can take. It can either A) process this request, whether because it is prioritized by the scheduler, or just because there are no other requests to be processed at the time, or B) hold the request whether because there is contention, other events are happening such as the current rank/bank is refreshing. Most of the scenarios here can be represented as a "state" like we previously discussed.

Our observation shows most DRAM request latencies fall into very few latency buckets, meaning that they are likely to be predictable: we plot the memory latency distribution of the 12 benchmarks we tested as Figure 6.1.

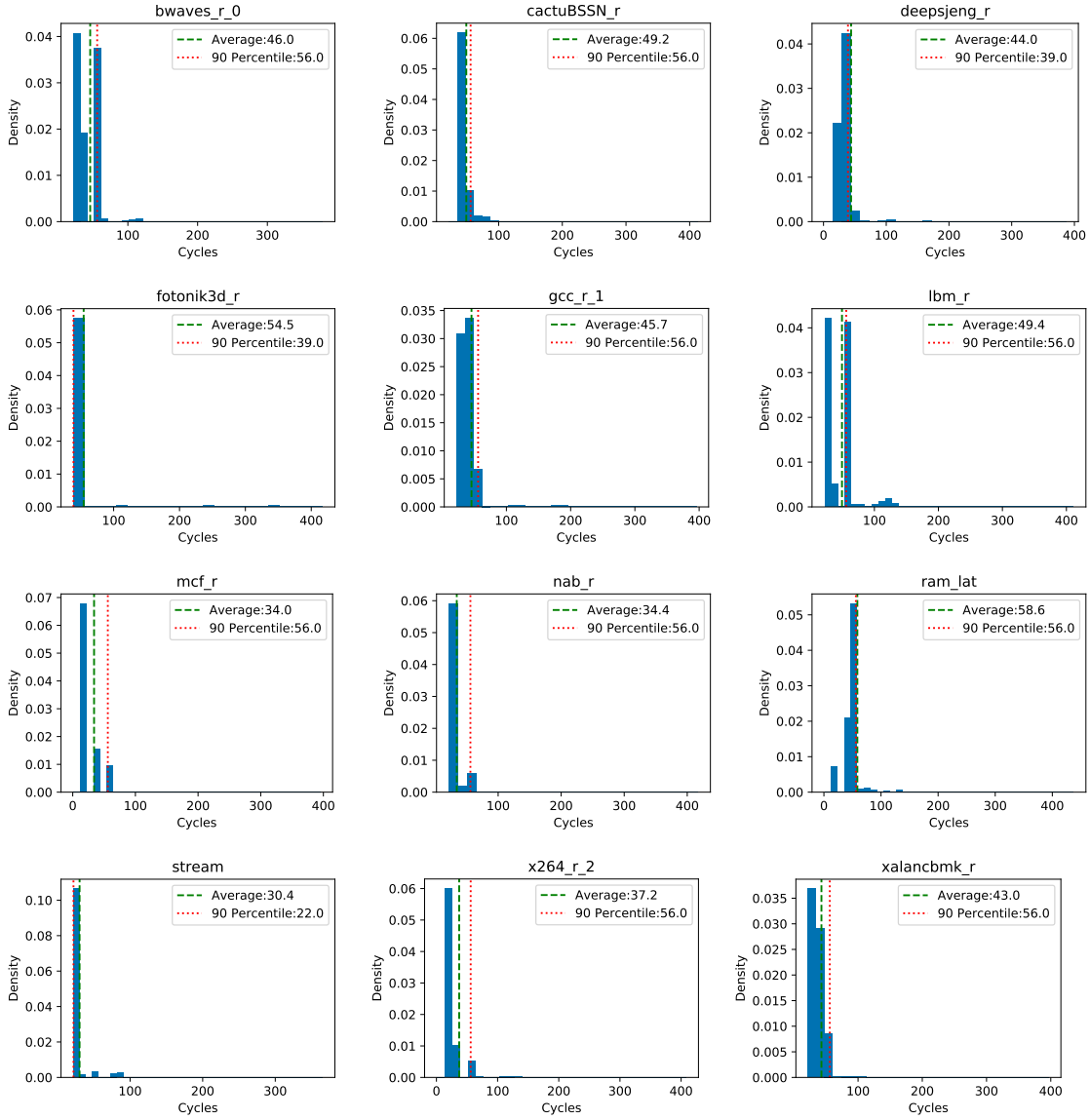


Figure 6.1: *Latency density histogram for each benchmark obtained by Gem5 O3 CPU and 1-channel DDR4 DRAM. X-axis of each graph is cut off at 99 percentile latency point, the average and 90-percentile point are marked in each graph for reference.*

Note we clip each histogram at the 99 percentile latency point for better visual. It can be seen that although every benchmark has a long tail latency that stretches to over 400 cycles (likely the result of having to wait for a refresh which is 420 cycles in this case), the 90-percentile line and the distribution itself both indicate that most of the memory latencies are limited to just quite a few latency buckets.

This distribution fits into a statistical or machine learning model very well: the majority of the cases are predictable while the corner cases are there to optimize. With a statistical or machine learning model, we cannot handle 100% of the requests accurately like cycle accurate simulator. However, if we can accurately predict, say 90% of the requests at the cost of a fraction of simulation time, then the trade-off may be worth the accuracy loss, especially for CPU and cache researchers who only need an "accurate enough" but preferably faster memory model.

6.2 Proposed Model

6.2.1 Classification

It is clear now that the latency distribution for most memory requests is concentrated in a very small range. But there can still be tens of numeric values in that small range. These numeric values create noises to prevent the model from converging. For example, some requests have the latency of 20 cycles, which is exactly the minimum cycles it takes to complete a row buffer hit. But requests of 21, 22, 23, and all the way to 30 cycles also represents row buffer hit conditions, because if it is not a row buffer hit, then the minimum latency will be $20 + tRCD$, which

is well over 30 cycles. All the variations of 20+ cycles are caused by reasons such as bus contention, or rank switching, but they are still essentially row buffer hits, and therefore they should all be classified as one category instead of 10 individual numbers.

As we stated in Section 6.1, the dominating factor of the latency of a memory request is the DRAM states. For instance, a row buffer hit results in 20 cycles latency; a request to an idled bank takes 35 cycles; a row buffer miss takes 50 cycles; a request blocked by refresh operations can take 400 cycles. These are far more influential than one or two cycles of bus contention. Plus, these smaller numbers are very specific to the DRAM protocol and are thus not portable/universal. Therefore we propose to classify requests into these collective categories as opposed to individual values.

Based on how DRAM works, we propose the following latency classes and their corresponding latency number in DRAM timing parameters:

- **idle**: this class of latency occurs when the memory request goes to an idle or precharged DRAM bank, requires an activation (ACT), and then read/write.
- **row-hit**: this class of latency occurs when the memory request happens to goes to a DRAM page that was left open by some previous memory requests.
- **row-miss** this class of latency occurs when the memory request goes to the a DRAM bank that has a different page opened by previous memory requests. Therefore, to complete this request, the controller must precharge the bank, then activate, and then read/write.

- **refresh** this class of latency occurs when the memory request is delayed by a refresh operation. Depending on whether the request comes before the refresh or during the refresh, the latency in this class may vary.

We do not seek to reproduce the exact latency as cycle accurate simulation, but extrapolate an appropriate latency number based on DRAM timing parameters. We will further explain this in Section 6.2.2.

6.2.2 Latency Recovery

Once we have latency classes in hand, combined with DRAM timing parameters, we can recover their latency into approximate DRAM cycles. By doing this we can avoid relying on any specific numbers but rather have a portable generic model. For example, we can simply plug in a DRAM profile with timing parameters to obtain latency numbers for that profile, and if we want latency numbers for a different profile, we simply plug in another DRAM profile without having to retrain the whole model.

We specify how we recover a latency cycle number from each latency class as follows:

- **idle**: the minimum latency of this class is a row access followed by a column access. In DRAM parameters it is typically $tRCD + tCL + BL/2$. Note there are some variances. For instance, read or write operations may have different $tRCD, tCL$ values, and for GDDR the burst cycles can be $BL/4$ as well.

- **row-hit**: the minimum latency of this class is simply the time of a column

access. In DRAM parameters it is typically $tCMD + tCL + BL/2$. Again, there are protocol specific variances like the **idle** class.

- **row-miss**: the minimum latency of this class is a full row cycle. In DRAM parameters it is typically $tRP + tRCD + tCL + BL/2$.

- **refresh**: We use a refresh counter similar to the refresh counters in DRAM controller, to provide timestamps of when each rank should refresh. We only use the timestamps as references to determine whether a request arrives right before a refresh or during a refresh. If the request comes right before the refresh, then we estimate the latency as $tRFC + tRCD + tCL + BL/2$. If the request arrives during the refresh cycle, e.g. n cycles after the reference refresh clock ($n < tRFC$), then we estimate the latency as $tRFC - n + tRCD + tCL + BL/2$. For example, the refresh counter marks cycle $7200, 14400, .etc$ as refresh cycles for rank 0, and if a request arrives at cycle 7201, then it will be regarded as arriving during a refresh. Now by no means our reference refresh timestamps can matches precisely the real refresh cycle in a cycle accurate simulation, but it is a good approximation for the impact of refresh.

6.2.3 Dynamic Feature Extraction

To train a statistical or machine learning model, we need "features" that provides distinctive information about the latency classes.

As we know, the latency of each memory request is largely dependent on the DRAM states, which are the results of previous memory requests. For example, a

previous request opened a page when the DRAM bank was idle, then a following request that goes to the same bank same row can take that advantage. So the features we are looking for here should come from the address streams, especially the previous requests, and we need to be able to extract features dynamically from these address streams.

There are two aspects of extracting features from address streams, temporal features and spatial features. Temporal features reflect the potential dependency between requests. For example, a previous request that is 5 cycles ahead should have more impact on the timing of the current request than another previous request 5000 cycles ahead. The difficulty is how to translate the timing intervals into useful features. Again we cannot rely on specific values because there would be too many features to be useful. But instead, we use generic DRAM parameters to classify how far or near is a previous request. For example, we consider a request "near" if it was arrived within tRC cycles, the intuition is that in tRC cycles, which represents the full row cycle, the DRAM can be activated and then precharged by a request, which renders that DRAM state unchanged to a following request outside of tRC cycles. Another line we draw here is the "far" line, which uses the number of $tRFC$, which is the number of cycles it takes to do a refresh. It may imply a reset state for the DRAM.

Spatial features need to reflect the structures of DRAM, in particular, banks and rows, because the state of each bank is the most determining factor for the incoming DRAM request. For example, if we are trying to predict the timing of one request, the previous requests that go to the same bank weigh more than the

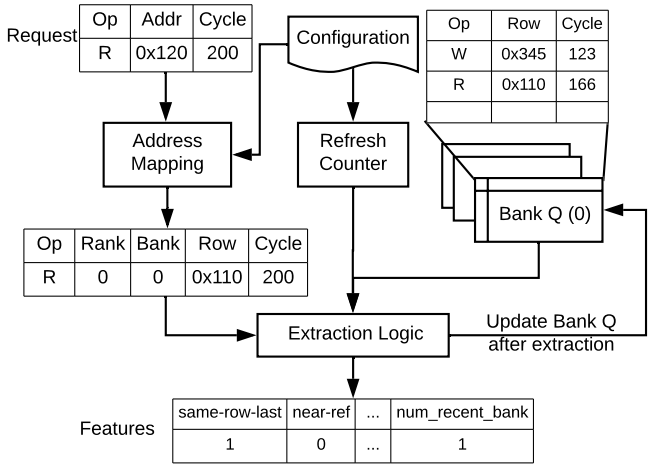


Figure 6.2: *Feature extraction diagram. We use one request as an example to show how the features are extracted.*

previous requests go to any other banks. And same as temporal features, we don't need to identify each bank and row by their specific bank number, but instead we identify them by "same row", "same bank", "same rank"(but different bank), or "different rank". We can evaluate a request with previous requests on these fields easily once they have their physical addresses translated to DRAM addresses(rank, bank, row, column). And to simplify and facilitate feature extraction, we maintain a request queue for each bank and put requests into each bank queue after the address translation. Unlike the queues in DRAM controllers, this bank queue is not actively managed and is strictly FIFO with a maximum length imposed for performance optimizations.

Combining the temporal features with spatial features, we can have features coded with both temporal feature and spatial feature. For instance, $num-recent-bank$ feature counts the number of previous requests that go to the same bank and

that are recent. We propose a list of features in Table 6.1; these features can give hints on the possible state that the DRAM banks are in and how DRAM controllers can make scheduling decisions etc.

The feature extraction using one request as an example is shown in Figure 6.2.

6.2.4 Model Training

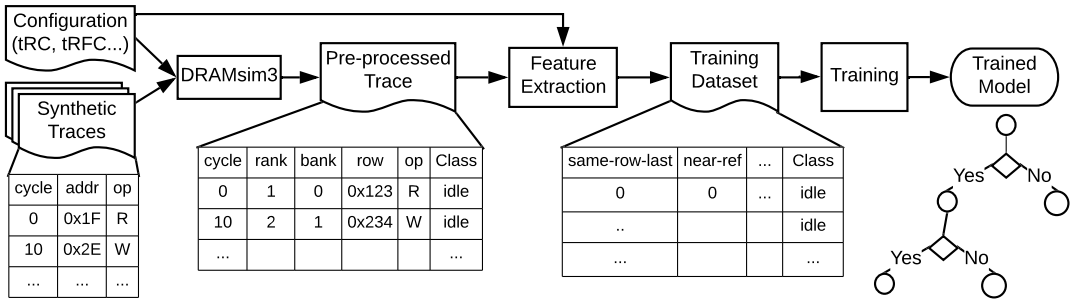


Figure 6.3: *Model Training Flow Diagram*

Having the features and classes ready, we now put pieces together and build the training flow shown in Figure 6.3. We use synthetic traces as training data, and use a cycle accurate DRAM simulator, DRAMsim3, to provide ground truth. The beauty of using synthetic traces is that we can use a small amount of synthetic traces to represent a wide range of real world workloads. For example, we can control the inter-arrival timings of the synthetic traces to reflect to intensity of workloads; we can also generate contiguous access streams and random access streams and interleave them to cover all types of memory access patterns of real workloads. Plus, we also don't have to worry about the contamination of testing dataset when we test the model with real workloads.

Table 6.1: *Features with Descriptions*

Feature	**Values**	**Description**	**Intuition**
same-row-last	0/1	whether the last request that goes to same bank has the same row (as this one)	key factor for the most recent bank state
is-last-recent	0/1	whether the last request to the same bank added recently (tRC)	relevancy of the last request to the same bank
is-last-far	0/1	whether the last request to the same bank added long ago (tRFC)	relevancy of the last request to the same bank
op	0/1	operation(read/write)	for potential R/W scheduling
last-op	0/1	operation of last request to the same bank	for potential R/W scheduling
ref-after-last	0/1	whether there is a refresh since last request to the same bank	refresh reset the bank to idle
near-ref	0/1	whether this cycle is near a refresh cycle	latency can be really high if it's near a refresh
same-row-prev	int	number of previous requests with same row to the same bank	if there is same row request then OOO may be possible
num-recent-bank	int	number of requests added recently to the same bank	contention/queuing in the bank
num-recent-rank	int	number of recent requests added recently to the same rank	contention
num-recent-all	int	number of recent requests added recently to all ranks	contention

We run the synthetic traces through DRAMsim3 with a DRAM configuration file as usual. To mark the ground truth, we modified DRAMsim3 so that it generates a trace output that can be used for training. Because the DRAMsim3 knows exactly what happens to each request inside its controller, it can precisely classify the requests into any of the categories we proposed in Section 6.2.1. And once the requests are classified, we run them through the feature extraction to obtain features. Finally, we run the features along with the classes into a model to obtain a model.

There are many machine learning models that can potentially handle this particular classification problem and we are not going to test every one of them as it is out of the scope of this thesis. In this study we start with simple and efficient models like decision tree [63] and random forest [64] for this study:

- These models are simple, intuitive, and explainable, which is quite important for prototyping work like this: it helps to be able to look at the model to examine and debug.
- The an ensemble tree model mimics how a DRAM controller works naturally, but instead of doing it in a series of cycles, the decision tree makes the decision instantly.
- The simplicity of these models makes both training and inference fast, the later is crucial for simulations performance.

As far as hyperparameter tuning goes, while decision tree model is relatively simple, there are still many hyper-parameters to tune. Luckily the model is not

hard to train, and it does not take much time to go through many parameters. We tried several different approaches(including brute force), and they all work decently. But what we have found that produced best accuracy is Stratified K-fold Cross Validation [65]. Stratification samples help reduce the imbalance of classes in our training dataset, especially the **refresh** class that is much rarer than other classes. K-fold Cross Validation divides the training data sets into k folds, and for each fold, uses it as test set and the rest $k-1$ folds as training sets.. This further reduces the bias and overfitting of the model. The details of hyperparameter training can be found later in Section 6.3.1.

6.2.5 Inference

Inference is relatively straightforward, as shown in Figure 6.4. However, one thing to note is that, if we are to compare the inference results to the results of cycle accurate simulation, we have to use the same DRAM configuration profile as the cycle accurate simulation. Otherwise we are not required to use the same DRAM configuration profiles.

In implementation, the entire inference process only takes one function call combining the request cycle, address, and operation(read/write), and the inference function returns the number of cycles that the request is going to take to complete.

This is great relief from the cycle accurate interface where the frontend has to always stay synchronized with the DRAM model. It allows much more flexible integration into other models.

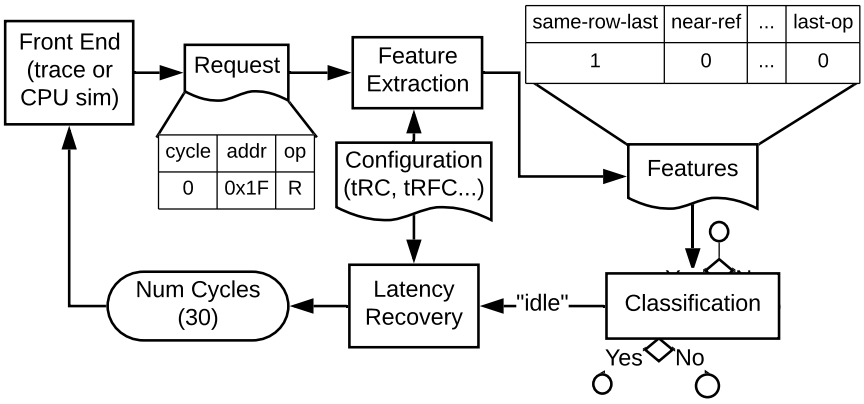


Figure 6.4: *Model Inference Flow Diagram*

6.2.6 Other Potential Models

The innovation of this work is to translate what is an essentially time-series problem into a non-time-series problems. We are aware of that there are models that work on time-series problem. Some of the temporal features in the data are easy to extract, whereas if we use models to automatically extract features, it will be costly when it comes to training. Additionally, our approach preserves portability and model reusability when it comes to different DRAM profiles, which we believe is not easy to preserve in other models. That being said, we certainly look forward to other efficient implementations of this problem.

6.3 Experiments & Evaluation

6.3.1 Hyperparameters

We use the *Scikit-learn* [66] package to train our model, which contains a set of tools and models that are readily available. The hyperparameters we use for training the decision tree model is shown in Table 6.2. We use the $StratifiedShuffleSplit$ module to conduct a grid search on the these hyperparameters to find the best fitting model.

Table 6.2: *Hyperparameters of the decision tree model.*

Hyperparameter	Values	Explanation
max-depth	None, 3, 5, 8, 10	Max depth of any path in the decision tree ("None" means unlimited)
min-samples-leaf	5, 20, 20, 30, 0.1, 0.2	Min number of samples needed to create a leaf node in the tree. Float number means ratio.
min-samples-split	5, 10, 20, 0.05, 0.1	Min number of samples to create a split in the tree. Float number means ratio.
min-weight-fraction-leaf	0, 0.05, 0.1	Min weighted fraction of the sum total of weights required to be at a leaf node.
max-features	auto, 0.2, 0.5, 0.8	Max number of features to consider. Auto means square root of number of features.
random state	1, 3, 5	Help train reproducible model.

On top of these hyperparameters, we also use $k - fold = 5$ for K-fold cross validation. The end result is that there are a total of $5400 \times 5 = 27000$ models to

train. Fortunately, each model trains quickly and the training can be distributed to multiple cores/threads in parallel. It takes less than a minute to train and evaluate all 27000 models using a 4-core desktop CPU.

The best hyperparameters will be automatically selected among all models trained based on accuracy. The values of the "best" hyperparameters are listed in Table 6.3. The best accuracy is **96.76%** (for all cross-validation data).

Table 6.3: *Hyperparameter Values of Best Accuracy*

Hyperparameter	Value
max-depth	None
min-samples-leaf	20
min-samples-split	5
min-weight-fraction-leaf	0
max-features	0.8
random state	3

As for the hyperparameters of random forest model, the default parameters provided by Scikit-learn package works out of the box, trains in seconds, and produces an accuracy as good as the decision tree model. Therefore we did not explore the hyperparameters of random forest model.

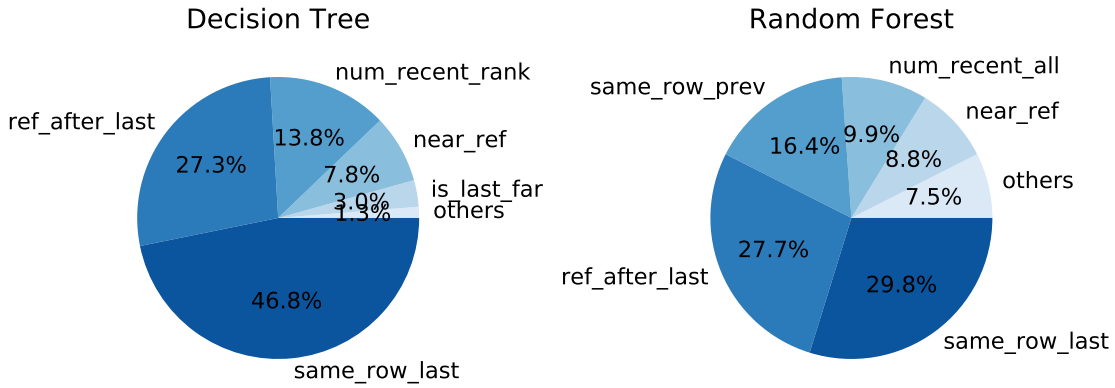


Figure 6.5: *Feature importance in percentage for decision tree and random forest*

6.3.2 Feature Importance

We analyze the trained models and see how they treat the features differently. To better visualize the importance of features, we clip the least important features into one category("others"), and plot the pie chart as Figure 6.5.

The two models handles different features differently, with the two most important features the same: $same-row-last$ and $ref-afterr-last$, contributing to more than 50% combined. It can also be seen that the distribution of importance is more balanced in the random forest model than the decision tree model.

6.3.3 Accuracy

We now apply the trained model on real-world benchmarks. The benchmarks are the same as what we used in Chapter 5. We run all the benchmarks with cycle

accurate, out-of-order Gem5 CPU along with DRAMsim3, and this will provide us the golden standard for our accuracy tests. A address trace for each of the benchmark is generated as the input to the statistical models, this allows the statistical model and cycle accurate model to have exactly the same inputs to work with. For each request we also record its latency class and latency value in cycles labeled by DRAMsim3 so that we can use it for comparison with the statistical models.

Note that there are two aspects of accuracy, classification accuracy, which represents how many requests the statistical model can correctly classify according to the cycle accurate models; and latency accuracy, which is the numeric latency values of the request produced by the statistical models.

First we look at **classification accuracy** for decision tree and random forest models, as shown in Figure 6.6 and Figure 6.7.

As can be seen in Figure 6.6 and Figure 6.7, overall the predictors produces great classification accuracy across all benchmarks for both models. The average classification accuracy is 97.9% for decision tree and 98.0% for random forest. In most cases the classification accuracy even exceeds our training accuracy. This is because our training traces generally contain more address patterns than most real-world workloads, and is thus harder to work with. Also note that the accuracy between random forest and decision tree models is almost indistinguishable, the largest difference being a mere 0.4% for *lbm* benchmark, in all other cases the difference is often 0.1% to 0% difference. This shows signs of our model converging.

Being able to correctly classify the latency categories is the important first step. The next step is to verify that our latency recovery model can also reproduce

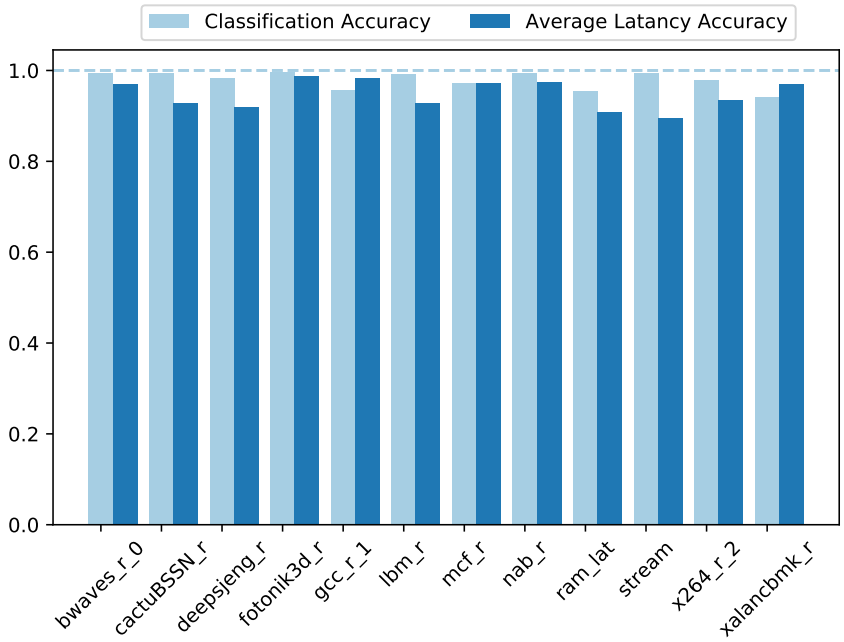


Figure 6.6: *Classification accuracy and average latency accuracy for decision tree model on various benchmarks.*

the latency value in cycles according to the DRAM configuration profile. Our model translate the latency class for each memory request to a numeric value in DRAM cycles, and we compare these numeric latency values against our cycle accurate model baseline. To put into perspective of the classification accuracy, we again use the cycle accurate numbers as the baseline and plot the average latency value from our statistical models normalized to the cycle accurate model as the accuracy measurement. We call this measurement "*Average Latency Accuracy*". The average latency accuracy are plotted side by side with classification accuracy in both Figure 6.6 and Figure 6.7. The average relative latency for both decision tree model and random forest model are **0.969** comparing to cycle accurate base with worst case 0.94 for *stream* and *ram_lat* benchmarks. This is rather impressive, considering that our

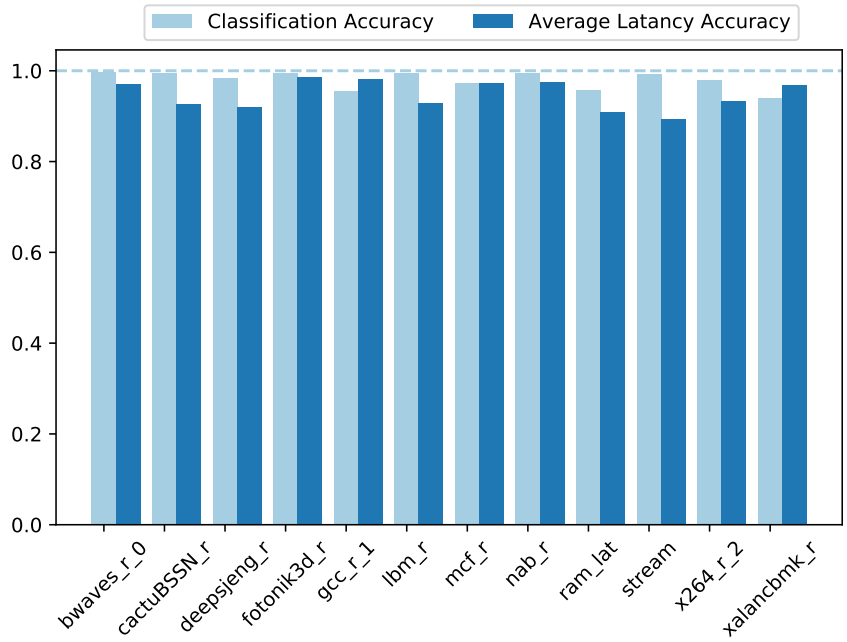


Figure 6.7: *Classification accuracy and average latency accuracy for random forest model on various benchmarks.*

cycle based MegaTick model even has a higher error than this. We also have a discussion on how to further reduce the latency accuracy in Section 6.3.5.

6.3.4 Performance

We now compare the simulation performance. The simulation time of cycle accurate is measured as usual. The simulation time of inference models is measured from end to end, which starts from the time of parsing the trace to all the predictions are done. We use the inverse of simulation time as simulation speed, and plot the simulation speed normalized to the cycle accurate simulations, as shown in Figure 6.8.

It can be seen that our prediction model runs 2.2x to 250x faster than the cycle

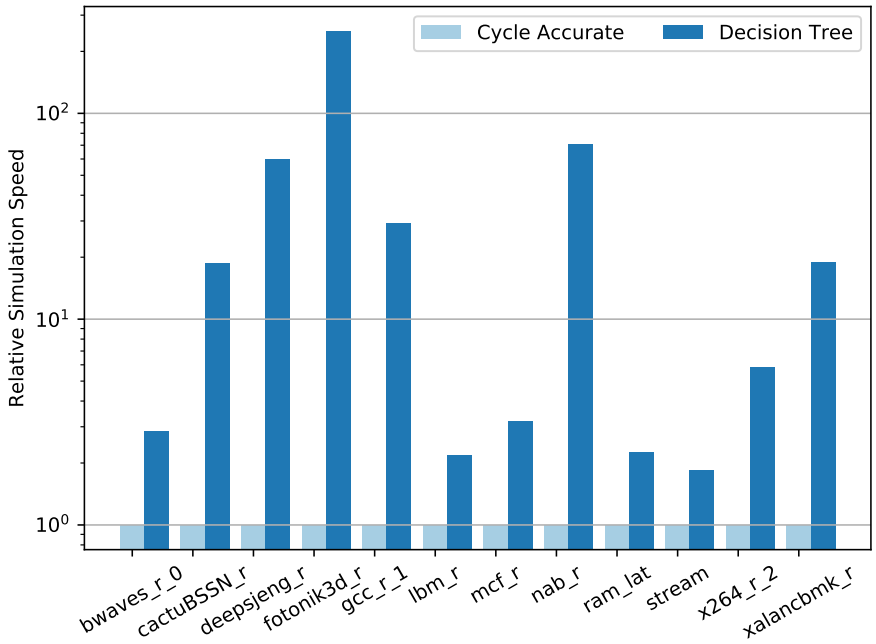


Figure 6.8: *Simulation speed relative to cycle accurate model, y-axis is log scale.*

accurate model. The simulation speed of an inference model is solely dependent on the number of requests, because the work to predict each request is the same. In contrast, there are many more factors for cycle accurate simulations: first off, each cycle has to be simulated even if there is no memory request at all; the memory address patterns, which alter the behavior of scheduler, also impact the simulation performance.

To demonstrate linearity of inference models, we sort the benchmarks by the number of memory requests they generate, and plot the simulation time over the the number of requests as in Figure 6.9. Both random forest and decision tree model exhibit almost strictly linear performance. The random forest model is slightly slower than decision tree, due to its more complex inference structures, but it is

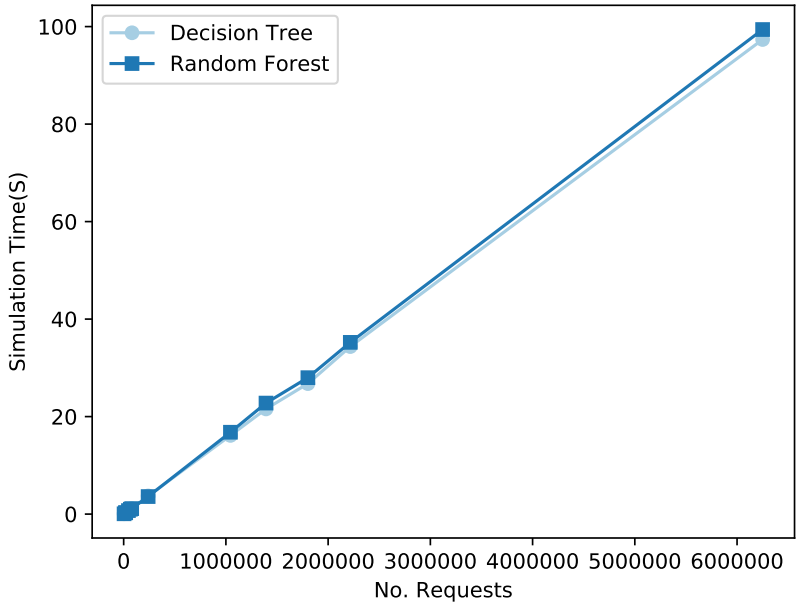


Figure 6.9: *Simulation speed vs number of memory requests per simulation.*

linear as well. We can conclude that the time complexity of our model is $O(n)$ where n is the number of requests, and hence $O(1)$ for each request.

Note that our implementation of the inference flow is far from perfect, especially the feature extraction which is coded in a plain *python* script, takes about 90% to 95% of the overall inference flow. With an efficient *C/C++* implementation we should be able to see another order of magnitude speedup.

6.3.5 Multi-core Workloads

Previous experiments show our proposed model can successfully model DRAM timings for single core workloads, no matter the memory activity intensity. The success is based on the premise that the high accuracy classification can translate to

Table 6.4: *Randomly mixed multi-workloads.*

Mix	Benchmarks
0	*stream, xalancbmk_r, lbm_r, bwaves_r*
1	*bwaves_r, mcf_r, lbm_r, fotonik3d_r*
2	*deepsjeng_r, bwaves_r, gcc_r, fotonik3d_r*
3	*xalancbmk_r, x264_r, bwaves_r, gcc_r*
4	*mcf_r, stream, ram_lat, lbm_r*

high accuracy latency prediction because the variances are low in each class. While this might be true for single core workloads, multi-core workloads may break the assumption.

To validate how well our model holds against scaling workloads, we amplify the workload by randomly mixing 4 traces of different workloads together to reflect intensive multi-core memory activities, and we use the same methodology to evaluate the accuracy. The mix of benchmarks is shown in Table 6.4.

It can be seen, in Figure 6.10, that our model still demonstrate very high classification accuracy with 0.99 for each mix, but the average latency sees a decrease down to 0.88 in the worst case (mix 4). The accuracy disparity between classification accuracy and latency accuracy is due to the long gap between the latency category edges. For instance, in the DDR4 configuration we tested, the $row - hit$ class is 22 cycles while the next near class, *idle*, is 39 cycles, leaving 17 cycles in between.

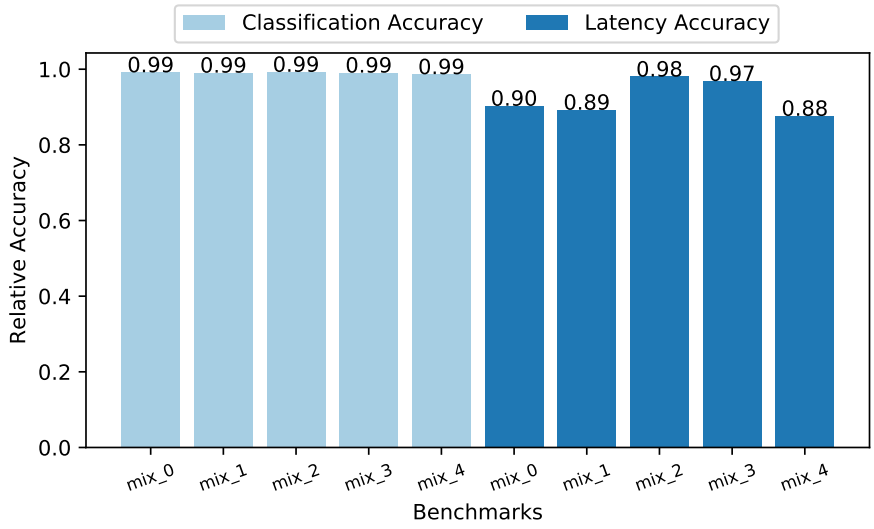


Figure 6.10: *Classification accuracy and average latency accuracy for randomly mixed multi-workloads.*

To further quantify this effect, we breakdown each latency class to those whose actual (cycle accurate simulated) latency matches exactly with their predicted latency; and those whose actual latency is more than their predicted latency, which we name as "$Class+$". For instance, the in the DDR4 configuration, $row-hit$ class translate to 22 cycles, while $row-hit+$ class represents those requests that are "row hit" situations but with more than 23 cycles due to contention. Figure 6.11 shows the breakdown of such classes for each mix. Each bar in the graph represents the percentage of the total requests for each class. Note that the predicted latency of $refresh$ classes is a variation itself so it does not accompany a "+" class like others. It can be seen that for mixes that have higher latency accuracy such as $Mix2$ and $Mix3$, the percentage of the "+" classes are much smaller, typically less than 10 percent combined. The opposite can be observed from other mixes such as 0, 1

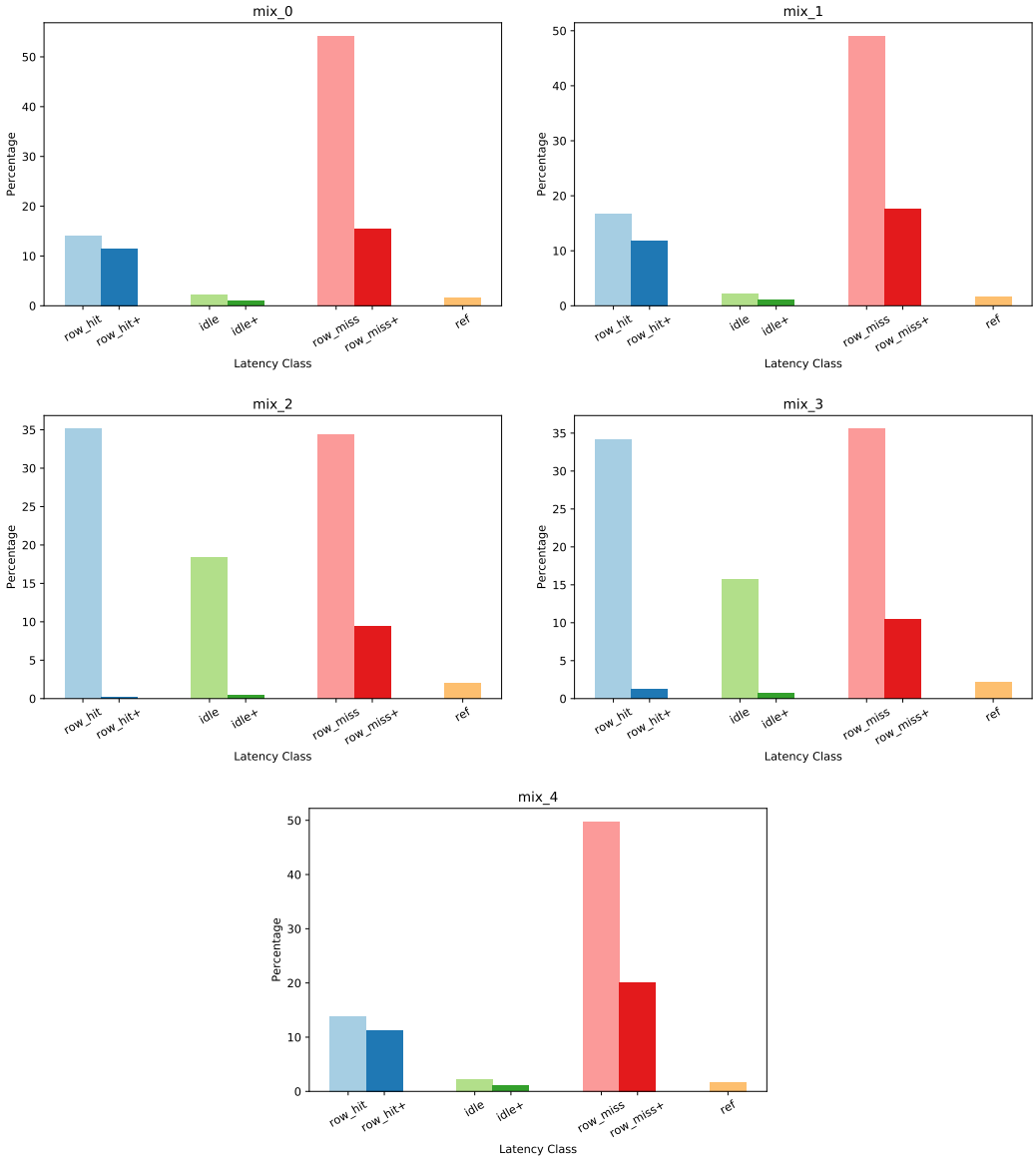


Figure 6.11: *Request percentage breakdown of latency classes and their associated contention classes for randomly mixed multi-workloads. "+" classes are the contention classes apart from their base classes.*

and 4, where the "+" classes contribute to more than 20% of the total requests, resulting the inaccuracy in their latencies. Further looking into the specific benchmarks in each mix, we can confirm that the mixes with higher percentage of "+" all consist of more than 2 memory intensive benchmarks, whereas the mixes with lower percentage of "+" have at most 1 memory intensive benchmark.

One way to combat the extra latencies introduced by contention is to train the model with more latency classes, i.e., filling the latency gap between current classes with more latency classes. This may increase the training efforts but should reduce the latency discrepancy between our statistical model and cycle accurate model.

6.4 Discussion

6.4.1 Implications of Using Fewer Features

In the early stage of prototyping the machine learning model, we did not obtain results as good as Section 6.3.3. However, these results are still valuable in providing insights to the future improvement of the model. Therefore, we document the early prototype and results in this discussion.

One early prototype we had did not have the FIFO queue structure, but instead only keeping the latest previous memory request to the same bank, i.e. effectively a $depth = 1$ queue. This only allows us to extract features such as *same-row-last, is-last-recent, is-last-far, op,* and *last-op*. We only trained decision tree for for evaluation, and the classification accuracy and average latency accuracy for each of the benchmarks we tested are shown in Figure 6.12.

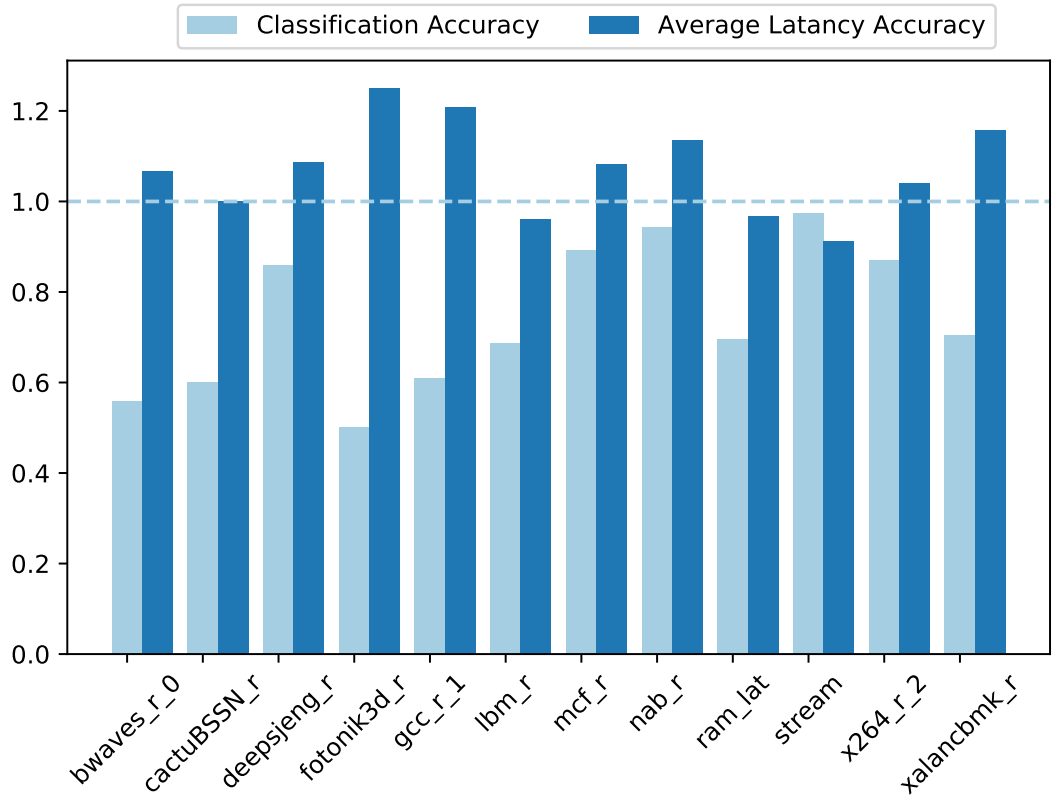


Figure 6.12: *Classification accuracy vs average latency accuracy of an early prototype of a decision tree model.*

As can be seen in Figure 6.12, the classification accuracy ranges from 0.5 to 0.94, with an average of 0.74; perhaps surprisingly, the average latency accuracy is better on numbers: ranging from 0.91 to 1.25, with an average accuracy of 1.07, or an absolute 10% error. In some benchmarks, classification accuracy can be 40 to 50 percent off while latency difference is much smaller. The reason behind this is that, with only the last request to the same bank being recorded, the model tends to predict more requests as *row-hit* or *row-miss* than it should, whereas in reality, a lot of these requests should be *idle*. Coincidentally, with the DDR4 DRAM parameters, the average latency of *row-hit, 22 cycles*, and *row-miss, 56 cycles*, is *39 cycles*, which

is the *idle* latency. Therefore, while lots are latency classes are mis-predicted, the average latency numbers are not too far off. This presents a good reason that we should examine both classification accuracy and latency accuracy instead of focusing solely on one measurement.

The lack of tracking for previous requests beyond one entry, and no account for refresh operations are the primary reasons for low classification accuracy. Tracking for previous requests beyond one entry allows the scheduler to make out-of-order scheduling decisions. Another wildcard that we did not anticipate is the role of *refresh*. Although there are typically only less than 3% of memory requests are directly blocked by DRAM refresh operations, the subsequent impact of refresh is larger: each DRAM refresh operation resets the bank(s) to idle state, which leads to the next round of requests to these banks to have idle latency. When there are not many requests issued to the refresh-impacted banks in between two refreshes, the refresh operation will render a much larger impact.

6.4.2 Interface & Integration

Traditionally, the cycle accurate DRAM **simulator interface** is "asynchronous", meaning that the request and response are separated in time: the CPU simulator sends a request to the DRAM simulator without knowing at which cycle the response comes back; while waiting for the memory request to finish, the CPU simulators has to work on something else every cycle; finally, when the DRAM simulator finishes the request, it calls back the CPU simulator, who processes this

memory request and its associated instructions. This asynchronous interface only works in cycle accurate simulator designs, as the CPU simulator has to check in with the DRAM simulator every cycle to get the correct timing of each memory request.

The statistical model, however, brings an "atomic" interface to the simulator design, meaning that upon the arrival of each request, the timing of this request can be provided back to the CPU simulator immediately with high fidelity. This will enable much easier integration into other models than cycle accurate models. For example, when integrated into an event-based simulator, the response memory event can be immediately scheduled to the future cycle provided by the statistical model, and no future event rearranging is needed.

Furthermore, the atomic interface provided by the statistical model will benefit parallel simulation framework. Because in a parallel simulation framework, simulated components interacting with each other generate synchronization events across the simulation framework, and frequent synchronization will negatively impact the simulation performance. The statistical model only needs to be accessed when needed, thus reducing the synchronization need to a minimum.

6.5 Conclusion & Future Work

In this chapter, we discussed the limitation of cycle accurate DRAM models and explore alternative modeling techniques. We proposed and implemented a novel machine learning based DRAM latency model. The model achieves highest accuracy among non-cycle-accurate models, and performs much faster than a cycle accurate

model, making it a competitive offering for cycle accurate model replacement.

The model still has room to improve as future works. First off, currently the model is implemented in *Python*, and if the entire flow can be implemented in *C++*, we can expect much more performance gain without any impact on classification accuracy. Secondly, introducing more latency classes can bridge the gap between latency accuracy and classification accuracy for memory intensive workloads. Or rather, more latency classes can be constructed to model the working mechanisms of more sophisticated controller/scheduler designs beyond our currently modeled out-of-order open-page scheduler, providing more flexibility to the model. Finally, we only trained and tested decision tree and random forest models for the purpose of prototyping, and we realize that there are lots of alternative machine learning models that could also work for this problem, so it may be worth exploring other models in the future.

Chapter 7: Memory System for High Performance Computing Systems

In this chapter, we introduce the background and challenges of the memory system design in high performance computing systems, present our proposed interconnect topology and routing algorithms, and describe our experiments and results.

7.1 Introduction

On large scale memory system such as the memory system in a High Performance Computing (HPC) system, computational efficiency is the fundamental barrier, and it is dominated by the cost of moving data from one point to another, not by the cost of executing floating-point operations [67–69]. Data movement has been the identified problem for many years and still dominates the performance of real applications in supercomputer environments today [70]. In a recent talk, Jack Dongarra showed the extent of the problem: his slide, reproduced in Figure 7.1, shows the vast difference, observed in actual systems (the top 20 of the Top 500 List), between peak FLOPS, the achieved FLOPS on Linpack (HPL), and the achieved FLOPS on Conjugate Gradients (HPCG), which has an all-to-all communication pattern within it. While systems routinely achieve 90% of peak performance

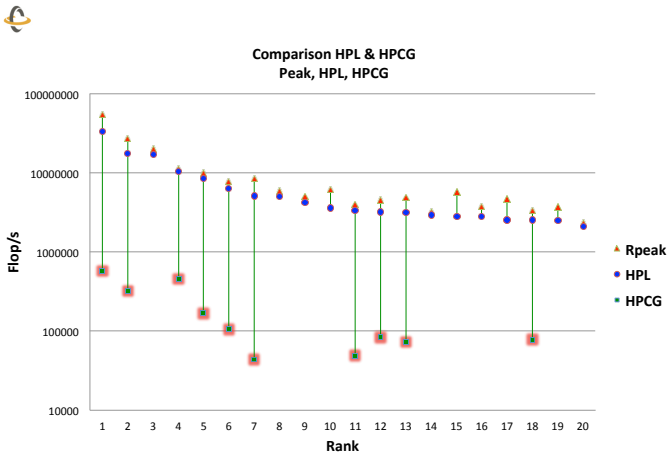


Figure 7.1: *A comparison of max theoretical performance, and real scores on Linpack (HPL) and Conjugate Gradients (HPCG). Source: Jack Dongarra*

on Linpack, they rarely achieve more than a few percent of peak performance on HPCG: as soon as data needs to be moved, system performance suffers by orders of magnitude. Considering when systems are moving towards exascale, which is roughly 10× the scale of most powerful system today, this problem will be more and more critical.

To ensure efficient system design at large scale system sizes, it is critical that the system interconnect provide good all-to-all communication: this means high bisection bandwidth and short inter-node latencies. We propose a new set of high bisection bandwidth, low latency topologies, namely, Fishnet interconnect, to alleviate this issue.

To accurately evaluate proposed topology, simulations comparing against existing topologies are needed. Therefore, we implemented Fishnet into Booksim [71], a cycle-accurate network simulator. Using Booksim, we are able to obtain preliminary results for various topologies under synthetic traffic workloads. To further

improve the accuracy and verify the results, we implemented Fishnet topologies in Structural Simulation Toolkit (SST) [13], which allows us to simulate the system with more detailed hardware models and realistic workloads. We use SST to conduct large scale simulations up to more than 100,000 nodes for both existing and proposed topologies, and taking into account of the other factors that can be critical to system performance such as routing and flow control, interface technology, and physical link properties (latency, bandwidth).

With the help of the large volume of data obtained from simulation, we are able to gain a comprehensive view of these topologies and it is therefore helpful for assessing their usage in Exascale. To conclude, we summarize our contributions in this section as follows:

- We perform large scale, fine grained network simulations and design space explorations. We simulate the network size with up to 100,000 nodes and collect more than 3,000 data points. To our knowledge, it is by far the largest design space exploration at such large scale.

- We propose and evaluate adaptive routing algorithms tailored for Fishnet and Fishnet-lite topologies. Our evaluation shows properly implemented routing can reduce the execution slowdown by 20x under heavy adversarial workloads.

- We show how the different network parameters influence the performance of each topology under different workloads. We observe that most topologies benefit from an increasing link bandwidth while they are less sensitive to longer link latency.

- We evaluate the interconnect topologies for their scaling efficiency and their capability to handle increasing workloads, which provides useful insights on the interconnect system design for Exascale.

Our findings show that highly efficient network topologies exist for tomorrow's supercomputer systems. For modest port costs, one can scale to extreme node counts, maintain high bisection bandwidths, and still retain low network diameters.

7.2 Background and Related Work

To achieve low latency high bandwidth exascale network, traditional low radix routers with wide ports are replaced by high radix routers with large amount of narrow ports in the system design to reduce the average hop count. Most recent interconnect network topology researches are based on high radix routers. Kim, J. et al. [72] introduced a dragonfly network with unity global diameter with increased effective radix virtual router which was built by a group of high radix routers. Cray Cascade system [73] was also based on dragonfly topology with 48-port router and four independent dual Xeon socket nodes per Cascade blade. The system size was range from 3,072 to 24,576 nodes. Mubarak, M. et al. [74] performed simulations of million-node dragonfly networks based on Rensselaer Optimistic Simulation System (ROSS) framework which is a discrete-event base simulator on IBM Blue Gene/P and Blue Gene/Q. HyperX [75] is a n-dimension network with each dimension being fully connected. It's an extension of the hypercube topology, n-dimension torus with 2 nodes in each dimension. A 4,096 nodes with 32-port router HyperX network was

simulated and compared against the fattree topology. [76] used Moore graphs to construct diameter-2 interconnect topologies. We will introduce some of the most representative topologies in detail in Section 7.2.1. Among these works, we did a comprehensive comparison between traditional network topologies, and also a novel topology, fishnet, which will be explained in the next section.

As for the methodology of studying large scale interconnect, it has always been challenging to study large scale systems. On the one hand, simulating a system with even thousands of nodes is difficult (let alone millions), because it can require significant resources and time. Therefore, carefully engineered tools are required to perform such simulations in an efficient way. Jiang et al. developed *Booksim*, a cycle accurate interconnection network simulator [71], but its single-threaded design makes it very hard to simulate large scale network. Carothers et al. designed and implemented the Rensselaer Optimistic Simulation System (ROSS) simulator [77] based on time warp technique and is highly efficient on parallel execution. Rodrigues et al. developed Structural Simulation Toolkit (SST) [13], a parallel discrete event simulation toolkit that aims to help design and evaluate supercomputer systems. Its modular design makes it easier to extend their models.

On the other hand, there are many factors that could influence the performance and cost of large scale interconnection networks and a variety of studies exist on characterizing such interconnection networks. Mubarak et al. has studied and simulated high dimension torus networks with up to 1 million nodes and showed that large scale simulations are critical for designing Exascale systems [78]. [72–74, 79] studied different aspects of Dragonfly networks and characterized Dragonfly as a

highly scalable and efficient interconnect. [80, 81] evaluated diameter-2 topologies with various routing algorithms and traffic patterns. Li et al. introduced more scalable Fishnet and Fishnet-lite diameter-4 interconnect topology and performed a preliminary performance/cost analysis [82].

7.2.1 Existing Interconnect Topologies

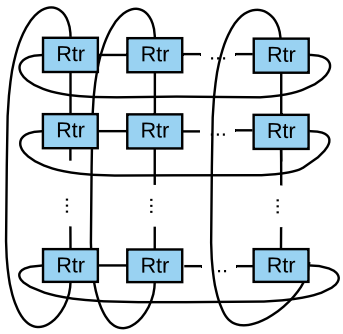


Figure 7.2: *Torus*

Torus (Figure 7.2) is organized as an n-dimensional grid with k nodes per dimension (k-ary n-cube) [83]. The simplicity of a torus makes it easier to implement and study compared to other topologies. Although low-dimension torus networks are not considered scalable due to a network diameter that increases significantly with the system size, as well as a relatively low bisection bandwidth, high-dimension torus networks have diameters that scale much more slowly and have shown interesting characteristics such as high bisection bandwidth [78]. Researchers have also built petascale supercomputers using a variant of a 6D torus (Tofu [84]), proving it to be a promising candidate for exascale.

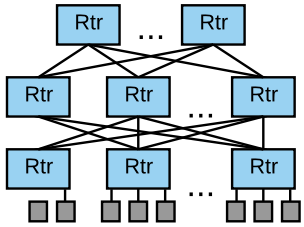


Figure 7.3: *3-level Fattree*

Fat-Tree (Figure 7.3), also referred as a folded Clos topology [85,86], enables low latency and high bisection bandwidth using high-radix routers. The architecture of Fat-tree provides very high bisection bandwidth and is used in real world supercomputer systems [87]. The disadvantage of the fat-tree topology is its scalability: for example, a 2-level Fat-tree would require 100 ports per router to scale to 50,000 nodes. The port cost and its associated power cost would be prohibitively expensive for building an Exascale network within 2 levels. Increasing the number of levels of a Fat-tree will alleviate its scalability issues but would also introduce more end-to-end latencies. Given the network scale that we target (100k node), we focus on 3- and 4-level Fat-tree topologies, which could scale to 100k node within 100 ports per router.

Dragonfly (Figure 7.4) [88] was proposed to address the scalability issues of previous topologies such as Fat-tree networks and flattened butterfly topologies [89]. The concept of a *virtual router* was introduced here as a means to reduce the port cost and achieve scalability. A virtual router is essentially a hierarchical design that groups routers and treats them as one, thereby reducing the number of

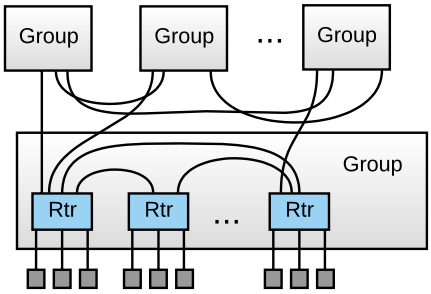


Figure 7.4: *Dragonfly*

expensive global links. There are different ways of setting up a Dragonfly topology so that one can obtain high efficiency (by maximizing system size with minimal number of ports per router) or high performance (by adding more global links per router). In Figure 7.6 we show how different setups of Dragonfly can result in very different scalability ("Dragonfly (max)" in 7.6 refers to the high efficiency setup while "Dragonfly (min)" refers to high performance setup). Because our goal is to explore extremely large scale networks, we follow the efficiency setup recommended by [88], that is, $a = 2p = 2h$, where a is number of routers per group, p is number of nodes per router, and h is number of global links per router. In this way we are able to scale up to 100,000 nodes with only 51 ports per router ($p = 13$). This allows us to scale up to 100k nodes with only 51 ports per router.

Slimfly: Bao et al. first proposed using Hoffman-Singleton graph (a 50-node diameter-2 graph) as an interconnection topology for high density servers [90]. Besta et al further proposed Slimfly topology to construct much larger scale networks based on Moore graphs [76] (An example of 10-node graph is shown in Figure 7.5). The low network diameter provides potentially lower latency and the Moore graph

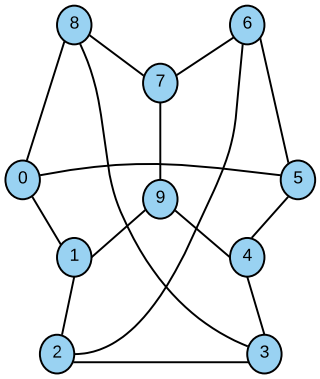


Figure 7.5: *A diameter-2 graph* ($n = 10, k' = 3$)

architecture provides high path diversity. An example of 10 node, diameter-2 Moore graph is shown in 7.5. In Slimfly, assuming the number of links of a nodes that connects to other nodes in the network is k' and number of endpoints attached to a node is p, then the number of endpoints in a system will be approximately $k'^2 \times p$. To avoid oversubscribing a router, a setup of $p \leq k'/2$ is recommended by [76]. Following this setup, we are able to construct a network with $k' = 79$ and $p = 18$ to get a 101,124 node network within 100 ports per router. We do not want to oversubscribe the router for it would not be a fair comparison against other topologies.

Routing There are already a variety of established routing algorithms for the topologies described in this study. For example, [91] proposed and compared deterministic routing and adaptive routing for Fat-tree. Their evaluation show both deterministic and adaptive routing are effective in reducing packet latencies. Minimal, VAL, and UGAL are successfully used in Dragonfly and diameter-2 topologies [76, 80, 88]. We will implement these routing algorithms and evaluate them

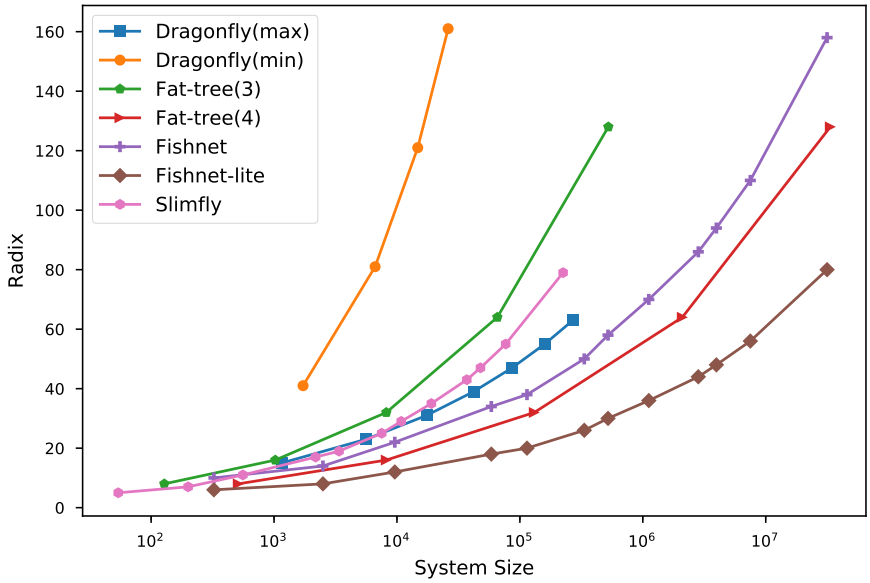


Figure 7.6: *Scalability of different topologies studied in this work*

by simulation to get a comprehensive understanding of the effectiveness of routing algorithms.

7.3 Fishnet and Fishnet-Lite Topologies

In this section, we present our proposed interconnect topology, Fishnet. We demonstrate how to construct a Fishnet topology, and discuss the routing algorithms tailored for Fishnet topologies.

7.3.1 Topology Construction

The Fishnet interconnection methodology is a novel means to connect multiple copies of a given subnetwork [92], for instance a 2-hop Moore graph or 2-hop Flattened Butterfly network. Each subnet is connected by multiple links, the origi-

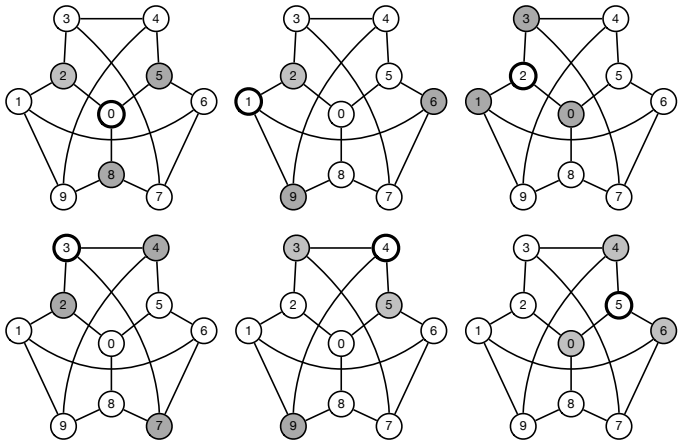


Figure 7.7: *Each node, via its set of nearest neighbors, defines a unique subset of nodes that lies at a maximum of 1 hop from all other nodes in the graph. In other words, it only takes 1 hop from anywhere in the graph to reach one of the nodes in the subset. Nearest-neighbor subsets are shown in a Petersen graph for six of the graph's nodes.*

nating nodes in each subnet chosen so as to lie at a maximum distance of 1 from all other nodes in the subnet. For instance, in a Moore graph, each node defines such a subset: its nearest neighbors by definition lie at a distance of 1 from all other nodes in the graph, and they lie at a distance of 2 from each other. Figure 7.7 illustrates.

Using nearest-neighbor subsets to connect the members of different subnetworks to each other produces a system-wide diameter of 4, given diameter-2 subnets: to reach remote subnetwork i, one must first reach one of the nearest neighbors of node i within the local subnetwork. By definition, this takes at most one hop. Another hop reaches the remote network, where it is at most two hops to reach the desired node. The "Fishnet Lite" variant uses a single link to connect each subnet, as in a typical Dragonfly, and has maximum five hops between any two nodes, as

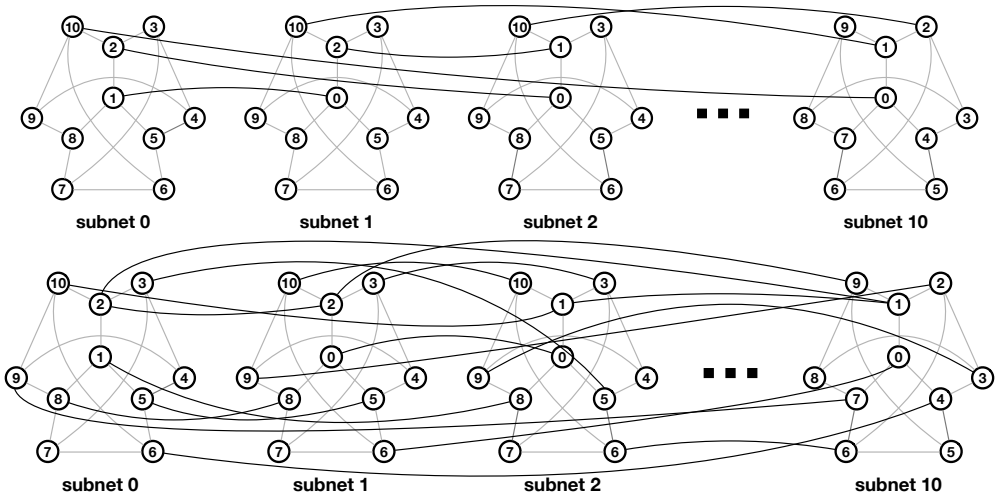


Figure 7.8: *Angelfish (bottom) and Angelfish Lite (top) networks based on a Petersen graph.*

opposed to four.

An example topology using the Petersen graph is illustrated in Figure 7.8: given a 2-hop subnet of n nodes, each node having p ports (in this case each subnet has 10 nodes, and each node has 3 ports), one can construct a system of $n + 1$ subnets, in two ways: the first uses $p + 1$ ports per node and has a maximum latency of five hops within the system; the second uses $2p$ ports per node and has a maximum latency of four hops.

7.3.2 Routing Algorithm

Routing algorithms play an important role in fully exploring the potentials of an interconnect topology. Previous studies have shown that applying proper routing algorithms could result in significant latency and throughput improvements [72, 80, 91] on various topologies. In this section, we will explore routing algorithms

for Fishnet topologies (we use Angelfish as an example) as well as review options for traditional topologies.

Fishnet and Fishnet-lite interconnects were briefly studied in [82] but only minimal routing was discussed in their study. It is necessary to further study more efficient routing schemes for such topologies to fully explore their potentials. Especially for Fishnet-lite, where only one global link is used to connect between subnets, using minimal routing could congest the global link easily and thus leads to performance degradation.

To address this problem, we propose Valiant random routing and adaptive routing algorithms tailored for the architecture of Fishnet and Fishnet-Lite.

7.3.3 Valiant Random Routing Algorithm (VAL)

The Valiant Random Routing algorithm [93] is used in multiple interconnect topologies to alleviate adversarial traffics [76, 88]. The idea of Valiant routing is to randomly select a intermediate router (other than the source and destination router) and route the packet through 2 shortest paths between the source to intermediate and between the intermediate to destination, respectively. By doing so, additional end-to-end distance is added into the path, but it may also avoid a congested link and balance the load on more links, and lower the overall latency.

Applying Valiant routing to Fishnet family will be similar to Dragonfly topology, where global links between groups/subnets are more likely to be congested when the traffic pattern requires more communication between groups/subnets. In

Dragonfly, a random intermediate group is used to reroute the packet to the target group.

Similarly, for Fishnet-lite, we randomly select a intermediate subnet and route the packet to the intermediate subnet and then to its destination subnet. This could increase the worst case hop count from 5 to 8, but would also increase the path diversity, with $k' - 1$ more paths, and reduce the minimal route link load to $1/k'$ of its previous value.

For Fishnet, we apply a similar technique, which will result in a hop count from 4 to 6 in worst case, but expand the path diversity from k' to k'^2.

7.3.4 Adaptive Routing

The idea of adaptive routing is to make routing decisions based on route information. One of the widely used adaptive routing schemes, Universal Globally-Adaptive Load-balanced Algorithm (UGAL), [94] takes VAL generated routes and compares them with the minimal route, selecting the one with less congestion. The key here is to decide which route has less congestion. Ideally, if we have global information of all routes and all routers, it would be easy to make such decisions. However, in real systems, it is impractical to have such information across the system. Therefore, a more reasonable approach is to only use local information, (UGAL-Local, or UGAL-L) such as examining the depth/usage of local output buffers.

UGAL-L works well on topologies such as Dragonfly and Slimfly. However, its effectiveness will be limited in Fishnet since the local information obtained from

output buffer cannot reflect route congestion accurately when the next link is congested and the information is not propagated back. This also happens in Dragonfly networks, as described in [95].

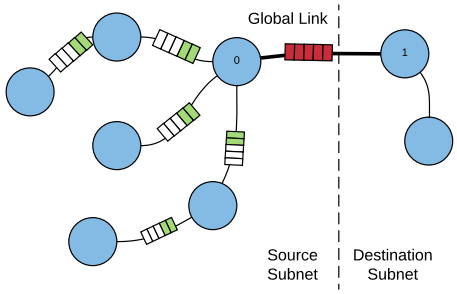


Figure 7.9: *Example of how inappropriate adaptive routing in Fishnet-lite will cause congestion. Green tiles means low buffer usage while red tiles means high buffer usage.*

An example of how traditional UGAL-L might not work well for Fishnet-lite is shown in Figure 7.9. Imagine the worst case scenario where all k' nodes in the source subnet want to send packets to the destination subnet. Because in Fishnet-lite there is only one global link connecting between the source and destination subnet, all the minimal routes will pass through that router (router 0 in fig. 7.9). If all the output buffers towards that router have very low usage, traditional adaptive routing will prefer the minimal path over Valiant path. This would keep happening until the intermediate buffers are almost full, and by then there will be a lot of packets in the buffers waiting for the global link to be available, jamming routers on both side of the global link.

To avoid this situation, we have tailored adaptive routing for the Fishnet family in the following way: When the router connects to the destination subnet

is not greater than 1 hop away, we adapt the minimal path, otherwise use a VAL path. By doing this, we effectively enforce the path diversity between subnets from 1 to k and reduce the number of packets to be routed minimally to the congested link from k'^2 to k' in the worst case traffic pattern. Moreover, because all other $k^2 - k$ packets are routed randomly to other $k - 1$ intermediate subnets through $k-1$ global links, those global links will route k packets per link as well. Therefore, all the global links will have equal workloads in worst case traffic pattern.

For Fishnet, there is another place where adaptive routing decision can be made. Since there are k' links between any two subnets, minimal routing would arbitrarily route to one of them. For adaptive routing, we can examine the output buffer usage to those k' routers that offer global links to the destination subnet and choose the one with the lowest buffer usage. Because there are at most 2 hops in this process, the back propagation problem discussed earlier will be less severe here.

We refer these routing algorithms as "adaptive routing" for the rest of the thesis and we will evaluate the effectiveness of these routing algorithms along with other comprehensive evaluations in the later parts of this chapter.

7.3.5 Deadlock Avoidance

In this study, we will adapt and implement the virtual channel method proposed in [96] for each topology. Since previous studies have illustrated how to implement such methods, we will not repeat the details here.

7.4 Experiment Setup

In this section we describe how our simulation is set up. We introduce the simulator used in this study, SST, and the network parameters and workloads chosen for this study.

7.4.1 Simulation Environments

As we have stated, it is inherently challenging to simulate a network at very large scale: given the enormous number of nodes in the system to simulate, it would require a huge amount of memory, and the simulations may take very long to finish if the simulator is not properly designed. Additionally, if we would also simulate a variety of network parameters and workloads, meaning that there are more simulations to perform.

We conduct a two-stage simulation: A) the first stage only model the router in detail and use synthetic traffic to model workloads, this simplified model is fast and thus allows us to get a quick but still reliable estimation of the topologies we studied. B) The second stage uses a more detailed model on not only the router, but also the compute nodes, physical links, software stack, and workloads. This detailed simulation is more time consuming but can give us more accurate results and allows us to simulate the topologies with a more parameters.

A summary of our detailed simulation configurations and workloads can be found in Table 7.1, and we will describe these parameters in more details.

We use SST as our simulator for this study. SST is a discrete event simulator

Table 7.1: *Simulated Configurations and Workloads*

System Size *	50,000 and 100,000
Topology	Dragonfly, Slimfly, Fat-tree (3 to 4 levels), Fishnet, Fishnet-Lite
Routing Algorithm †	Minimal, Valiant, Adaptive(UGAL)
Link Latency	10ns, 20ns, 50ns, 100ns, 200ns
Link Bandwidth	8GB/s, 16GB/s, 32GB/s, 48GB/s, 64GB/s
MPI Workloads	AllPingPong, AllReduce, Halo, Random
Total # configurations	> 3000

*Note that the system size here is approximate since most topologies cannot be configured to be these exact numbers.

† Not all topologies supports all of these routing algorithms. e.g. We only simulated deterministic and adaptive routing for Fat-tree.

developed by Sandia National Lab, designed for modeling and simulating DOE supercomputer systems [13,97]. To support massively parallel simulation, SST is built on top of MPI and is able to partition simulated objects across multiple MPI ranks; this can significantly speed up simulations. SST has a modular design that separate router models and end-point models. SST's *Merlin* high-radix router model has built-in support for torus, Fat-tree, and Dragonfly topologies, and it also provides a set of MPI workloads (by its *Ember* endpoint model). Additionally, SST also has built in configurable NIC model and middleware model such as *firefly* and *hermes* which provide the ability to simulate low-level protocols and message passing interfaces.

We extended SST's *Merlin* router model to support Slimfly, Fishnet and Fishnet-lite topologies along with their routing algorithms, and open sourced the code (link is not provided here for reviewing purposes). An overview of our simulated system can be found in Figure 7.10.

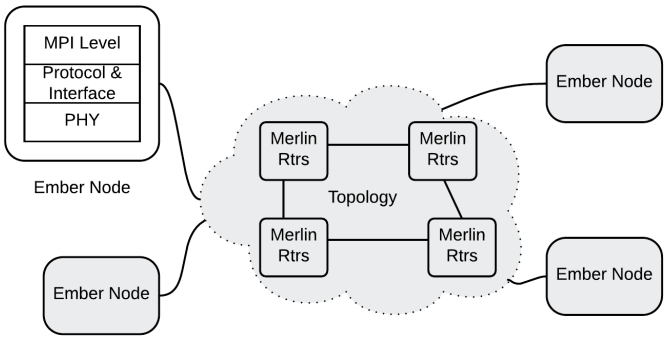


Figure 7.10: *Overview of Simulation Setup*

Each simulation is configured to run 10 iterations, and simulated execution time data is collected as the metric for performance.

7.4.2 Network parameters

In this study we primarily focus on link bandwidth and link latency as network parameter variables. Not only do these parameters have a significant influence on system performance, they also represent one of the more significant physical costs of the system.

To focus the range of the simulation parameters appropriately, we referenced some real world systems to get a perspective. Sunway TaihuLight reports the communication between nodes via MPI has a bandwidth of 12GB/s and a latency of

$1\mu s$ [98]. The Tianhe-2 (MilkyWay-2) supercomputer has a MPI broadcast bandwidth of 6.36GB/s and latency of $9\mu s$ [87]. The Titan supercomputer is built on Cray's Gemini interconnect, which can achieve a peak bandwidth of 6.9GB/s and has a latency of $1\mu s$ [99]. The Sequoia supercomputer has a 5D torus network, and each node has ten 2GB/s links [100].

For Exascale, we are envisioning better physical interconnection technologies than today, with 400Gbps fabric and even 1Tb Ethernet is on the way [101–103]. Therefore, we will simulate physical link bandwidth from 8GB/s to 64GB/s and latency from $10ns$ to $200ns$, which are likely to be achieved in the foreseeable future. Note that these are just the the properties of physical links; there are also other network parameters that are tunable in SST. However, simulating all of them is impractical and out of the scope of this study, therefore we use typical or default values for those parameters unless otherwise specified. For example, the flit size is 64 Bytes and each MPI message size (payload) is 4KB; the input latency (queuing/buffering) is $30ns$ and output latency (switching and routing decision) is $30ns$. There are also delays introduced on the host side, e.g. router to host NIC latency is $4ns$, etc.

7.4.3 Workloads

As for workloads, many previous studies of such large-scale networks have used synthetic traffic patterns [71,76,78,81,82,88,104], e.g. uniform random traffic, nearest neighbor traffic, etc. While it is a simple way to characterize networks, it also

hides communication overheads, which can be significant [105]. Therefore, we chose to use more fine-grained simulated MPI workloads offered by the SST simulator's Ember endpoint model [13]. SST not only generates traffic to the network, but also simulates the real-world behaviors during the entire life cycle of an MPI program, as well as low level protocol and interfaces. Here are brief descriptions of the workloads as well as a graph illustration of them in Figure 7.11.

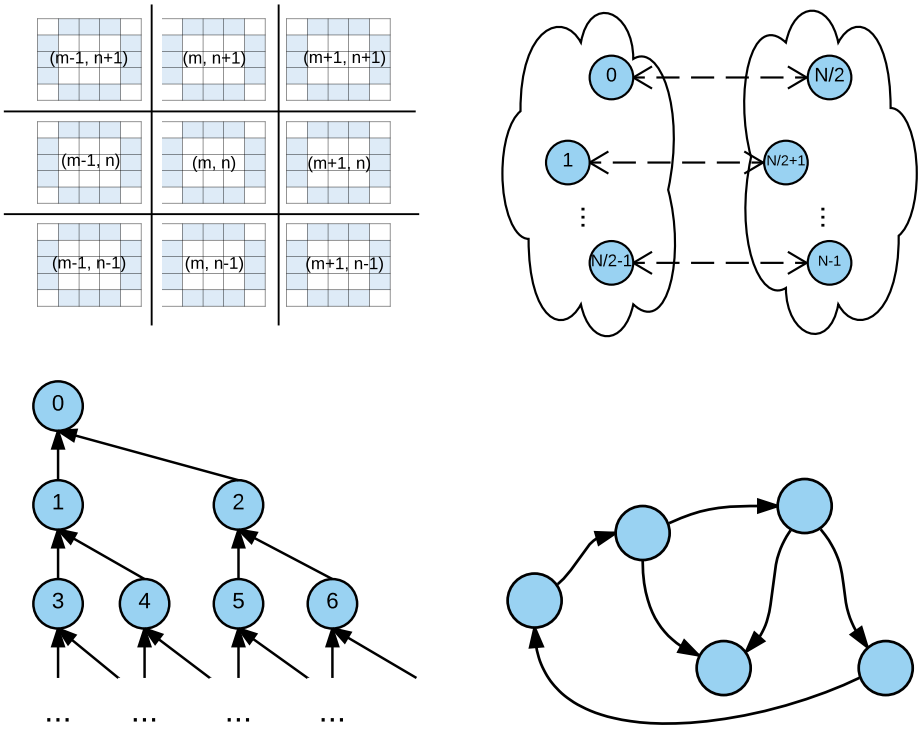


Figure 7.11: *Graphic illustrations of MPI workloads used in this study. Upper row: Halo(left), AllPingPong(right); Lower row: AllReduce(left), Random(right).*

Halo-2D: *Halo exchange* pattern is a commonly used communication pattern for domain decomposition problems [106]. Data is partitioned into grids which are mapped to MPI ranks, and at each time step, adjacent ranks exchange their boundary data.

AllPingPong: *AllPingPong* is a communication pattern that tests the network's bisection bandwidth performance: half of the ranks in the network send/receive packages to/from the other half of the network.

AllReduce: *AllReduce* tests the network's capability of data aggregation. The communication pattern resembles traffic from a tree's leaf nodes to its root. It is the reverse process of "mapping".

Random: *Random* pattern does as the name suggests: each node sending packets to uniformly random target nodes within the network. So unlike previous workloads which all have some locality or certain traffic patterns, Random does not has locality and could thus test the network's ability to handle global traffics.

Workload scaling There are 2 types of scalability measurements, strong scaling and weak scaling [107]. Strong scaling refers to a fixed problem size and increased system size, the efficiency is defined as the speed up weak scaling refers to fixed problem size on each node in the system therefore the overall workloads scales with the system size. Due to the irregular and various system sizes of topologies and different natures of workloads, it is hard to have a fixed workload and partition it across all the nodes in the system evenly. Therefore we use weak scaling workloads.

7.5 Synthetic Cycle-Accurate Simulation Results

To compare how the different topologies handle all-to-all traffic, we simulated them using a modified version of Booksim [108] , a widely used, cycle-accurate simulator for interconnect networks. It provides a set of built-in topology models and

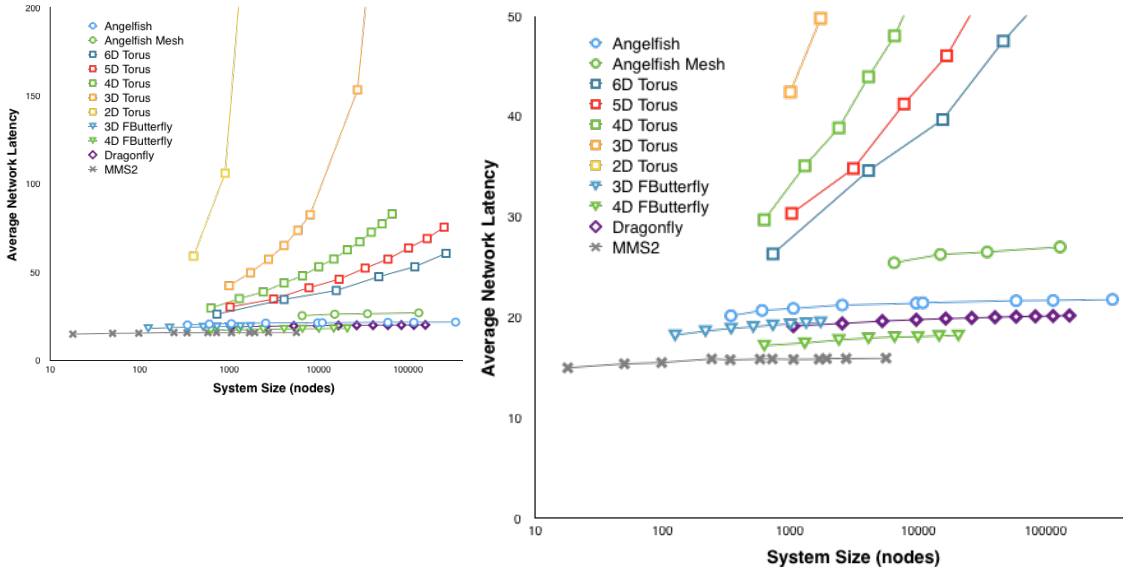


Figure 7.12: *Simulations of network topologies under constant load; the MMS2 graphs are the 2-hop Moore graphs based on MMS techniques that were used to construct the Angelfish networks.*

offers the flexibility for custom topologies by accepting a netlist. The tool uses Dijkstra's algorithm to build the minimum-path routing tables for those configurations that are not in its set of built-in topologies. We simulated injection mode with a uniform traffic pattern. The configurations simulated include the topologies described earlier, as well as 2-hop Moore graphs labeled "MMS2." These latter networks are not bounds but graphs, the same graphs used to construct the Angelfish networks studied in this analysis section; they represent sizes from 18 to 5618 nodes.

The results are shown in Figure 7.12, which presents average network latency, including transmission time as well as time spent in queues at routers. The figure shows the same graph twice, at different y-axis scales. The left graph shows enough data points to see the sharply increasing slope of the low-dimension tori. The graph

on the right shows details of the graphs with the lowest average latencies. There are several things to note. First, it is clear that, at the much higher dimensions, the high-D tori will have latencies on the same scale as the other topologies. Second, the Dragonfly networks are shown scaled out beyond 100,000 nodes, which requires several hundred ports per node, assuming routers are integrated on the CPU. Our simulations show that a configuration using an external router would incur an order of magnitude higher latencies due to congestion at the routers and longer hop counts. The Angelfish and Angelfish Mesh networks at this scale require 38 and 21 ports per node, respectively. Third, the 3D/4D Flattened Butterfly designs have identical physical organization as the 3D/4D tori; they simply use many more wires to connect nodes in each dimension. One can see the net effect: the Flattened Butterfly designs have half the latency of the same-sized tori.

7.6 Detailed Simulation Results

As mentioned earlier, our experiments cover the effective cross-product of the parameter ranges given in Table 7.1. We present slices through the dataset, from different angles, to provide the full scope of our results.

In each of the following subsections, we will discuss one aspect from our dataset.

Also, to increase the readability of data visualization, we applied the following general rules to process the graphs plotted from the dataset:

- We only present at most 2 routing algorithms for each topology in the graph

to reduce the number of datapoints in each graph. For example, the difference between deterministic and adaptive routing for Fat-tree is relatively small (comparing to Dragonfly and Fishnet-Lite) in most cases and therefore we only show the results of its adaptive routing. For those topologies with minimal, Valiant, and adaptive routings, we only present the results of adaptive and minimal routings in the graphs as they usually deliver best/worst results while VAL is often in between the two.

- We use the following abbreviations in graphs and tables for simplicity: FT3=3-level Fat-Tree, FT4=4-level Fat-Tree, DF=Dragonfly, SF=Slimfly, FN=Fishnet, FL=Fishnet-Lite, min=minimal routing, ada=adaptive routing. For example, DF-ada will refer to Dragonfly with adaptive routing.

7.6.1 Link Bandwidth

In this subsection we study the effects of link bandwidth while keeping the network scale and link latency constant, (100k-node and 100ns, respectively). We then break down our data sets by 4 workloads and plot each subset. In each plot, we use the execution time of DF-ada at 8GB/s as a performance baseline and the execution time ratio of other configurations to represent their relative performance. Each bar cluster in every graph is ordered in SF-ada, SF-min, DF-ada, DF-min, FT3-ada, FT4-ada, FL-ada, FL-min, FN-ada, FN-min from left to right

AllReduce Figure 7.13 shows the performance of different topology-routing configurations under AllReduce workloads at different bandwidths limits. AllReduce

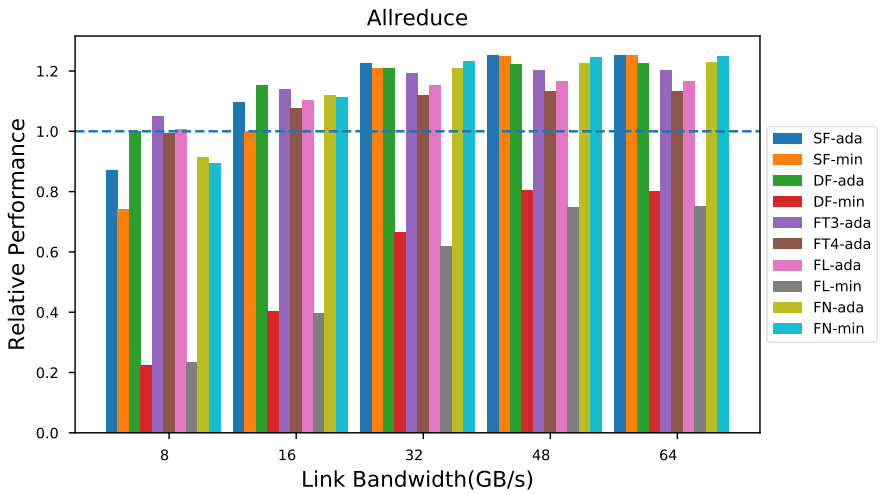


Figure 7.13: *AllReduce workload comparison for all topology-routing combinations*

aggregates traffic from all leaf nodes to one root node recursively. This traffic pattern is beneficial for topologies like Fat-tree, whose architecture resembles the software behavior, and is adversarial for irregularly constructed topologies. From Figure 7.13, we can see Dragonfly and Fishnet-lite suffers greatly when using minimal routing. Adaptive routing helps improve the performance by a factor of 5 in such cases.

We can also see that the advantage brought by topology is significant when lower bandwidths are available. e.g. both 3 level and 4 level Fat-trees have relatively better performances at 8GB/s bandwidth limits. As bandwidth limits increases, the difference in performance between topologies decreases, and ones with lower diameter and higher bisection bandwidth start to outperform others (although with thin margins).

AllPingPong The performance of AllPingPong workloads can represent the bisection bandwidth capability of a network. Not surprisingly, Fat-tree topologies

again performs very well when the bandwidth limit is lower due to its high bisection bandwidth design as shown in Figure 7.14. But as the bandwidth limit increases, other topologies are no longer bounded by bandwidth and start to outperform Fat-trees.

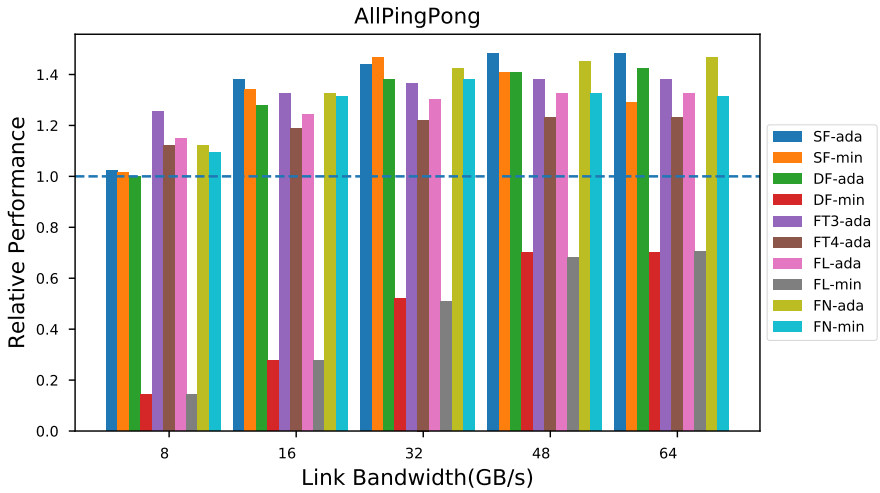


Figure 7.14: *AllPingpong workload comparison for all topology-routing combinations*

Halo: Halo represents a nearest neighbor communication pattern, therefore topologies with better locality generally perform better. Consequently, without sufficient bandwidth, Fat-trees performs better than other topologies. Also note that DF-ada performs almost as good as Fat-trees at 8GB/s bandwidth, while DF-min is the slowest among all setups. Part of the reason, as previously mentioned, is the global link between groups gets congested. The other part of the reason is that Dragonfly has better "locality" because all the routers within a group are fully connected and can guarantee 1 router hop for the hosts within a group. While Slimfly connects even more hosts within 1 router hop, it's not as good as Dragonfly under

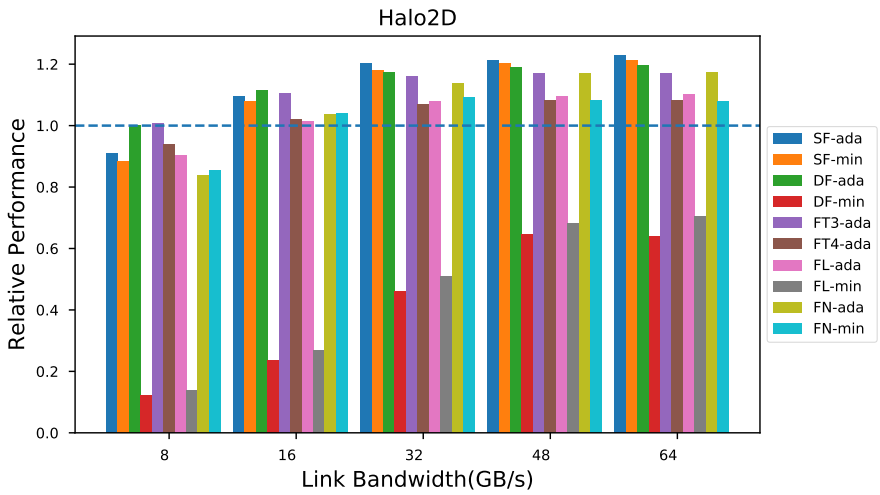


Figure 7.15: *Halo workload comparison for all topology-routing combinations*

8GB/s and 16GB/s bandwidth limit. The reason behind this is consecutive MPI Ranks (logical ranks) will be mapped to the hosts within a group for Dragonfly, but they are not guaranteed to be mapped to the adjacent routers in Slimfly. Consequently, some of the consecutive ranks in Slimfly will sometimes have 2-hop latency instead of 1-hop as in Dragonfly.

Random The results of random are largely different from all other workloads as shown in Figure 7.16. This is because random workloads generates uniform traffic pattern across all nodes, which would make use of almost no locality and the load on all the links are inherently balanced.

As a result, low diameter topologies with more path diversities, such as Fishnet, outperforms other topologies at lower bandwidth limits. The differences between topologies at lower bandwidth limits also significantly drops to a factor of less than 2 (comparing to a factor of 5 to 7 for other workloads). At higher band-

width limit (64GB/s), the performance differences are still very significant where diameter-2 graphs beats 4-level Fat-tree by 20.5%.

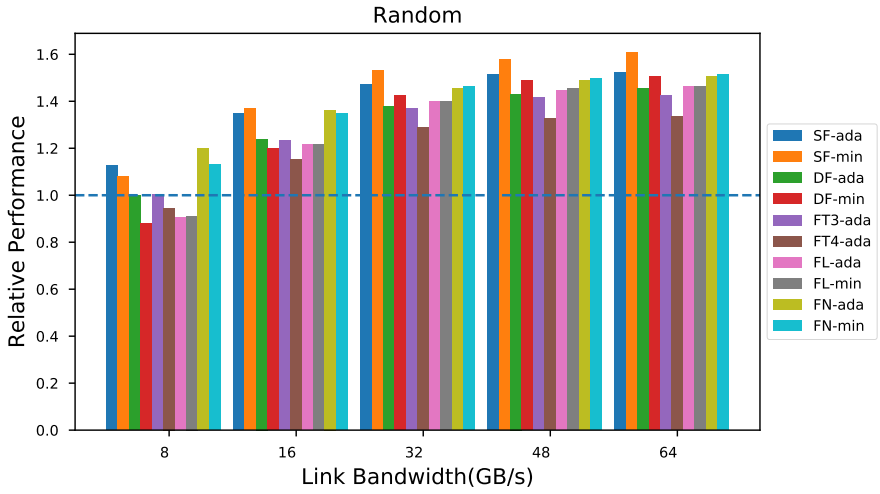


Figure 7.16: *Random workload comparison for all topology-routing combinations*

Discussion We now compare across the 4 workloads and see how bandwidth affects performance for each topology. Dragonfly and Fishnet-lite with minimal routing benefit most from the growths of global link bandwidth. Increasing the bandwidth from 8GB/s to 64GB/s decreases the execution time by up to 6 to 7 times. Other topology/routing combinations tend not to gain as much performance from the bandwidth increase, but there is still an average 20% to 50% performance gain from 8GB/s to 16GB/s. To be more specific, FN-ada has a gain of 17%, SF-ada 26%, FT3-ada 36%, FL-ada 43% and DF-ada 56%.

Under our setup, bandwidth will no longer be a major bottleneck from 32GB/s and beyond as evident from Figures 7.13 to 7.16. Moving forward, this is not saying that bandwidth is unimportant once it's greater than 32GB/s; the demand

for bandwidth can always be elevated by factors such as application behavior or node level architecture, e.g. if an endpoint utilizes GPUs or other accelerators that generate significantly more throughput, its demand for bandwidth can be very high. Therefore it might be more reasonable to assume that bandwidth demands will not be easily satisfied, and that the data points transition from 8GB/s to 16GB/s will be more likely to represent the real-world situations of how bandwidth increases can benefit performance.

7.6.2 Link Latency

In this section we will discuss how global link latency can affect network performance. We configured the physical link latency from 10ns to 200ns, and within this range, most network topologies only suffers a less than 20% slowdown moving from 10ns links to 200ns links. This indicates that most of these configurations are not latency sensitive in this range.

The only two exceptions here are Dragonfly and Fishnet-lite with minimal routing, both of which witness a slowdown of a factor of 2 moving from 10ns to 200ns latency. The global links between router groups here once again becomes the bottleneck, and it can be alleviated by using adaptive routing algorithms.

These results imply that within 200ns, link latency does not significantly sway the overall performance. Therefore, system architects may be able to exchange an increase in link latency, for greater benefits elsewhere in the system. For example, allowing more latency will extend the maximum allowable physical space to build the

system, enabling more flexibility in physical cabinets placement, cable management, and thermal dissipation, etc.

7.6.3 Performance Scaling Efficiency

All the topologies that we choose to study in this chapter have constant network diameters with regards to the scale of the network. So, as the network size scales up, the average distance between 2 nodes will remain the same. This does not mean there will be no performance degradations, as we will explain later with examples from our simulation data.

We simulated both 50k-node and 100k-node scale networks, each with more than 1,000 data points, on different topologies, workloads, and network parameters. By looking at this broad range of configurations, we are able to get a comprehensive view of how each topology scales.

To measure the scaling efficiency, we first find a pair of simulation data points that have the exact same configuration except for the number of nodes. Then we take the ratio of execution time of the one with 50k-node to the one with 100k-node. If there is performance degradation, meaning the same amount of workload takes more time to finish on 100k-node than 50k-node network, then this ratio will be less than 1. So the closer this ratio is to one, the better scaling efficiency the topology has.

By doing so, we obtain more than 1,000 scaling efficiency ratios for different workload, topology, and network parameter combinations. Due to the large volume

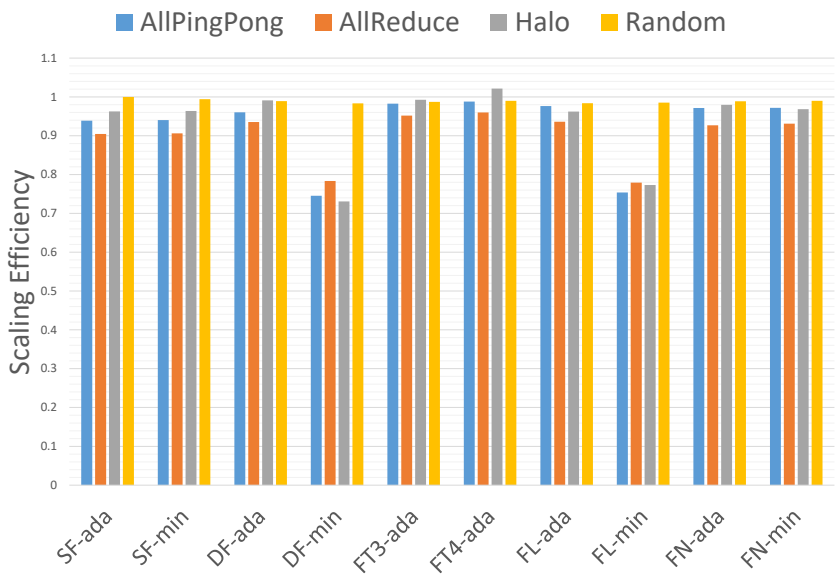


Figure 7.17: *Averaged scaling efficiency from 50k-node to 100k-node*

of the data, we turn to a statistical approach. We observed that the scaling efficiency is relatively consistent for each workload-topology-routing combination, therefore we took the average of all the data points with the same workload-topology-routing configuration, and further reduced the number of data points to 40, as shown in Figure 7.17. We calculated the standard deviation for the averaged data points, and most standard deviations are below 0.01 (about 1% of the basis), indicating these averaged numbers are representative for their samples.

One would immediately notice in Figure 7.17 that unlike all other setups, Dragonfly and Fishnet-lite both have poor scaling efficiency when using minimal routing. The reason being that, even though the network diameter does not change, the number of nodes within a group/subnet increases. Dragonfly and Fishnet-lite both only have one global link per router group, and they will be more likely to be congested under non-uniform pattern workloads. For *Random* workload, the

increased traffic generated by the host in the group/subnet are evenly distributed to more global links instead of a specific global link, thus it has good scaling efficiency. (In fact, for the same reason, *Random* has the best scaling efficiency over almost all setups)

Also note that the scaling efficiency for 4-level Fat-tree with Halo workload exceeds 1. This is because when we scale from 50,000 to 100,000 nodes, the number of nodes per router at bottom level of the Fat-tree increases, therefore more "neighbor" nodes are available within a shorter distance, which benefits nearest neighbor traffic such as Halo.

To conclude, all the topologies studied in this chapter have decent scaling efficiency (greater than 0.9) with appropriate routing algorithms, which is a desired feature when moving to even larger system.

7.6.4 Stress Test

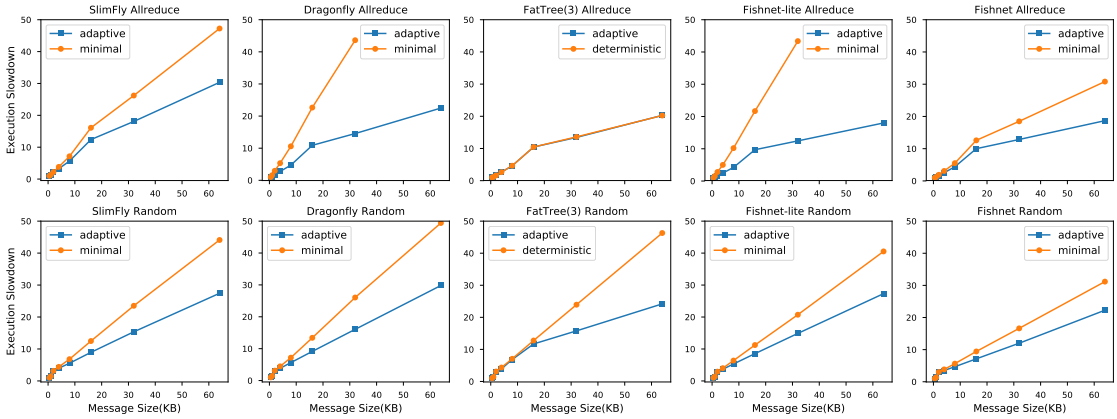


Figure 7.18: *Execution slowdown of different topologies under increasing workload*

In this subsection, we stress test topologies with increasing workloads. We will

keep the network parameters constant and increase the workload on each topology. Then we evaluate the topology's ability to handle increasing workload by observing the increase in execution time.

In this series of tests, we limit the physical link bandwidth to $8GB/s$ to make sure that light workloads are also able to cause congestions in the network, so that the efforts of increasing workloads will not be offset by high performance network parameters.

As for workloads, previous results have shown AllReduce generates adversarial traffics for most topologies while Random is benign to most topologies. Therefore we choose these two workloads for this test. To increase the workload, we double the MPI message size each time, from 512 Bytes to 64KB, which results in: 1. more packets to be sent for a message and thus more congestion in a network; 2. the input/output buffers will be filled more quickly, and NICs will have to stall to wait until the buffer is available.

Figure 7.18 shows the execution slowdown of different topologies under increasing AllReduce and Random workloads, respectively. The execution time of 512B message size for each configuration is chosen as the baseline (1).

Looking at the upper row of Figure 7.18, we can tell that Fishnet and Fishnet-lite has the modest slowdown of less than 20x in AllReduce workload when using adaptive routing, while all other configurations have more than 20x slowdown. This indicates the high bisection bandwidth and high path diversity designs of Fishnet/Fishnet-lite contributes to their performance in handling adversarial workloads.

The lower row of Figure 7.18 shows the slowdown of Random workload. Due to the benign nature of Random workload, the difference in performance is not as huge as it is for AllReduce workloads when the workload increases, but it can still be seen that high bisection bandwidth architectures such as Fishnet, and Fat-tree outperform others under increasing workloads.

The effectiveness of routing algorithms against adversarial traffics could also be reflected here. By applying proper routing algorithms, the topology's ability to handle heavy workloads can be strengthened. For example, Fishnet-lite reduced the slowdown from 40x to 20x in AllReduce workload when moving from minimal routing to adaptive routing. This further proves the effectiveness of our proposed routing algorithms for Fishnet topologies.

The performance difference from routing algorithm for Fat-tree is almost negligible for AllReduce workload. This is because AllReduce is considered to be a benign traffic pattern for Fat-tree, and increasing workload does not affect the routing decision heavily. As a contrast, routing algorithm under Random workload, which causes packets to traverse more distances than AllReduce, makes more of a difference for Fat-tree, as shown in Figure 7.18.

7.7 Conclusion

In this chapter, we study a wide range of network topologies that are promising candidates for large scale high performance computing systems. We extend SST to perform large scale, fine-grained simulations for each concerned topology with

different routing algorithms, various workloads and network parameters at different scales.

From a network parameter perspective, our study shows all topologies can gain a decent amount of performance from the increase of physical link bandwidth. However, the amount of performance gain from the growth of bandwidth differs greatly from topology to topology (ranging from 17% to 56%), as shown in Section 7.6.1. As for physical link latency, topologies with higher network diameters are naturally more sensitive to link latency, but in general, the latency range studied in this chapter (10ns to 200ns) makes less contributions to the overall system performance. If allowing more latency will be beneficial for the overall system design, it might be a worthy trade-off.

The results of performance scaling efficiency and the stress test show that the studied topologies all have good performance scaling efficiency if properly set up, but their ability to handle increased workloads differs. This provides useful insights on the scenarios that we are yet unable to simulate in this study. e.g. larger scale network with even heavier workloads.

Furthermore, we identified various cases during our study where software behavior can result in significant differences in system performance. Although it is well known, we are the first to provide examples based on simulation data for a lot of the recently proposed topologies combined with network parameters, and these examples will be helpful for software optimization.

Chapter 8: Conclusions

This dissertation proposes a series of measures and methods to address the issues that memory system simulation could not keep up with the heterogeneity and scalability of modern memory systems.

We first developed a feature-rich, validated cycle accurate simulator that can simulate a variety of modern DRAM and even non-volatile memory protocols. We extensively validated the simulator, and conduct a thorough DRAM architecture characterization with cycle accurate simulations, which provides insights on DRAM architectures and system designs.

Based on the validated cycle accurate simulator, we explored methods to promote the scalability of memory simulator with minimized impact on accuracy, and overcame the limitations of cycle accurate memory models.

We proposed and implemented an effective parallel memory simulator with a relaxed synchronization scheme named MegaTick. We also improved the method with accuracy mitigation, which helps achieve more than a factor of two speedup on multi-channel memory simulation at the cost of one percent or less overall accuracy.

We further explored the feasibility of using a statistical/machine learning model to accelerate DRAM modeling. We propose modeling DRAM timings as

a classification problem and successfully prototyped a decision tree model that sped up simulation 2 to 200 times with modest errors in latency modeling.

Finally, we studied and experimented large scale interconnect topologies for high performance computing memory systems with a parallel distributed simulator, and demonstrated the effectiveness and scalability of our proposed topology design.

Bibliography

[1] Sadagopan Srinivasan. *Prefetching vs. the Memory System: Optimizations for Multi-core Server Platforms*. PhD thesis, University of Maryland, Department of Electrical & Computer Engineering, 2007.

[2] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, and Disk*. Morgan Kaufmann, 2007.

[3] Doug Burger, James R. Goodman, and Alain Kagi. Memory bandwidth limitations of future microprocessors. In *Proc. 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 78–89, Philadelphia PA, May 1996.

[4] Brian Dipert. The slammin, jammin, DRAM scramble. *EDN*, 2000(2):68–82, January 2000.

[5] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary DRAM architectures. In *Proc. International Symposium on Computer Architecture (ISCA)*, pages 222–233, June 1999.

[6] Vinodh Cuppu and Bruce Jacob. Concurrency, latency, or system overhead: Which has the largest impact on uniprocessor DRAM-system performance? In *Proc. 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pages 62–71, Goteborg, Sweden, June 2001.

[7] Steven Przybylski. *New DRAM Technologies: A Comprehensive Analysis of the New Architectures*. MicroDesign Resources, Sebastopol CA, 1996.

[8] Paul Rosenfeld. *Performance Exploration of the Hybrid Memory Cube*. PhD thesis, University of Maryland, Department of Electrical & Computer Engineering, 2014.

[9] JEDEC. *Low Power Double Data Rate (LPDDR4), JESD209-4A*. JEDEC Solid State Technology Association, November 2015.

[10] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[11] DRAM Micron. System power calculators, 2014.

[12] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Norbert Wehn, and Kees Goossens. Drampower: Open-source dram power & energy estimation tool. *URL: http://www. drampower. info*, 22, 2012.

[13] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, R Risen, Jeanine Cook, Paul Rosenfeld, E Cooper-Balls, et al. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 2011.

[14] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer architecture news*, volume 41, pages 475–486. ACM, 2013.

[15] 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*, 39(2):1–7, 2011.

[16] Matthias Jung, Carl C Rheinländer, Christian Weis, and Norbert Wehn. Reverse engineering of drams: Row hammer with crosshair. In *Proceedings of the Second International Symposium on Memory Systems*, pages 471–476. ACM, 2016.

[17] Yunus Cengel. *Heat and mass transfer: fundamentals and applications*. McGraw-Hill Higher Education, 2014.

[18] James W Demmel, John R Gilbert, and Xiaoye S Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952, 1999.

[19] Tiantao Lu, Caleb Serafy, Zhiyuan Yang, Sandeep Kumar Samal, Sung Kyu Lim, and Ankur Srivastava. Tsv-based 3-d ics: Design methods and tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(10):1593–1619, 2017.

[20] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[21] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2016.

[22] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. Usimm: the utah simulated memory module. *University of Utah, Tech. Rep*, 2012.

[23] Min Kyu Jeong, Doe Hyun Yoon, and Mattan Erez. Drsim: A platform for flexible dram system research. *Accessed in: http://lph. ece. utexas. edu/public/DrSim*, 2012.

[24] William A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[25] Milan Radulovic, Darko Zivanovic, Daniel Ruiz, Bronis R. de Supinski, Sally A. McKee, Petar Radojković, and Eduard Ayguadé. Another trip to the wall: How much will stacked DRAM benefit HPC? In *Proceedings of the 2015 International Symposium on Memory Systems*, MEMSYS '15, pages 31–36, Washington DC, DC, USA, 2015. ACM.

[26] JEDEC. *DDR3 SDRAM Standard, JESD79-3*. JEDEC Solid State Technology Association, June 2007.

[27] JEDEC. *DDR4 SDRAM Standard, JESD79-4*. JEDEC Solid State Technology Association, September 2012.

[28] JEDEC. *Low Power Double Data Rate 3 (LPDDR3), JESD209-3*. JEDEC Solid State Technology Association, May 2012.

[29] JEDEC. *Graphics Double Data Rate (GDDR5) SGRAM Standard, JESD212C*. JEDEC Solid State Technology Association, February 2016.

[30] JEDEC. *High Bandwidth Memory (HBM) DRAM, JESD235*. JEDEC Solid State Technology Association, October 2013.

[31] JEDEC. *High Bandwidth Memory (HBM) DRAM, JESD235A*. JEDEC Solid State Technology Association, November 2015.

[32] HMC Consortium. *Hybrid Memory Cube Specification 1.0*. Hybrid Memory Cube Consortium, 2013.

[33] HMC Consortium. *Hybrid Memory Cube Specification 2.0*. Hybrid Memory Cube Consortium, 2014.

[34] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. Buffer On Board memory systems. In *Proc. 39th International Symposium on Computer Architecture (ISCA 2012)*, pages 392–403, Portland OR, June 2012.

[35] Brinda Ganesh, Aamer Jaleel, David Wang, and Bruce Jacob. Fully-Buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling. In *Proc. 13th International Symposium on High Performance Computer Architecture (HPCA 2007)*, pages 109–120, Phoenix AZ, February 2007.

[36] Richard Sites. It's the memory, stupid! *Microprocessor Report*, 10(10), August 1996.

[37] David Zaragoza Rodríguez, Darko Zivanović, Petar Radojković, and Eduard Ayguadé. *Memory Systems for High Performance Computing*. Barcelona Supercomputing Center, 2016.

[38] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–35. ACM, 2011.

[39] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[40] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation. *Web Copy: http://www. glue. umd. edu/ajaleel/workload*, 2010.

[41] Piotr R Luszczek, David H Bailey, Jack J Dongarra, Jeremy Kepner, Robert F Lucas, Rolf Rabenseifner, and Daisuke Takahashi. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 213, 2006.

[42] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312:150, 2013.

[43] Gwangsun Kim, John Kim, Jung Ho Ahn, and Jaeha Kim. Memory-centric system interconnect design with Hybrid Memory Cubes. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 145–156. IEEE Press, 2013.

[44] Bruce Jacob and David Tawei Wang. System and method for performing multi-rank command scheduling in DDR SDRAM memory systems, June 2009. US Patent No. 7,543,102.

[45] Micron. TN-41-01 Technical Note—calculating memory system power for DDR3. Technical report, Micron, August 2007.

[46] Dean Gans. Low power DRAM evolution. In *JEDEC Mobile and IOT Forum*, 2016.

[47] J. Thomas Pawlowski. Hybrid Memory Cube (HMC). In *HotChips 23*, 2011.

[48] AMD. High-bandwidth memory–reinventing memory technology. https://www.amd.com/Documents/High-Bandwidth-Memory-HBM.pdf, 2015.

[49] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 318–319. ACM, 2003.

[50] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA-16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12. IEEE, 2010.

[51] Trevor E Carlson, Wim Heirmant, and Lieven Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[52] Sadagopan Srinivasan, Li Zhao, Brinda Ganesh, Bruce Jacob, Mike Espig, and Ravi Iyer. Cmp memory modeling: How much does accuracy matter? 2009.

[53] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: a memory system simulator. *ACM SIGARCH Computer Architecture News*, 33(4):100–107, 2005.

[54] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer architecture letters*, 15(1):45–49, 2015.

[55] Andreas Hansson, Neha Agarwal, Aasheesh Kolli, Thomas Wenisch, and Aniruddha N Udipi. Simulating dram controllers for future system architecture exploration. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 201–210. IEEE, 2014.

[56] Matthias Jung, Christian Weis, Norbert Wehn, and Karthik Chandrasekar. Tlm modelling of 3d stacked wide i/o dram subsystems: a virtual platform for memory controller design space exploration. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, page 5. ACM, 2013.

[57] Hyojin Choi, Jongbok Lee, and Wonyong Sung. Memory access pattern-aware dram performance model for multi-core systems. In *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, pages 66–75. IEEE, 2011.

[58] George L Yuan, Tor M Aamodt, et al. A hybrid analytical dram performance model. In *Proc. 5th Workshop on Modeling, Benchmarking and Simulation*, 2009.

[59] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. Wait of a decade: Did spec cpu 2017 broaden the performance horizon? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 271–282. IEEE, 2018.

[60] Rommel Sánchez Verdejo, Kazi Asifuzzaman, Milan Radulovic, Petar Radojković, Eduard Ayguadé, and Bruce Jacob. Main memory latency simulation: the missing link. In *Proceedings of the International Symposium on Memory Systems*, pages 107–116. ACM, 2018.

[61] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[62] Annalisa Barla, Francesca Odone, and Alessandro Verri. Histogram intersection kernel for image classification. In *Proceedings 2003 international conference on image processing (Cat. No. 03CH37429)*, volume 3, pages III–513. IEEE, 2003.

[63] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[64] Andy Liaw, Matthew Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[65] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[66] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.

[67] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, W Carson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, et al. Exascale computing study: Technology challenges in achieving exascale systems. 2008.

[68] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[69] J Dongarra, P Luszczek, and M Heroux. Hpcg: one year later. *ISC14 Top500 BoF*, 2014.

[70] Richard Murphy. On the effects of memory latency and bandwidth on supercomputer application performance. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 35–43. IEEE, 2007.

[71] N Jiang, G Michelogiannakis, D Becker, B Towles, and W Dally. Booksim interconnection network simulator. *Online, https://nocs. stanford. edu/cgibin/trac. cgi/wiki/Resources/BookSim*.

[72] John Kim, William J. Dally, Steve Scott, and Dennis Abts. Cost-efficient dragonfly topology for large-scale systems. In *Optical Fiber Communication Conference and National Fiber Optic Engineers Conference*, page OTuI2. Optical Society of America, 2009.

[73] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray cascade: A scalable hpc system based on a dragonfly network. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.

[74] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 366–376, Nov 2012.

[75] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S Schreiber. Hyperx: topology, routing, and packaging of efficient large-scale networks. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 41. ACM, 2009.

[76] Maciej Besta and Torsten Hoefler. Slim fly: a cost effective low-diameter network topology. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 348–359. IEEE Press, 2014.

[77] Christopher D Carothers, David Bauer, and Shawn Pearce. Ross: A high-performance, low-memory, modular time warp system. *Journal of Parallel and Distributed Computing*, 62(11):1648–1669, 2002.

[78] Misbah Mubarak, Christopher D Carothers, Robert B Ross, and Philip Carns. A case study in using massively parallel simulation for extreme-scale torus network codesign. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 27–38. ACM, 2014.

[79] Cristóbal Camarero, Enrique Vallejo, and Ramón Beivide. Topological characterization of hamming and dragonfly networks and its implications on routing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):39, 2015.

[80] Georgios Kathareios, Cyriel Minkenberg, Bogdan Prisacari, German Rodriguez, and Torsten Hoefler. Cost-effective diameter-two topologies: analysis and evaluation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015.

[81] Noah Wolfe, Christopher D Carothers, Misbah Mubarak, Robert Ross, and Philip Carns. Modeling a million-node slim fly network using parallel discrete-event simulation. In *Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation*. ACM, 2016.

[82] Shang Li, Po-Chun Huang, David Banks, Max DePalma, Ahmed Elshaarany, Scott Hemmert, Arun Rodrigues, Emily Ruppel, Yitian Wang, Jim Ang, et al. Low latency, high bisection-bandwidth networks for exascale memory systems. In *Proceedings of the Second International Symposium on Memory Systems*, pages 62–73. ACM, 2016.

[83] William James Dally and Brian Patrick Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[84] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6d mesh/torus interconnect for exascale computers. *Computer*, 42(11):0036–41, 2009.

[85] Charles E Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE transactions on Computers*, 100(10):892–901, 1985.

[86] Charles Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 1953.

[87] Jack Dongarra. Visit to the national university for defense technology changsha, china. *Oak Ridge National Laboratory, Tech. Rep., June*, 2013.

[88] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 77–88. IEEE Computer Society, 2008.

[89] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182. IEEE Computer Society, 2007.

[90] Wen-Tao Bao, Bin-Zhang Fu, Ming-Yu Chen, and Li-Xin Zhang. A high-performance and cost-efficient interconnection network for high-density servers. *Journal of computer science and Technology*, 29(2):281–292, 2014.

[91] Crispín Gomez, Francisco Gilabert, María Engracia Gomez, Pedro López, and José Duato. Deterministic versus adaptive routing in fat-trees. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[92] Bruce Jacob. The 2 petaflop, 3 petabyte, 9 tb/s, 90 kw cabinet: A system architecture for exascale and big data.

[93] Leslie G. Valiant. A scheme for fast parallel communication. *SIAM journal on computing*, 11(2):350–361, 1982.

[94] Arjun Singh. *Load-balanced routing in interconnection networks*. PhD thesis, Stanford University, 2005.

[95] Nan Jiang, John Kim, and William J Dally. Indirect adaptive routing on large scale interconnection networks. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 220–231. ACM, 2009.

[96] William J Dally and Charles L Seitz. Interconnection networks. *IEEE Transactions on computers*, 36(5), 1987.

[97] Sst. `http://sst-simulator.org`, 2017.

[98] Jack Dongarra. Report on the sunway taihulight system. *PDF). www. netlib. org. Retrieved June*, 20, 2016.

[99] Abhinav Vishnu, Monika ten Bruggencate, and Ryan Olson. Evaluating the potential of cray gemini interconnect for pgas communication runtime systems. In *2011 IEEE 19th Annual Symposium on High Performance Interconnects*, pages 70–77. IEEE, 2011.

[100] Sébastien Rumley, Dessislava Nikolova, Robert Hendry, Qi Li, David Calhoun, and Keren Bergman. Silicon photonics for exascale systems. *Journal of Lightwave Technology*, 33(3):547–562, 2015.

[101] Bob Metcalfe. Toward terabit ethernet. In *Conference on Optical Fiber Communication (OFC)*. Optical Society of America, 2008.

[102] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[103] Xiaogeng Xu, Enbo Zhou, Gordon Ning Liu, Tianjian Zuo, Qiwen Zhong, Liang Zhang, Yuan Bao, Xuebing Zhang, Jianping Li, and Zhaohui Li. Advanced modulation formats for 400-gbps short-reach optical inter-connection. *Optics express*, 23(1):492–500, 2015.

[104] Ning Liu, Adnan Haider, Xian-He Sun, and Dong Jin. Fattreesim: Modeling large-scale fat-tree networks for hpc systems and data centers using parallel and discrete event simulation. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pages 199–210. ACM, 2015.

[105] Douglas Doerfler and Ron Brightwell. Measuring mpi send and receive overhead and application availability in high performance network interfaces. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 331–338. Springer, 2006.

[106] Caoimhín Laoide-Kemp. Investigating mpi streams as an alternative to halo exchange. 2015.

[107] Measuring parallel scaling performance. `https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance`.

[108] Nan Jiang, James Balfour, Daniel U Becker, Brian Towles, William J Dally, George Michelogiannakis, and John Kim. A detailed and flexible cycle-accurate network-on-chip simulator. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 86–96. IEEE, 2013.