Architectual Techniques to Enhance DRAM Scaling

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Yoongu Kim

B.S., Electrical Engineering, Seoul National University

Carnegie Mellon University

Pittsburgh, PA

June, 2015

Abstract

For decades, main memory has enjoyed the continuous scaling of its physical substrate: DRAM (Dynamic Random Access Memory). But now, DRAM scaling has reached a threshold where DRAM cells cannot be made smaller without jeopardizing their robustness. This thesis identifies two specific challenges to DRAM scaling, and presents architectural techniques to overcome them.

First, DRAM cells are becoming less reliable. As DRAM process technology scales down to smaller dimensions, it is more likely for DRAM cells to electrically interfere with each other's operation. We confirm this by exposing the vulnerability of the latest DRAM chips to a reliability problem called disturbance errors. By reading repeatedly from the same cell in DRAM, we show that it is possible to corrupt the data stored in nearby cells. We demonstrate this phenomenon on Intel and AMD systems using a malicious program that generates many DRAM accesses. We provide an extensive characterization of the errors, as well as their behavior, using a custom-built testing platform. After examining various potential ways of addressing the problem, we propose a low-overhead solution that effectively prevents the errors through a collaborative effort between the DRAM chips and the DRAM controller.

Second, DRAM cells are becoming slower due to worsening variation in DRAM process technology. To alleviate the latency bottleneck, we propose to unlock fine-grained parallelism within a DRAM chip so that many accesses can be served at the same time. We take a close look at how a DRAM chip is internally organized, and find that it is divided into small partitions of DRAM cells called subarrays. Although the subarrays are mostly independent, they occasionally rely upon some global circuit components that force the subarrays to be operated one at a time. To overcome this limitation, we devise a series of non-intrusive changes to DRAM architecture that increases the autonomy of the subarrays and allows them to be accessed concurrently. We show that such parallelism across subarrays provides large performance gains at low cost.

Lastly, we present a powerful DRAM simulator that facilitates the design space exploration of main memory. Unlike previous simulators, our simulator is easy to modify, allowing DRAM architectural changes to be modeled quickly and accurately. This is why our simulator is able to provide out-of-the-box support for a wide array of contemporary DRAM standards. Our simulator is also the fastest, outperforming the next fastest simulator by more than a factor of two.

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Onur Mutlu. He embodies the quotation from Quintilian, the Roman rhetorician who said, “We should not write so that it is possible for the reader to understand us, but so that it is impossible for him to misunderstand us.” He always strived for the highest level of clarity and thoroughness in research, and propelled me to become a better thinker, writer, and presenter. He gave me the opportunities, the resources, and the guidance that allowed me to be who I am today. This thesis was only made possible by his encouragement and nurturing throughout all of my years in the SAFARI research group.

I am grateful to the members of my thesis committee: Professor Onur Mutlu, Professor James Hoe, Professor Todd Mowry, and Professor Trevor Mudge. They truly cared about my research and me as an individual. I also thank Professor Mor Harchol-Balter who introduced me to research, and taught me how to use precise language and high-level abstractions in technical prose.

I am grateful to my internship mentors, who gave me the freedom to pursue my ideas, the counsel to achieve my goals, and their friendship in my times of turmoil: Chris Wilkerson, Moin Qureshi, Sangyeun Choi, Konrad Lai, Uksong Kang, Suresh Chittor, Suneeta Sah, and Michele Franceschini. I thank the Korea Foundation for Advanced Studies, Intel Corporation, and Samsung Electronics for their generous financial support.

I am grateful to Samantha Goldstein and Elaine Lawrence, ECE departmental advisors who sent me many a reminder to make sure that I turned in my paperwork on time, and who took a personal interest in my academic progress.

During graduate school, members of the SAFARI research group have been like family to me. Justin Meza was a force for calm in my life with his even-keeled, zen-like outlook on everything and everyone. Chris Fallin was a source of inspiration for me with his computer wizardry, self discipline, and engaging insights about technology and research. Chris Craik would never fail to catch me off-guard with his dry, situational one-liners that made me burst out in laughter. Rachata Ausavarungnirun was not only a gifted chef, but was also one of the nicest persons I have ever met, and I am grateful for the countless occasions when he lent me a helping hand. Donghyuk Lee was our resident DRAM expert, who ever so graciously and patiently gave me indispensable guidance on many facets of my research. Hongyi Xin spiced up my dinners by introducing me to the newest Chinese restaurants, where he taught me so much about Chinese history and politics. Kevin Chang was always kind enough to tolerate my dumb jokes, and we had lots of fun hanging out together — especially when drunk. Samira Khan was a thoughtful and empathetic listener, who — with her sidekick Omar Chowdhury — could also be hilariously and mercilessly self-deprecating. Vivek Seshadri was never hesitant to offer incisive criticism that would invariably rescue me from an impasse in writing. Lavanya Subramanian is a genuinely caring person who would always look out for everyone around her. Gennady Pekhimenko tempered me in my thoughts and opinions by challenging my biases and preventing me from jumping to conclusions. Jamie Liu was a talented programmer who brought depth and clarity to every discussion. I also thank other members of the SAFARI research group for their companionship and assistance: HanBin Yoon, Ben Jaiyen, Yixin Luo, Nandita Vijaykumar, Yang Li, Kevin Hsieh, Amirali Boroumand, and Saugata Ghose.

At Carnegie Mellon, I had the privilege of making many great friends. Michael Papamichael blew me away with his work ethic and detail-orientedness, and I am indebted to him for the many all-nighters that we pulled together for that one paper. Michelle Goodstein and I were often the only ones to be left in the office during the late evenings, and she was my shoulder to cry on when I was particularly stressed over a looming deadline. Eric Chung was the wise and senior graduate student whom I wanted to grow up and be like. Peter Klemperer had the most fascinating stories to tell about his escapades and misdemeanors from a previous life. Evangelos Vlachos was a party animal whom I sorely missed after he left for Switzerland. I thank Ross Daly, Weikun Yang, Jeremie Kim, and Ji Hye Lee for their tremendous research contributions as undergraduate/masters interns. I thank Seungil Huh, Jee Eun Song, Sung Won Park, and Dongsu Han for helping me get settled into Pittsburgh when I first arrived, and Jane Seo for her care and understanding. I also thank Mei Chen, Michael Kozuch, Varun Gupta, Peter Milder, Olatunji Ruwase, Anshul Gandhi, Gabe Weisz, Alexey Tumanov, Berkin Akin, and Eugene Wang.

Although our paths diverged geographically, it was comforting to know that my old buddies — Heeyoung Lee and Hongsup Shin — were always there to support me with their camaraderie and solidarity. Lastly, I would like to acknowledge the unwavering love from my family: my grandmother Worsoon Kim, my parents Hyosoon and Dohyun Kim, and my sister Taeyeun Kim.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Main memory is a fundamental building block of computing systems. As its name suggests, main memory is the primary repository of data, storing the working set of software applications which must be accessed quickly and frequently. For four decades and counting, the overwhelmingly preferred physical substrate for implementing main memory has been Dynamic Random Access Memory (DRAM) — so much so that they have come to be synonymous with one another. Compared to other alternatives, DRAM strikes the most suitable balance between cost and latency that is conducive for achieving both large capacity and high performance at the same time. And with the seemingly inexorable march of Moore's Law, DRAM process technology too has scaled down to ever smaller dimensions, providing main memory with compounded increases in capacity and performance over the course of forty years. In fact, such scaling of DRAM — in the domain of circuits and devices — has been the principal driving force behind the improvements to main memory, allowing each successive generation of computing systems to satisfy the growing memory requirements of increasingly data-intensive applications.

1.1. The Problem: DRAM Scaling at Risk

However, DRAM scaling — taken for granted for so long — has now reached a critical juncture in which it faces difficult technological hurdles [21, 23, 77, 119, 165] that threaten its future sustainability. In the past, the continuous scaling of DRAM was achieved by the winning combination of two important factors: *(i)* technological feasibility and *(ii)* economic viability. Simply put, making smaller DRAM cells was both possible and worthwhile. But what has changed now is the second factor — although smaller cells are still attainable, they are not as attractive as they once were. In fact, DRAM cells fabricated using the latest generations of process technology (20–40 nanometers) are proving to be significantly slower and faultier than they were in the past. This is because, at smaller dimensions, cells are more susceptible to imperfections arising from process variation, as well as being more vulnerable to data corruption caused by electrical noise. Importantly, the severity of these problems is of the magnitude that — if left unaddressed — they have the potential to have a detrimental impact on the *reliability* and *performance* of main memory.

Consequently, this leaves DRAM manufacturers in a position where they must choose between two undesirable courses of action if they are to continue scaling DRAM: *(i)* sacrifice chip yield by throwing away a larger fraction of their faulty/slow chips, or *(ii)* sacrifice chip area by implementing bulky components in their chips for protective/enhancement purposes. Both these options, however, are self-defeating to a certain degree, because they raise the effective cost-per-bit of DRAM, which runs counter to the very rationale behind DRAM scaling: to lower the effective cost-per-bit. As a result, it is becoming more difficult for DRAM manufacturers to justify the billions of dollars in capital expenditure that is required for upgrading to the latest generation of DRAM process technology. While having served us so well in the past, the traditional approach of scaling DRAM — characterized primarily by breakthroughs and innovations in circuits/devices — is now showing telltale signs of declining effectiveness and diminishing returns.

1.2. This Thesis: A Higher-Level Approach to DRAM Scaling

In this thesis, our research objective is to answer the following question. *Can we tolerate the deteriorating physical characteristics of DRAM cells — especially with respect to reliability and performance — by developing mitigation techniques at a higher level of abstraction?* By doing so, we would be able to enjoy the traditional benefits of DRAM scaling, while we separately address its emerging drawbacks at the architectural/systems level.

As a matter of fact, such an approach has already been employed to great success in a different type of memory technology: NAND flash. Despite the fact that flash chips are highly unreliable and difficult to manage, their shortcomings are effectively hidden when they are used as part of a modern flash drive, which contains an intelligent flash controller that performs a wide variety of recovery and optimization duties to compensate for the weaknesses of flash. Instead of placing the burden of reliability and performance entirely on the flash chips, this approach allows some of it to be shifted onto the flash controller, thereby easing the path for further scaling.

DRAM chips too have their own controller, which is most often implemented as part of the processor itself. However, there are two distinguishing features of DRAM that restrict its controller from employing the same strategy to the extent of the flash controller. First, DRAM operates under a very constrained latency budget — tens of nanoseconds as opposed to tens of microseconds — which leaves little time for the DRAM controller to take any remedial action. Second, for compatibility reasons, the interface between the DRAM controller and the DRAM chips is not only rigidly standardized, but is done so in a minimalistic fashion that does not easily allow for a close cooperation between them. As a result, complexity-effectiveness and cost-consciousness are crucial factors to consider when taking an architectural approach to DRAM scaling. In this thesis, we seek out to demonstrate the following.

The reliability and performance of main memory can be improved by making low-overhead, non-intrusive modifications to how DRAM chips and controllers are designed.

In particular, we show the effectiveness of such an approach by identifying and tackling two critical problems that affect the reliability and performance of DRAM at advanced process technology generations. First, we expose a new type of failure that is found in only the most recently manufactured DRAM chips [84]. We then propose architectural solutions to prevent these failures, which pose a risk to not just memory reliability, but also to system security. Second, we highlight a critical component of the DRAM access latency that is expected to increase by more than 5x in the coming years, as projected by the DRAM industry [77]. We then propose a new, parallel DRAM architecture that overlaps the latency of multiple accesses and reduces the effective latency of DRAM [87]. Lastly, we present a new DRAM simulator that allows us to examine the strengths and weaknesses of different DRAM architectures quickly and easily [88].

1.2.1. Improving DRAM Reliability

As process technology scales down to smaller dimensions, DRAM becomes more vulnerable to *disturbance*, a phenomenon in which nearby cells interfere with each other's operation. For the first time in academic literature, we expose the widespread existence of *disturbance errors* in commodity DRAM chips that are sold and used today [84]. By reading repeatedly from the one cell, we show that it is possible to corrupt the data stored in adjacent cells.

After testing a large sample population of DRAM modules (the oldest of which dates back to 2008), we determine that the problem first arose in 2010 and that it still persists to this day. We found disturbance errors in modules from all three major manufacturers, as well as in all of their modules assembled between 2012–2013.

We demonstrate that disturbance errors are an actual hardware vulnerability affecting real systems. We construct a user-level kernel which induces many errors on general-purpose processors from Intel (Sandy Bridge, Ivy Bridge, Haswell) and AMD (Piledriver). With its ability to bypass memory protection (OS/VMM), the kernel can be deployed as a disturbance attack to corrupt the memory state of a system and its software.

We characterize the cause and symptoms of disturbance errors based on a large-scale study, involving 129 DRAM modules (972 DRAM chips) sampled from a time span of six years. We extensively test the modules using a custom-built FPGA infrastructure to determine the specific conditions under which the errors occur, as well as the specific manner in which they occur. From this, we build a comprehensive understanding of disturbance errors that serves as the foundation for developing a solution against them.

We examine a total of seven solutions that could be employed to prevent disturbance errors. We find that the most cost-efficient approaches are the ones that involve a collaborative effort between the DRAM chips and the DRAM controller. This is because only the DRAM chip knows which of its cells are physically adjacent to each other, whereas only the DRAM controller knows which cells have been accessed the most frequently. Both pieces of information are needed in order to identify the cells that are likely to be at risk from disturbance errors. According to our analysis, our solutions provide a strong reliability guarantee even under the worst-case conditions.

Related Work. As a generalized class of DRAM faults, disturbance errors are not new. They can occur whenever there is a strong enough interaction between two circuit components that were meant to be isolated from each other. In fact, disturbance errors are known to have manifested in the very first DRAM chips [115, 134]. Nevertheless, DRAM manufacturers in the past have achieved success in containing the errors, and have prevented them from being released into the wild. They were able to do so by improving circuit-level isolation [36, 59, 110, 146, 166] and subsequently screening for the errors after the chips have been fabricated [7, 8, 153]. Despite their efforts, however, this thesis demonstrates that a new type of disturbance error is now plaguing DRAM chips from all three major manufacturers of DRAM. One of the main reasons why this error has escaped early detection is because it requires a very large number of accesses (>100K) to the same address before it occurs. This is unlike disturbance errors from the past, which had manifested only after a much smaller number of accesses. In addition, the fact that this error is being experienced by all three manufacturers — and also at a similar timeframe — points to a more fundamental issue in DRAM scaling as the cause, rather than it being an isolated incidence of failure in quality control or circuit design. We provide more detail on related work in Chapter 2.

1.2.2. Improving DRAM Performance

To be able to serve multiple memory accesses in parallel, DRAM chips consist of multiple *banks* which can be accessed independently of each other. Nevertheless, if two memory accesses converge on the same bank, they must be served one after the other, experiencing what is referred to as a *bank conflict*. In the worst case, bank conflicts could delay a memory request by thousands of nanoseconds.

Bank conflicts cause three specific problems that degrade the performance of main memory. First, bank conflicts serialize accesses that could potentially have been served in parallel. Second, bank conflicts are likely to induce thrashing the *row-buffer*, which is a small cache that is included in every bank. Third, an access that follows a write access to the same bank experiences an extra delay called the *write-recovery penalty*. Critically, this penalty is expected to increase by more than 5x in the near future as a result of worsening variation in process technology [77].

A naive solution to bank conflicts would be to increase the number of banks in each DRAM chip. However, this is expensive because it wastes a large amount of chip area for implementing the extra support structures that are required by every new bank.

Instead, as a cost-effective solution to this problem, we propose to extract an additional degree of parallelism from DRAM banks called *subarray parallelism* [87]. This is based on two key observations that we make about modern DRAM architecture. First, a DRAM bank is physically implemented as a collection of smaller tiles (i.e., *subarrays*) — similar to how a city consists of urban blocks. Second, subarrays are mostly independent from each other except when being operated by the support structures which are globally shared by all subarrays within a bank. We then devise a set of small, non-intrusive changes to DRAM architecture that increases the autonomy of each subarray. Some of these changes transform a global structure to a local structure at each subarray, while other changes allow a subarray to relinquish a global structure more quickly, so that it can be used by another subarray. By enabling subarray parallelism at the DRAM chips and exploiting it at the DRAM controller, we achieve large performance improvements at low cost.

Related Work. While there have been many prior approaches for modifying the organization of DRAM to improve its performance, most of them are costly to implement because they increase the chip size by a significant amount. For example, Fujitsu [136] and Micron [80] have proposed latency-optimized DRAM chips which include a larger number of row-buffers that allow electrical charge to be driven more quickly into or out of the cells. Unfortunately, this increases the chip size by 30-80% [136, 80]. On the other hand, many in the past have proposed to augment DRAM chips with an additional SRAM cache for storing the most recently accessed pieces of data [34, 44, 46, 49, 79, 121, 135, 159, 170]. But this approach also incurs a large chip size overhead — according to our analysis, 5.0% for just 64Kbits of SRAM. We provide more detail on related work in Chapter 3.

1.2.3. Facilitating DRAM Research

Architectural design space exploration of DRAM requires a powerful and expressive DRAM simulator. However, existing DRAM simulators are slow and inflexible, primarily due to the fact that they are verbosely hardcoded for one specific DRAM system (e.g., DDR3). As an alternative, we develop a new simulator, called *Ramulator*, to aid us in our own research [88]. In brief, Ramulator is a fast and cycle-accurate DRAM simulator that is built from the ground up for extensibility.

Ramulator is based on our observation that a DRAM system can be modeled as a collection of state-machines, whose state transitions are triggered by a main memory event, such as a DRAM command being issued by the DRAM controller. By leveraging this observation, Ramulator defines a DRAM system by the number of its state-machines, how they behave, and how they interact with one another. All of this information is then consolidated into just a small number of lookup tables that become associated with that particular DRAM system. In Ramulator, implementing a new DRAM system is easy as constructing a new set of lookup tables, and populating their entries in a disciplined and localized manner. Subsequently, Ramulator uses these lookup tables to instantiate as many state-machines in however different configurations as is needed. Thanks to the modularity of the lookup tables, Ramulator provides support for the largest number of contemporary DRAM standards — DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, etc. — compared to previous simulators.

Ramulator is also optimized to accelerate simulation. For example, when none of the state-machines are yet ready to make a state transition, Ramulator fast-forwards the simulation into the future. In addition, Ramulator precomputes and memoizes important information about the state-machines that are frequently accessed. As a result, Ramulator is able to outperform the next fastest DRAM simulator by a factor of 2.5x.

Related Work. In the domain of CPU simulation, there are many simulators that have been developed to allow other researchers to implement their ideas easily (e.g., [16, 18]). However, there has been considerably less development in the domain of DRAM simulation. We provide more detail on related work in Chapter 4.

1.3. Thesis Contributions

This thesis makes the following contributions.

1. We show that DRAM scaling is negatively affecting reliability. We do so by exposing a new type of failure — called disturbance errors — that manifests in only the most recently manufactured DRAM chips. We demonstrate the errors on real systems to prove that they could potentially be exploited as a security vulnerability. We build a deep understanding of the errors by extensively characterizing their behavior under a wide variety of experimental conditions. We then propose a cost-effective solution at the architectural level that can be employed to prevent the errors with strong reliability guarantees.

2. We propose a highly parallel DRAM architecture that mitigates the large latency of DRAM access. We do so by clarifying many aspects of DRAM microarchitecture that were not well understood by computer architects, and identifying what are called subarrays as the smallest unit of DRAM internal organization. Although they exist in abundance, we observe that the subarrays cannot be accessed at the same time due to some structural bottlenecks that serialize their operation. We then propose a set of small circuit modifications that alleviates the bottlenecks and allows the subarrays to operate more independently. We show that such subarray parallelism provides large performance gains at low cost.

3. We develop a fast and extensible DRAM simulator for facilitating main memory research. Due to the modularity of its design, Ramulator can easily be modified to model new and experimental DRAM architectures. Compared to existing simulators, Ramulator provides not only the most comprehensive support for contemporary DRAM standards, but also the highest simulation speed.

1.4. Outline

This thesis is organized into five chapters. Chapter 2 discusses disturbance errors in DRAM. Chapter 3 discusses subarray parallelism. Chapter 4 discusses Ramulator. Chapter 5 concludes this thesis.

Chapter 2

Disturbance Errors: A New Reliability Problem in DRAM

The continued scaling of DRAM process technology has enabled smaller cells to be placed closer to each other. Cramming more DRAM cells into the same area has the well-known advantage of reducing the cost-per-bit of memory. Increasing the cell density, however, also has a negative impact on memory reliability due to three reasons. First, a small cell can hold only a limited amount of charge, which reduces its noise margin and renders it more vulnerable to data loss [21, 101, 165]. Second, the close proximity of cells introduces electromagnetic coupling effects between them, causing them to interact with each other in undesirable ways [21, 91, 101, 129]. Third, higher variation in process technology increases the number of outlier cells that are exceptionally susceptible to inter-cell crosstalk, exacerbating the two effects described above.

As a result, high-density DRAM is more likely to suffer from *disturbance*, a phenomenon in which different cells interfere with each other's operation. If a cell is disturbed beyond its noise margin, it malfunctions and experiences a *disturbance error*. Historically, DRAM manufacturers have been aware of disturbance errors since as early as the Intel 1103, the first commercialized DRAM chip [115, 134]. To mitigate disturbance errors, DRAM manufacturers have been employing a two-pronged approach: *(i)* improving inter-cell isolation through circuit-level techniques [36, 59, 110, 146, 166] and *(ii)* screening for disturbance errors during post-production testing [7, 8, 153]. In this chpater, we demonstrate that their efforts to contain disturbance errors have not always been successful, and that erroneous DRAM chips have been slipping into the field.[1]

2.1. Disturbance Errors in Today's DRAM Chips

In this chapter, we expose the existence and the widespread nature of disturbance errors in *commodity* DRAM chips sold and used today. Among 129 DRAM modules we analyzed (comprising 972 DRAM chips), we discovered disturbance errors in 110 modules (836 chips). In particular, *all* modules manufactured in the past two years (2012 and 2013) were vulnerable, which implies that the appearance of disturbance errors in the field is a relatively recent phenomenon affecting more advanced generations of process technology. We show that it takes as few as 139K reads to a DRAM address (more generally, to a DRAM row) to induce a disturbance error. As a proof of concept, we construct a user-level program that continuously accesses DRAM by issuing many loads to the same address while flushing the cache-line in between. We demonstrate that such a program induces many disturbance errors when executed on Intel or AMD machines.

We identify the root cause of DRAM disturbance errors as voltage fluctuations on an internal wire called the *wordline*. DRAM comprises a two-dimensional array of cells, where each *row* of cells has its own wordline. To access a cell within a particular row, the row's wordline must be enabled by raising its voltage — i.e., the row must be *activated*. When there are many activations to the same row, they force the wordline to toggle on and off repeatedly. According to our observations, such voltage fluctuations on a row's wordline have a disturbance effect on nearby rows, inducing some of their cells to leak charge at an

[1]The industry has been aware of this problem since at least 2012, which is when a number of patent applications were filed by Intel regarding the problem of "row hammer" [13, 11, 12, 10, 40, 39].

accelerated rate. If such a cell loses too much charge before it is restored to its original value (i.e., *refreshed*), it experiences a disturbance error.

We comprehensively characterize DRAM disturbance errors on an FPGA-based testing platform to understand their behavior and symptoms. Based on our findings, we examine a number of potential solutions (e.g., error-correction and frequent refreshes), which all have some limitations. We propose an effective and low-overhead solution, called *PARA*, that prevents disturbance errors by probabilistically refreshing only those rows that are likely to be at risk. In contrast to other solutions, PARA does not require expensive hardware structures or incur large performance penalties. This chapter makes the following contributions.

- To our knowledge, we are the first to expose the widespread existence of disturbance errors in commodity DRAM chips from recent years.
- We construct a user-level program that induces disturbance errors on real systems (Intel/AMD). Simply by reading from DRAM, we show that such a program could potentially breach memory protection and corrupt data stored in pages that it should not be allowed to access.
- We provide an extensive characterization of DRAM disturbance errors using an FPGA-based testing platform and 129 DRAM modules. We identify the root cause of disturbance errors as the repeated toggling of a row's wordline. We observe that the resulting voltage fluctuation could disturb cells in nearby rows, inducing them to lose charge at an accelerated rate. Among our key findings, we show that *(i)* disturbable cells exist in 110 out of 129 modules, *(ii)* up to one in 1.7K cells is disturbable, and *(iii)* toggling the wordline as few as 139K times causes a disturbance error.
- After examining a number of possible solutions, we propose PARA (*probabilistic adjacent row activation*), a low-overhead way of preventing disturbance errors. Every

time a wordline is toggled, PARA refreshes the nearby rows with a very small probability ($p \ll 1$). As a wordline is toggled many times, the increasing disturbance effects are offset by the higher likelihood of refreshing the nearby rows.

2.2. DRAM Background

In this section, we provide the necessary background on DRAM organization and operation to understand the cause and symptoms of disturbance errors.

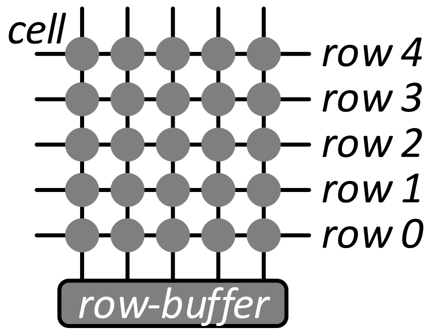
2.2.1. High-Level Organization

DRAM chips are manufactured in a variety of configurations [68], currently ranging in capacities of 1–8 Gbit and in data-bus widths of 4–16 pins. (A particular capacity does not imply a particular data-bus width.) By itself, an individual DRAM chip has only a small capacity and a narrow data-bus. That is why multiple DRAM chips are commonly ganged together to provide a large capacity and a wide data-bus (typically 64-bit). Such a “gang” of DRAM chips is referred to as a DRAM *rank*. One or more ranks are soldered onto a circuit board to form a DRAM *module*.

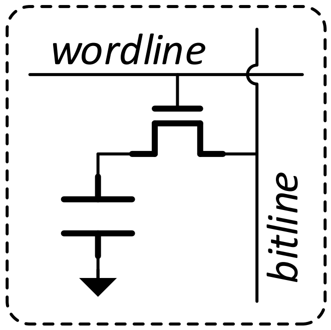
2.2.2. Low-Level Organization

As Figure 2.1a shows, DRAM comprises a two-dimensional array of DRAM *cells*, each of which consists of a *capacitor* and an *access-transistor*. Depending on whether its capacitor is fully charged or fully discharged, a cell is in either the *charged state* or the *discharged state*, respectively. These two states are used to represent a binary data value.

As Figure 2.1b shows, every cell lies at the intersection of two perpendicular wires: a horizontal *wordline* and a vertical *bitline*. A wordline connects to all cells in the horizontal direction (*row*) and a bitline connects to all cells in the vertical direction (*column*). When a row’s wordline is *raised* to a high voltage, it enables all of the access-transistors within the row, which in turn connects all of the capacitors to their respective bitlines. This allows



(a) Rows of cells



(b) A single cell

Figure 2.1. DRAM consists of cells

the row's data (in the form of charge) to be transferred into the *row-buffer* shown in Figure 2.1a. Better known as *sense-amplifiers*, the row-buffer reads out the charge from the cells — a process that destroys the data in the cells — and immediately writes the charge back into the cells [80, 87, 96]. Subsequently, all accesses to the row are served by the row-buffer on behalf of the row. When there are no more accesses to the row, the wordline is *lowered* to a low voltage, disconnecting the capacitors from the bitlines. A group of rows is called a *bank*, each of which has its own dedicated row-buffer. (The organization of a bank is similar to what was shown in Figure 2.1a.) Finally, multiple banks come together to form a rank. For example, Figure 2.2 shows a 2GB rank whose 256K rows are vertically partitioned into eight banks of 32K rows, where each row is 8KB (=64Kb) in size [68]. Having multiple banks increases parallelism because accesses to different banks can be served concurrently.

2.2.3. Accessing DRAM

An access to a rank occurs in three steps: *(i)* "opening" the desired row within a desired bank, *(ii)* accessing the desired columns from the row-buffer, and *(iii)* "closing" the row.

1. Open Row. A row is opened by raising its wordline. This connects the row to the bitlines, transferring all of its data into the bank's row-buffer.

2. Read/Write Columns. The row-buffer's data is accessed by reading or writing any of its

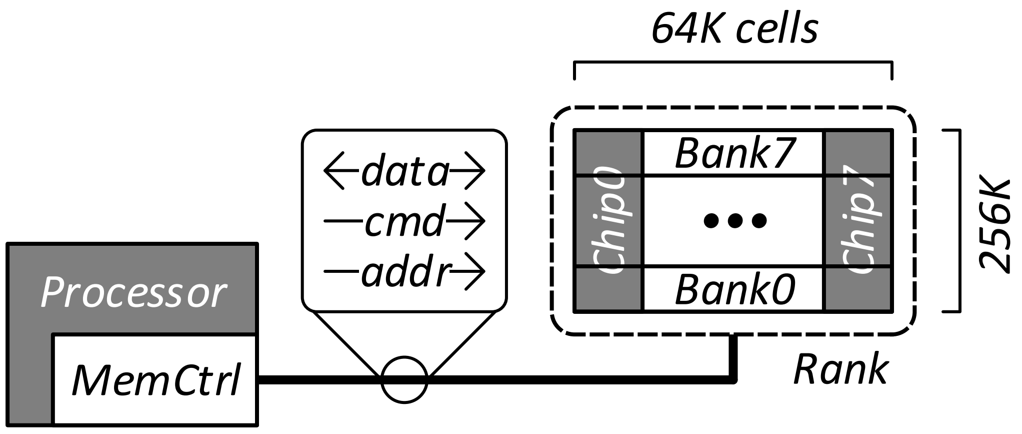


Figure 2.2. Memory controller, buses, rank, and banks

columns as needed.

3. Close Row. Before a different row in the same bank can be opened, the original row must be closed by lowering its wordline. In addition, the row-buffer is cleared.

The *memory controller*, which typically resides in the processor (Figure 2.2), guides the rank through the three steps by issuing *commands* and *addresses* as summarized in Table 2.1. After a rank accepts a command, some amount of delay is required before it becomes ready to accept another command. This delay is referred to as a DRAM *timing constraint* [68]. For example, the timing constraint defined between a pair of `ACTIVATE`s to the same row (in the same bank) is referred to as t_{RC} (row cycle time), whose typical value is ∼50 *nanoseconds* [68]. When trying to open and close the same row as quickly as possible, t_{RC} becomes the bottleneck — limiting the maximum rate to once every t_{RC}.

Operation	*Command*	*Address(es)*
1. Open Row	`ACTIVATE (ACT)`	Bank, Row
2. Read/Write Column	`READ/WRITE`	Bank, Column
3. Close Row	`PRECHARGE (PRE)`	Bank
Refresh (Section 2.2.4)	`REFRESH (REF)`	—

Table 2.1. DRAM commands and addresses [68]

2.2.4. Refreshing DRAM

The charge stored in a DRAM cell is not persistent. This is due to various leakage mechanisms by which charge can disperse: e.g., subthreshold leakage [132] and gate-induced drain leakage [133]. Eventually, the cell's charge-level would deviate beyond the noise margin, causing it to lose data — in other words, a cell has only a limited *retention time*. Before this time expires, the cell's charge must be restored (i.e., *refreshed*) to its original value: fully charged or fully discharged. The DDR3 DRAM specifications [68] guarantee a retention time of at least 64 *milliseconds*, meaning that all cells within a rank need to be refreshed at least once during this time window. Refreshing a cell can be accomplished by opening the row to which the cell belongs. Not only does the row-buffer read the cell's altered charge value but, at the same time, it restores the charge to full value (Section 2.2.2). In fact, *refreshing a row and opening a row are identical operations* from a circuits perspective. Therefore, one possible way for the memory controller to refresh a rank is to issue an `ACT` command to every row in succession. In practice, there exists a separate `REF` command which refreshes many rows at a time (Table 2.1). When a rank receives a `REF`, it automatically refreshes several of its least-recently-refreshed rows by internally generating `ACT` and `PRE` pairs to them. Within any given 64*ms* time window, the memory controller issues a sufficient number of `REF` commands to ensure that every row is refreshed exactly once. For a DDR3 DRAM rank, the memory controller issues 8192 `REF` commands during 64*ms*, once every 7.8*us* (=64*ms*/8192) [68].

2.3. Mechanics of Disturbance Errors

In general, disturbance errors occur whenever there is a strong enough interaction between two circuit components (e.g., capacitors, transistors, wires) that should be isolated from each other. Depending on which component interacts with which other component and also how they interact, many different modes of disturbance are possible.

Among them, we identify one particular disturbance mode that afflicts commodity DRAM chips from all three major manufacturers. *When a wordline's voltage is toggled repeatedly, some cells in nearby rows leak charge at a much faster rate.* Such cells cannot retain charge for even 64*ms*, the time interval at which they are refreshed. Ultimately, this leads to the cells losing data and experiencing disturbance errors.

Without analyzing DRAM chips at the device-level, we cannot make definitive claims about how a wordline interacts with nearby cells to increase their leakiness. We hypothesize, based on past studies and findings, that there may be three ways of interaction.[2] First, changing the voltage of a wordline could inject noise into an adjacent wordline through *electromagnetic coupling* [25, 110, 129]. This partially enables the adjacent row of access-transistors for a short amount of time and facilitates the leakage of charge. Second, *bridges* are a well-known class of DRAM faults in which conductive channels are formed between unrelated wires and/or capacitors [7, 8]. One study on embedded DRAM (eDRAM) found that toggling a wordline could accelerate the flow of charge between two bridged cells [50]. Third, it has been reported that toggling a wordline for hundreds of hours can permanently damage it by *hot-carrier injection* [29]. If some of the hot-carriers are injected into the neighboring rows, this could modify the amount of charge in their cells or alter the characteristic of their access-transistors to increase their leakiness.

Disturbance errors occur only when the cumulative interference effects of a wordline become strong enough to disrupt the state of nearby cells. In the next section, we demonstrate a small piece of software that achieves this by continuously reading from the same row in DRAM.

[2]At least one major DRAM manufacturer has confirmed these hypotheses as potential causes of disturbance errors.

2.4. Real System Demonstration

We induce DRAM disturbance errors on Intel (Sandy Bridge, Ivy Bridge, and Haswell) and AMD (Piledriver) systems using a 2GB DDR3 module. We do so by running Code 1a, which is a program that generates a read to DRAM on every data access. First, the two `mov` instructions read from DRAM at address `X` and `Y` and install the data into a register and also the cache. Second, the two `clflush` instructions evict the data that was just installed into the cache. Third, the `mfence` instruction ensures that the data is fully flushed before any subsequent memory instruction is executed.[3] Finally, the code jumps back to the first instruction for another iteration of reading from DRAM. (Note that Code 1a does not require elevated privileges to execute any of its instructions.)

```
code1a:
  mov (X), %eax
  mov (Y), %ebx
  clflush (X)
  clflush (Y)
  mfence
  jmp code1a
```

a. Induces errors

```
code1b:
  mov (X), %eax
  clflush (X)


  mfence
  jmp code1b
```

b. Does not induce errors

Code 1. Assembly code executed on Intel/AMD machines

On out-of-order processors, Code 1a generates multiple DRAM read requests, all of which queue up in the memory controller before they are sent out to DRAM: ($req_{\mathtt{X}}$, $req_{\mathtt{Y}}$, $req_{\mathtt{X}}$, $req_{\mathtt{Y}}$, $\cdots$). Importantly, we chose the values of `X` and `Y` so that they map to the *same* bank, but to *different* rows within the bank.[4] As we explained in Section 2.2.3, this forces

[3]Without the `mfence` instruction, there was a large number of hits in the processor's *fill-buffer* [56] as shown by hardware performance counters [57].

[4]Whereas AMD discloses *which* bits of the physical address are used and *how* they are used to compute the DRAM bank address [9], Intel does not. We partially reverse-engineered the addressing scheme for Intel processors using a technique similar to prior work [99, 145] and determined that setting `Y` to `X`+8M achieves

the memory controller to open and close the two rows repeatedly: ($\texttt{ACT}_\texttt{X}$, $\texttt{READ}_\texttt{X}$, $\texttt{PRE}_\texttt{X}$, $\texttt{ACT}_\texttt{Y}$, $\texttt{READ}_\texttt{Y}$, $\texttt{PRE}_\texttt{Y}$, $\cdots$). Using the address-pair (X, Y), we then executed Code 1a for millions of iterations. Subsequently, we repeated this procedure using many different address-pairs until every row in the 2GB module was opened/closed millions of times. In the end, we observed that Code 1a caused many bits to flip. For each processor, Table 2.2 reports the total number of bit-flips induced by Code 1a for two different initial states of the module: all '0's or all '1's.[5,6] Since Code 1a does not write any data into DRAM, we conclude that the bit-flips are the manifestation of disturbance errors. We will show later in Section 2.6.1 that this particular module — which we named A_{19} (Section 2.5) — yields *millions* of errors under certain testing conditions.

Bit-Flip	Sandy Bridge	Ivy Bridge	Haswell	Piledriver
'0' → '1'	7,992	10,273	11,404	47
'1' → '0'	8,125	10,449	11,467	12

Table 2.2. Bit-flips induced by disturbance on a 2GB module

As a control experiment, we also ran Code 1b which reads from only a single address. Code 1b did not induce any disturbance errors as we expected. For Code 1b, all of its reads are to the same row in DRAM: ($req_\texttt{X}$, $req_\texttt{X}$, $req_\texttt{X}$, $\cdots$). In this case, the memory controller minimizes the number of DRAM commands by opening and closing the row just *once*, while issuing many column reads in between: ($\texttt{ACT}_\texttt{X}$, $\texttt{READ}_\texttt{X}$, $\texttt{READ}_\texttt{X}$, $\texttt{READ}_\texttt{X}$, $\cdots$, $\texttt{PRE}_\texttt{X}$). As we explained in Section 2.3, DRAM disturbance errors are caused by the repeated opening/-closing of a row, *not* by column reads — which is precisely why Code 1b does *not* induce any errors.

our goal for all four processors. We ran Code 1a within a customized Memtest86+ environment [1] to bypass address translation.

[5]The faster a processor accesses DRAM, the more bit-flips it has. Expressed in the unit of accesses-per-second, the four processors access DRAM at the following rates: 11.6M, 11.7M, 12.3M, and 6.1M. (It is possible that not all accesses open/close a row.)

[6]We initialize the module by making the processor write out all '0's or all '1's to memory. But before this data is actually sent to the module, it is *scrambled* by the memory controller to avoid electrical resonance on the DRAM data-bus [57]. In other words, we do not know the exact "data" that is received by the module. We examine the significance of this in Section 2.6.4.

Disturbance errors violate two invariants that memory should provide: *(i)* a read access should *not* modify data at any address and *(ii)* a write access should modify data *only* at the address being written to. As long as a row is repeatedly opened, both read and write accesses can induce disturbance errors (Section 2.6.2), all of which occur in rows other than the one being accessed (Section 2.6.3). Since different DRAM rows are mapped (by the memory controller) to different software pages [75], Code 1a — just by accessing its own page — could corrupt pages belonging to other programs. Left unchecked, disturbance errors can be exploited by a malicious program to breach memory protection and compromise the system. With some engineering effort, we believe we can develop Code 1a into a *disturbance attack* that injects errors into other programs, crashes the system, or perhaps even hijacks control of the system. We leave such research for the future since the primary objective in this thesis is to understand and prevent DRAM disturbance errors.

2.5. Experimental Methodology

To develop an understanding of disturbance errors, we characterize 129 DRAM modules on an FPGA-based testing platform. Our testing platform grants us precise control over how and when DRAM is accessed on a cycle-by-cycle basis. Also, it does *not* scramble the data it writes to DRAM.[6]

Testing Platform. We programmed eight Xilinx FPGA boards [162] with a DDR3-800 DRAM memory controller [163], a PCIe 2.0 core [161], and a customized test engine. After equipping each FPGA board with a DRAM module, we connected them to two host computers using PCIe extender cables. We then enclosed the FPGA boards inside a heat chamber along with a thermocouple and a heater that are connected to an external temperature controller. Unless otherwise specified, all tests were run at 50±2.0°C (ambient).

Tests. We define a *test* as a sequence of DRAM accesses specifically designed to induce disturbance errors in a module. Most of our tests are derived from two snippets of pseudocode listed above (Code 2): TestBulk and TestEach. The goal of TestBulk is to quickly identify the union of all cells that were disturbed after toggling every row many times. On the other hand, TestEach identifies which specific cells are disturbed when each row is toggled many times. Both tests take three input parameters: *AI* (activation interval), *RI* (refresh interval), and *DP* (data pattern). First, AI determines how frequently a row is toggled — i.e., the time it takes to execute one iteration of the inner for-loop. Second, RI determines how frequently the module is refreshed during the test. Third, DP determines the initial data values with which the module is populated before errors are induced. TestBulk (Code 2a) starts by writing DP to the entire module. It then toggles a row at the rate of AI for the full duration of RI — i.e., the row is toggled $N = (2 \times \mathit{RI})/\mathit{AI}$ times.[7] This procedure is then repeated for every row in the module. Finally, TestBulk reads out the entire module and identifies all of the disturbed cells. TestEach (Code 2b) is similar except that lines 6, 12, and 13 are moved inside the outer for-loop. After toggling just one row, TestEach reads out the module and identifies the cells that were disturbed by the row.

[7]Refresh intervals for different rows are not aligned with each other (Section 2.2.4). Therefore, we toggle a row for *twice* the duration of *RI* to ensure that we fully overlap with at least one refresh interval for the row.

```
TestBulk($AI, RI, DP$)
  setAI($AI$)
  setRI($RI$)
  $N \leftarrow (2 \times RI)/AI$

  writeAll($DP$)
  for $r \leftarrow 0 \cdots ROW_{MAX}$
    for $i \leftarrow 0 \cdots N$
      ACT $r^{th}$ row
      READ $0^{th}$ col.
      PRE $r^{th}$ row
  readAll()
  findErrors()
```

a. Test all rows at once

```
TestEach($AI, RI, DP$)
  setAI($AI$)
  setRI($RI$)
  $N \leftarrow (2 \times RI)/AI$

  for $r \leftarrow 0 \cdots ROW_{MAX}$
    writeAll($DP$)
    for $i \leftarrow 0 \cdots N$
      ACT $r^{th}$ row
      READ $0^{th}$ col.
      PRE $r^{th}$ row
    readAll()
    findErrors()
```

b. Test one row at a time

Code 2. Two types of tests synthesized on the FPGA

Test Parameters. In most of our tests, we set AI=55*ns* and RI=64*ms*, for which the corresponding value of N is 2.33×10^6. We chose 55*ns* for AI since it approaches the maximum rate of toggling a row without violating the t_{RC} timing constraint (Section 2.2.3). In some tests, we also sweep AI up to 500*ns*. We chose 64*ms* for RI since it is the default refresh interval specified by the DDR3 DRAM standard (Section 2.2.4). In some tests, we also sweep RI down to 10*ms* and up to 128*ms*. For DP, we primarily use two data patterns [154]: *RowStripe* (even/odd rows populated with '0's/'1's) and its inverse $\sim$*RowStripe*. As Section 2.6.4 will show, these two data patterns induce the most errors. In some tests, we also use *Solid*, *ColStripe*, *Checkered*, as well as their inverses [154].

DRAM Modules. As listed in Tables 2.3, 2.4, and 2.5, we tested for disturbance errors in a total of 129 DDR3 DRAM modules. They comprise 972 DRAM chips from three manufacturers whose names have been anonymized to A, B, and C.[8] The three manufacturers represent a large share of the global DRAM market [32]. We use the following notation to reference the modules: M_i^{yyww} (M for the manufacturer, i for the numerical identifier, and *yyww* for the manufacture date in year and week).[9] Some of the modules are indistinguishable from each other in terms of the manufacturer, manufacture date, and chip type (e.g., $\mathrm{A}_{3\text{-}5}$). We collectively refer to such a group of modules as a *family*. For multi-rank modules, only the first rank is reflected in Tables 2.3, 2.4, and 2.5, which is also the only rank that we test. We will use the terms module and rank interchangeably.

[8]We tried to avoid third-party modules since they sometimes obfuscate the modules, making it difficult to determine the actual chip manufacturer or the exact manufacture date. Modules $\mathrm{B}_{14\text{-}31}$ are engineering samples.

[9]Manufacturers do not explicitly provide the technology node of the chips. Instead, we interpret recent manufacture dates and higher die versions as rough indications of more advanced process technology.

Manufacturer	*Module*	*Date**	*Timing†*		*Organization*		*Chip*			*Victims-per-Module*			*RI_{th} (ms)*
		(yy-ww)	*Freq (MT/s)*	*t_{RC} (ns)*	*Size (GB)*	*Chips*	*Size (Gb)‡*	*Pins*	*Die§*	*Average*	*Minimum*	*Maximum*	*Min*
A Total of 43 Modules	A_1	10-08	1066	50.625	0.5	4	1	×16	$\mathcal{B}$	0	0	0	–
	A_2	10-20	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	$A_{3\text{-}5}$	10-20	1066	50.625	0.5	4	1	×16	$\mathcal{B}$	0	0	0	–
	$A_{6\text{-}7}$	11-24	1066	49.125	1	4	2	×16	$\mathcal{D}$	7.8×10^1	5.2×10^1	1.0×10^2	21.3
	$A_{8\text{-}12}$	11-26	1066	49.125	1	4	2	×16	$\mathcal{D}$	2.4×10^2	5.4×10^1	4.4×10^2	16.4
	$A_{13\text{-}14}$	11-50	1066	49.125	1	4	2	×16	$\mathcal{D}$	8.8×10^1	1.7×10^1	1.6×10^2	26.2
	$A_{15\text{-}16}$	12-22	1600	50.625	1	4	2	×16	$\mathcal{D}$	9.5	9	1.0×10^1	34.4
	$A_{17\text{-}18}$	12-26	1600	49.125	2	8	2	×8	$\mathcal{M}$	1.2×10^2	3.7×10^1	2.0×10^2	21.3
	$A_{19\text{-}30}$	12-40	1600	48.125	2	8	2	×8	$\mathcal{K}$	8.6×10^6	7.0×10^6	$\mathbf{1.0 \times 10^7}$	**8.2**
	$A_{31\text{-}34}$	13-02	1600	48.125	2	8	2	×8	–	1.8×10^6	1.0×10^6	3.5×10^6	11.5
	$A_{35\text{-}36}$	13-14	1600	48.125	2	8	2	×8	–	4.0×10^1	1.9×10^1	6.1×10^1	21.3
	$A_{37\text{-}38}$	13-20	1600	48.125	2	8	2	×8	$\mathcal{K}$	1.7×10^6	1.4×10^6	2.0×10^6	9.8
	$A_{39\text{-}40}$	13-28	1600	48.125	2	8	2	×8	$\mathcal{K}$	5.7×10^4	5.4×10^4	6.0×10^4	16.4
	A_{41}	14-04	1600	49.125	2	8	2	×8	–	2.7×10^5	2.7×10^5	2.7×10^5	18.0
	$A_{42\text{-}43}$	14-04	1600	48.125	2	8	2	×8	$\mathcal{K}$	0.5	0	1	62.3

∗ We report the manufacture date marked on the chip packages, which is more accurate than other dates that can be gleaned from a module.
† We report timing constraints stored in the module's on-board ROM [71], which is read by the system BIOS to calibrate the memory controller.
‡ The maximum DRAM chip size supported by our testing platform is 2Gb.
§ We report DRAM die versions marked on the chip packages, which typically progress in the following manner: $\mathcal{M} \rightarrow \mathcal{A} \rightarrow \mathcal{B} \rightarrow \mathcal{C} \rightarrow \cdots$.

Table 2.3. DDR3 DRAM modules from A manufacturer (43 out of 129) sorted by manufacture date

Manufacturer	*Module*	*Date**	*Timing†*		*Organization*		*Chip*			*Victims-per-Module*			*RI_{th} (ms)*
		(yy-ww)	*Freq (MT/s)*	*t_{RC} (ns)*	*Size (GB)*	*Chips*	*Size (Gb)‡*	*Pins*	*Die§*	*Average*	*Minimum*	*Maximum*	*Min*
	B_1	08-49	1066	50.625	1	8	1	×8	$\mathcal{D}$	0	0	0	–
	B_2	09-49	1066	50.625	1	8	1	×8	$\mathcal{E}$	0	0	0	–
	B_3	10-19	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	B_4	10-31	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_5	11-13	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_6	11-16	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
	B_7	11-19	1066	50.625	1	8	1	×8	$\mathcal{F}$	0	0	0	–
B	B_8	11-25	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
	B_9	11-37	1333	49.125	2	8	2	×8	$\mathcal{D}$	1.9×10^6	1.9×10^6	1.9×10^6	11.5
Total of	$B_{10\text{-}12}$	11-46	1333	49.125	2	8	2	×8	$\mathcal{D}$	2.2×10^6	1.5×10^6	$\mathbf{2.7 \times 10^6}$	11.5
54	B_{13}	11-49	1333	49.125	2	8	2	×8	$\mathcal{C}$	0	0	0	–
Modules	B_{14}	12-01	1866	47.125	2	8	2	×8	$\mathcal{D}$	9.1×10^5	9.1×10^5	9.1×10^5	**9.8**
	$B_{15\text{-}31}$	12-10	1866	47.125	2	8	2	×8	$\mathcal{D}$	9.8×10^5	7.8×10^5	1.2×10^6	11.5
	B_{32}	12-25	1600	48.125	2	8	2	×8	$\mathcal{E}$	7.4×10^5	7.4×10^5	7.4×10^5	11.5
	$B_{33\text{-}42}$	12-28	1600	48.125	2	8	2	×8	$\mathcal{E}$	5.2×10^5	1.9×10^5	7.3×10^5	11.5
	$B_{43\text{-}47}$	12-31	1600	48.125	2	8	2	×8	$\mathcal{E}$	4.0×10^5	2.9×10^5	5.5×10^5	13.1
	$B_{48\text{-}51}$	13-19	1600	48.125	2	8	2	×8	$\mathcal{E}$	1.1×10^5	7.4×10^4	1.4×10^5	14.7
	$B_{52\text{-}53}$	13-40	1333	49.125	2	8	2	×8	$\mathcal{D}$	2.6×10^4	2.3×10^4	2.9×10^4	21.3
	B_{54}	14-07	1333	49.125	2	8	2	×8	$\mathcal{D}$	7.5×10^3	7.5×10^3	7.5×10^3	26.2

Table 2.4. DDR3 DRAM modules from B manufacturer (54 out of 129) sorted by manufacture date

Manufacturer	*Module*	*Date*[*]	*Timing*[†]		*Organization*		*Chip*			*Victims-per-Module*			*RI_{th} (ms)*
		(yy-ww)	*Freq (MT/s)*	*t_{RC} (ns)*	*Size (GB)*	*Chips*	*Size (Gb)*[‡]	*Pins*	*Die*[§]	*Average*	*Minimum*	*Maximum*	*Min*
C Total of 32 Modules	C_1	10-18	1333	49.125	2	8	2	×8	$\mathcal{A}$	0	0	0	–
	C_2	10-20	1066	50.625	2	8	2	×8	$\mathcal{A}$	0	0	0	–
	C_3	10-22	1066	50.625	2	8	2	×8	$\mathcal{A}$	0	0	0	–
	$C_{4\text{-}5}$	10-26	1333	49.125	2	8	2	×8	$\mathcal{B}$	8.9×10^2	6.0×10^2	1.2×10^3	29.5
	C_6	10-43	1333	49.125	1	8	1	×8	$\mathcal{T}$	0	0	0	–
	C_7	10-51	1333	49.125	2	8	2	×8	$\mathcal{B}$	4.0×10^2	4.0×10^2	4.0×10^2	29.5
	C_8	11-12	1333	46.25	2	8	2	×8	$\mathcal{B}$	6.9×10^2	6.9×10^2	6.9×10^2	21.3
	C_9	11-19	1333	46.25	2	8	2	×8	$\mathcal{B}$	9.2×10^2	9.2×10^2	9.2×10^2	27.9
	C_{10}	11-31	1333	49.125	2	8	2	×8	$\mathcal{B}$	3	3	3	39.3
	C_{11}	11-42	1333	49.125	2	8	2	×8	$\mathcal{B}$	1.6×10^2	1.6×10^2	1.6×10^2	39.3
	C_{12}	11-48	1600	48.125	2	8	2	×8	$\mathcal{C}$	7.1×10^4	7.1×10^4	7.1×10^4	19.7
	C_{13}	12-08	1333	49.125	2	8	2	×8	$\mathcal{C}$	3.9×10^4	3.9×10^4	3.9×10^4	21.3
	$C_{14\text{-}15}$	12-12	1333	49.125	2	8	2	×8	$\mathcal{C}$	3.7×10^4	2.1×10^4	5.4×10^4	21.3
	$C_{16\text{-}18}$	12-20	1600	48.125	2	8	2	×8	$\mathcal{C}$	3.5×10^3	1.2×10^3	7.0×10^3	27.9
	C_{19}	12-23	1600	48.125	2	8	2	×8	$\mathcal{E}$	1.4×10^5	1.4×10^5	1.4×10^5	18.0
	C_{20}	12-24	1600	48.125	2	8	2	×8	$\mathcal{C}$	6.5×10^4	6.5×10^4	6.5×10^4	21.3
	C_{21}	12-26	1600	48.125	2	8	2	×8	$\mathcal{C}$	2.3×10^4	2.3×10^4	2.3×10^4	24.6
	C_{22}	12-32	1600	48.125	2	8	2	×8	$\mathcal{C}$	1.7×10^4	1.7×10^4	1.7×10^4	22.9
	$C_{23\text{-}24}$	12-37	1600	48.125	2	8	2	×8	$\mathcal{C}$	2.3×10^4	1.1×10^4	3.4×10^4	18.0
	$C_{25\text{-}30}$	12-41	1600	48.125	2	8	2	×8	$\mathcal{C}$	2.0×10^4	1.1×10^4	3.2×10^4	19.7
	C_{31}	13-11	1600	48.125	2	8	2	×8	$\mathcal{C}$	3.3×10^5	3.3×10^5	$\mathbf{3.3 \times 10^5}$	**14.7**
	C_{32}	13-35	1600	48.125	2	8	2	×8	$\mathcal{C}$	3.7×10^4	3.7×10^4	3.7×10^4	21.3

Table 2.5. DDR3 DRAM modules from C manufacturer (32 out of 129) sorted by manufacture date

2.6. Characterization Results

We now present the results from our characterization study. Section 2.6.1 explains how the number of disturbance errors in a module varies greatly depending on its manufacturer and manufacture date. Section 2.6.2 confirms that repeatedly activating a row is indeed the source of disturbance errors. In addition, we also measure the minimum number of times a row must be activated before errors start to appear. Section 2.6.3 shows that the errors induced by such a row (i.e., the *aggressor row*) are predominantly localized to two other rows (i.e., the *victim rows*). We then provide arguments for why the victim rows are likely to be the immediate neighbors. Section 2.6.4 demonstrates that disturbance errors affect only the charged cells, causing them to lose data by becoming discharged.

2.6.1. Disturbance Errors are Widespread

For every module in Tables 2.3, 2.4, and 2.5, we tried to induce disturbance errors by subjecting them to two runs of TestBulk:

1. TestBulk(55*ns*, 64*ms*, RowStripe)
2. TestBulk(55*ns*, 64*ms*, $\sim$RowStripe)

If a cell experienced an error in either of the runs, we refer to it as a *victim cell* for that module. Interestingly, virtually no cell in any module had errors in both runs — meaning that the number of errors summed across the two runs is equal to the number of unique victims for a module.[10] (This is an important observation that will be examined further in Section 2.6.4.)

For each family of modules, three right columns in Tables 2.3, 2.4, and 2.5 report the avg/min/max number of victims among the modules belonging to the family. As shown in the table, we were able to induce errors in all but 19 modules, most of which are also

[10] In some of the B modules, there were some rare victim cells ($\leq$15) that had errors in both runs. We will revisit these cells in Section 2.6.3.

the oldest modules from each manufacturer. In fact, there exist date boundaries that separate the modules with errors from those without. For A, B, and C, their respective date boundaries are 2011-24, 2011-37, and 2010-26. Except for A_{42}, B_{13}, and C_6, every module manufactured on or after these dates exhibits errors. These date boundaries are likely to indicate process upgrades since they also coincide with die version upgrades. Using manufacturer B as an example, 2Gb$\times$8 chips before the boundary have a die version of $\mathcal{C}$, whereas the chips after the boundary (except B_{13}) have die versions of either $\mathcal{D}$ or $\mathcal{E}$. Therefore, we conclude that disturbance errors are a relatively recent phenomenon, affecting almost all modules manufactured within the past 3 years.

Using the data from Tables 2.3, 2.4, and 2.5, Figure 2.3 plots the normalized number of errors for each family of modules versus their manufacture date. The error bars denote the minimum and maximum for each family. From the figure, we see that modules from 2012 to 2013 are particularly vulnerable. For each manufacturer, the number of victims per 10^9 cells can reach up to 5.9×10^5, 1.5×10^5, and 1.9×10^4. Interestingly, Figure 2.3 reveals a jigsaw-like trend in which sudden jumps in the number of errors are followed by gradual descents. This may occur when a manufacturer migrates away from an old-but-reliable process to a new-but-unreliable process. By making adjustments over time, the new process may eventually again become reliable — which could explain why the most recent modules from manufacturer A ($A_{42\text{-}43}$) have little to no errors.

2.6.2. Access Pattern Dependence

So far, we have demonstrated disturbance errors by repeatedly opening, reading, and closing the same row. We express this access pattern using the following notation, where N is a large number: (*open–read–close*)^N. However, this is not the only access pattern to induce errors. Table 2.6 lists a total of four different access patterns, among which two induced errors on the modules that we tested: A_{23}, B_{11}, and C_{19}. These three modules were chosen because they had the most errors (A_{23} and B_{11}) or the second most errors

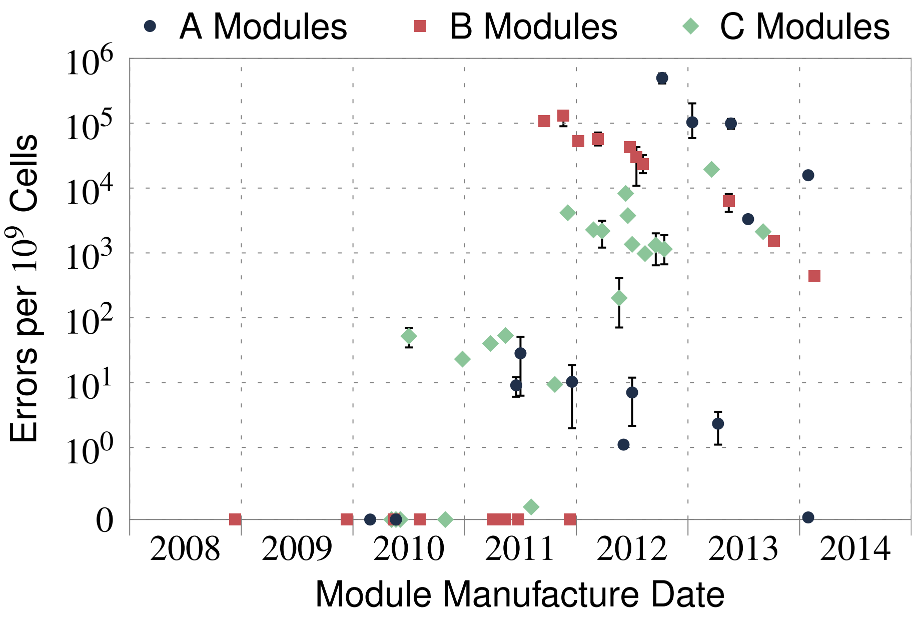


Figure 2.3. Normalized number of errors vs. manufacture date

(C_{19}) among all modules from the same manufacturer. What is in common between the first two access patterns is that they open and close the same row repeatedly. The other two, in contrast, do so just once and did not induce any errors. From this we conclude that the repeated toggling of the same wordline is indeed the cause of disturbance errors.[11]

Access Pattern	Disturbance Errors?
1. $(open\text{–}read\text{–}close)^N$	**Yes**
2. $(open\text{–}write\text{–}close)^N$	**Yes**
3. $open\text{–}read^N\text{–}close$	No
4. $open\text{–}write^N\text{–}close$	No

Table 2.6. Access patterns that induce disturbance errors

Refresh Interval (RI). As explained in Section 2.5, our tests open a row once every 55*ns*. For each row, we sustain this rate for the full duration of an RI (default: 64*ms*). This is so that the row can maximize its disturbance effect on other cells, causing them to leak the most charge before they are next refreshed. As the RI is varied between 10–128*ms*, Figure 2.4 plots the numbers of errors in the three modules. Due to time limitations, we

[11]For write accesses, a row cannot be opened and closed once every t_{RC} due to an extra timing constraint called t_{WR} (write recovery time) [68]. As a result, the second access pattern in Table 2.6 induces fewer errors.

tested only the first bank. For shorter RIs, there are fewer errors due to two reasons: *(i)* a victim cell has less time to leak charge between refreshes; *(ii)* a row is opened fewer times between those refreshes, diminishing the disturbance effect it has on the victim cells. At a sufficiently short RI — which we refer to as the *threshold refresh interval* (RI_{th}) — errors are completely eliminated not in just the first bank, but for the entire module. For each family of modules, the rightmost column in Tables 2.3, 2.4, and 2.5 reports the minimum RI_{th} among the modules belonging to the family. The family with the most victims at $RI = 64ms$ is also likely to have the lowest RI_{th}: 8.2*ms*, 9.8*ms*, and 14.7*ms*. This translates into $7.8\times$, $6.5\times$, and $4.3\times$ increase in the frequency of refreshes.

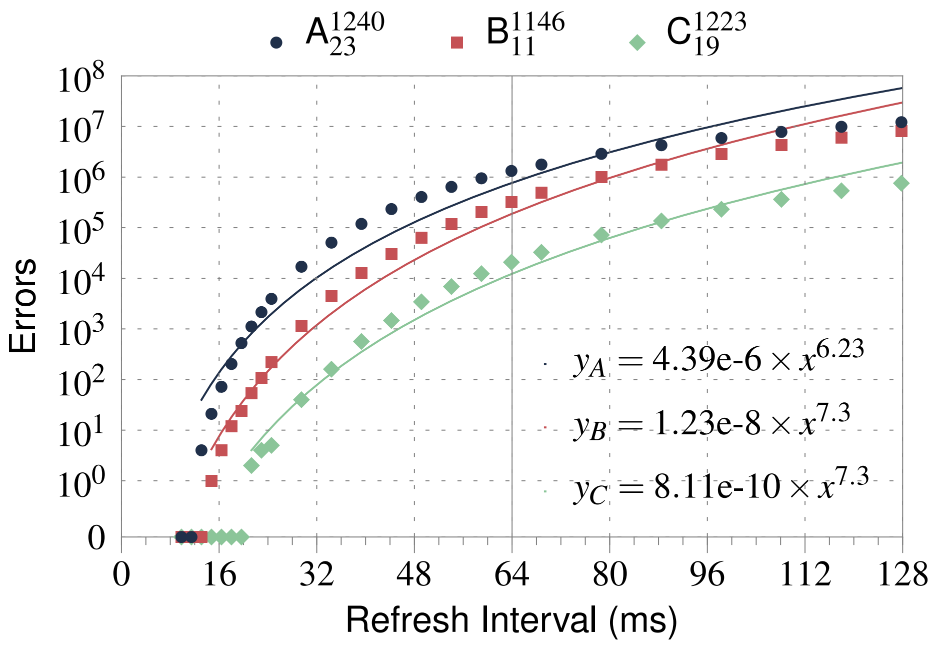


Figure 2.4. Number of errors as the refresh interval is varied

Activation Interval (AI). As the AI is varied between 55–500*ns*, Figure 2.5 plots the numbers of errors in the three modules. (Only the first bank is tested, and the RI is kept constant at 64*ms*.) For longer AIs, there are fewer errors because a row is opened less often, thereby diminishing its disturbance effect. When the AI is sufficiently long, the three modules have no errors: ~500*ns*, ~450*ns*, and ~250*ns*. At the shortest AIs, however, there is a notable reversal in the trend: B_{11} and C_{19} have fewer errors at 60*ns* than at 65*ns*. How can there be fewer errors when a row is opened more often? This anomaly can be explained only if the disturbance effect of opening a row is weaker at 60*ns* than at 65*ns*. In general, row-coupling effects are known to be weakened if the wordline voltage is not raised quickly while the row is being opened [129]. The wordline voltage, in turn, is raised by a circuit called the *wordline charge-pump* [80], which becomes sluggish if not given enough time to "recover" after performing its job.[12] When a wordline is raised every 60*ns*, we hypothesize that the charge-pump is unable to regain its full strength by the end of each interval, which leads to a slow voltage transition on the wordline and, ultimately, a weak disturbance effect. In contrast, an AI of 55*ns* appears to be immune to this phenomenon, since there is a large jump in the number of errors. We believe this to be an artifact of how our memory controller schedules refresh commands. At 55*ns*, our memory controller happens to run at 100% utilization, meaning that it always has a DRAM request queued in its buffer. In an attempt to minimize the latency of the request, the memory controller de-prioritizes a pending refresh command by $\sim$64*us*. This technique is fully compliant with the DDR3 DRAM standard [68] and is widely employed in general-purpose processors [57]. As a result, the effective refresh interval is slightly lengthened, which again increases the number of errors.

Number of Activations. We have seen that disturbance errors are heavily influenced by the lengths of RI and AI. In Figure 2.6, we compare their effects by superimposing the two previous figures on top of each other. Both figures have been normalized onto the same x-axis whose values correspond to the number of activations per refresh interval: RI/AI.[13] (Only the left-half is shown for Figure 2.4, where RI $\leq$ 64*ms*.) In Figure 2.6, the number of activations reaches a maximum of 1.14×10^6 (=64*ms*/55*ns*) when RI and AI are set to their default lengths. At this particular point, the numbers of errors between the

[12]The charge-pump "up-converts" the DRAM chip's supply voltage into an even higher voltage to ensure that the wordline's access-transistors are completely switched on. A charge-pump is essentially a large reservoir of charge which is slowly refilled after being tapped into.

[13]The actual formula we used is $(RI - 8192 \times t_{RFC})/AI$, where t_{RFC} (refresh cycle time) is the timing constraint between a `REF` and a subsequent `ACT` to the same module [68]. Our testing platform sets t_{RFC} to 160*ns*, which is a sufficient amount of time for all of our modules.

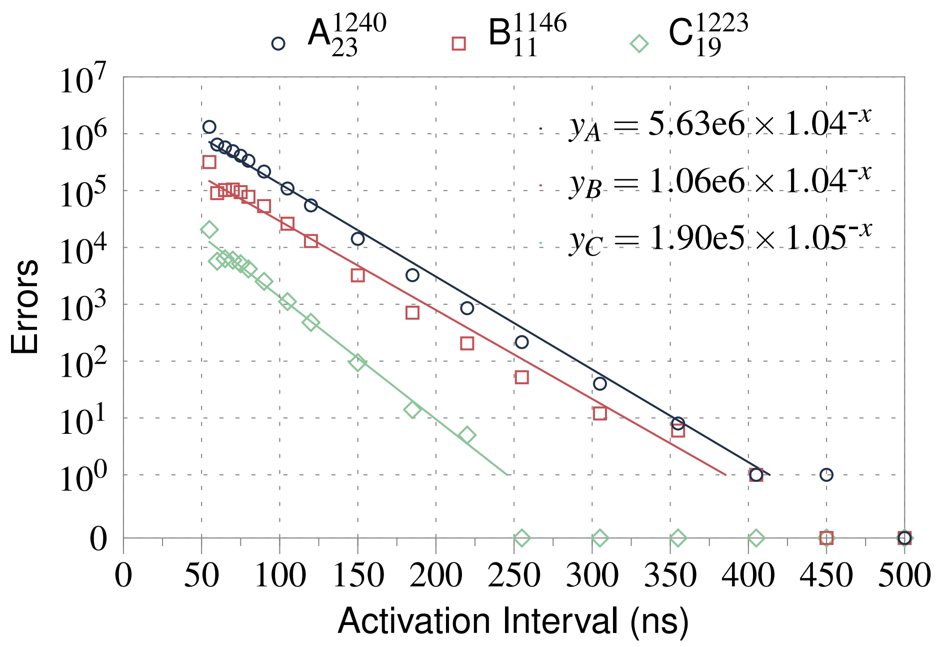


Figure 2.5. Number of errors as the activation interval is varied

two studies degenerate to the same value. It is clear from the figure that fewer activations induce fewer errors. For the same number of activations, having a long RI and a long AI is likely to induce more errors than having a short RI and a short AI. We define the *threshold number of activations* (N_{th}) as the minimum number of activations that is required to induce an error when RI=64*ms*. The three modules (for only their first banks) have the following values for N_{th}: 139K, 155K, and 284K.

2.6.3. Address Correlation: Aggressor & Victim

Most rows in A_{23}, B_{11}, and C_{19} have at least one cell that experienced an error: 100%, 99.96%, and 41.54%. We analyzed the addresses of such victim cells to determine whether they exhibit any spatial locality. We were unable to identify any distinct pattern or skew. By chance, however, some victim cells could still end up being located near each other. For the three modules, Table 2.7 shows how many 64-bit words in their full address-space (0–2GB) contain 1, 2, 3, or 4 victim cells. While most words have just a single victim, there are also some words with multiple victims. This has an important consequence

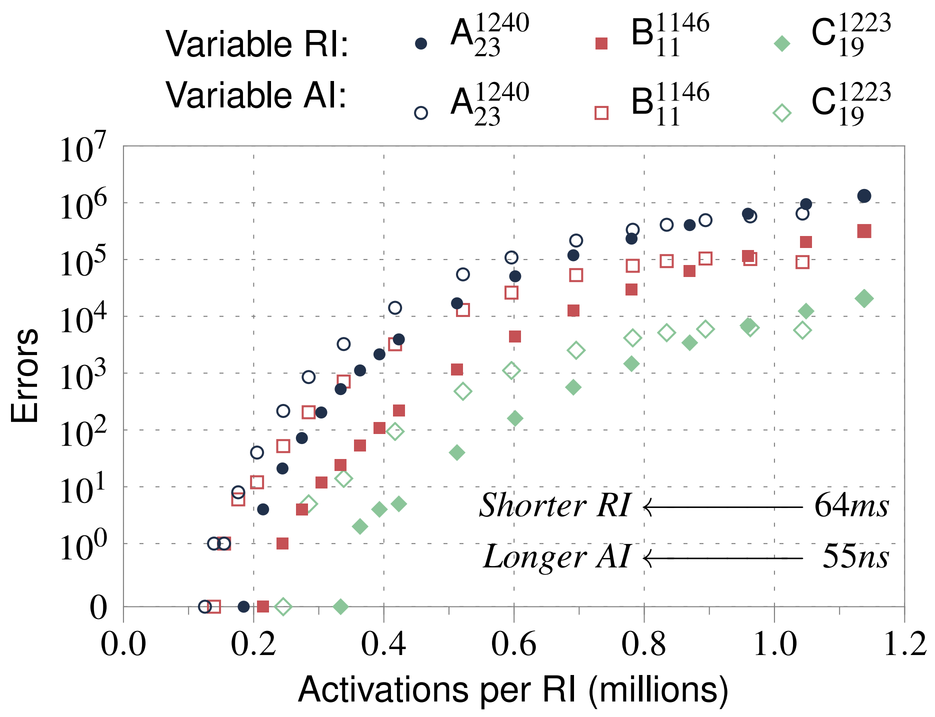


Figure 2.6. Number of errors vs. number of activations

for error-correction codes (ECC). For example, SECDED (single error-correction, double error-detection) can correct only a single-bit error within a 64-bit word. If a word contains two victims, however, SECDED cannot correct the resulting double-bit error. And for three or more victims, SECDED cannot even detect the multi-bit error, leading to silent data corruption. Therefore, we conclude that SECDED is not failsafe against disturbance errors.

Module	*Number of 64-bit words with X errors*			
	$X=1$	$X=2$	$X=3$	$X=4$
A_{23}	9,709,721	**181,856**	**2,248**	**18**
B_{11}	2,632,280	**13,638**	**47**	0
C_{19}	141,821	**42**	0	0

Table 2.7. Uncorrectable multi-bit errors (in bold)

Most rows in A_{23}, B_{11}, and C_{19} cause errors when they are repeatedly opened. We refer to such rows as *aggressor rows*. We exposed the aggressor rows in the modules by subjecting them to two runs of TestEach for only the first bank:

1. TestEach(55*ns*, 64*ms*, RowStripe)

2. TestEach(55*ns*, 64*ms*, ~RowStripe)

The three modules had the following numbers of aggressor rows: 32768, 32754, and 15414. Considering that a bank in the modules has 32K rows, we conclude that large fractions of the rows are aggressors: 100%, 99.96%, and 47.04%.

Each aggressor row can be associated with a set of victim cells that were disturbed by the aggressor during either of the two tests. Figure 2.7 plots the size distribution of this set for the three modules. Aggressor rows in A_{23} are the most potent, disturbing as many as 110 cells at once. (We cannot explain the two peaks in the graph.) On the other hand, aggressors in B_{11} and C_{19} can disturb up to 28 and 5 cells, respectively.

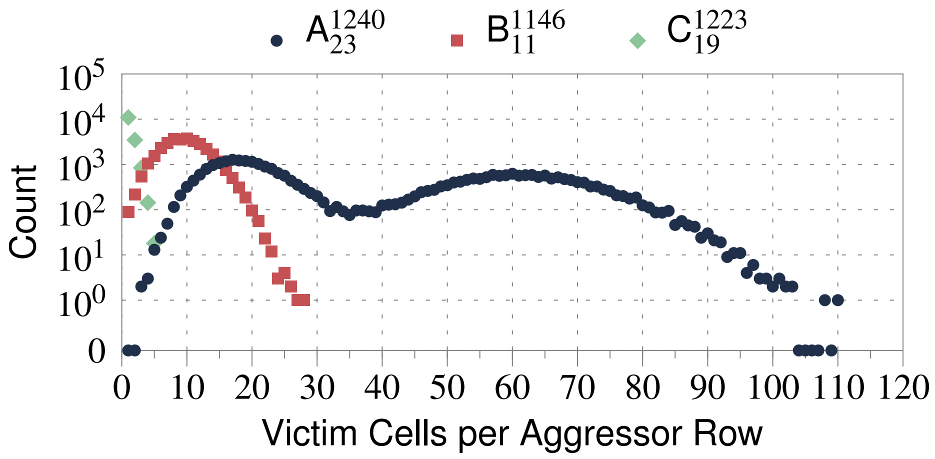


Figure 2.7. How many cells are affected by an aggressor row?

Similarly, we can associate each aggressor row with a set of *victim rows* to which the victim cells belong. Figure 2.8 plots the size distribution of this set. We see that the victim cells of an aggressor row are predominantly localized to two rows or less. In fact, only a small fraction of aggressor rows affect three rows or more: 2.53%, 0.0122%, and 0.00649%.

To see whether any correlation exists between the address of an aggressor row and those of its victim rows, we formed every possible pair between them. For each such pair,

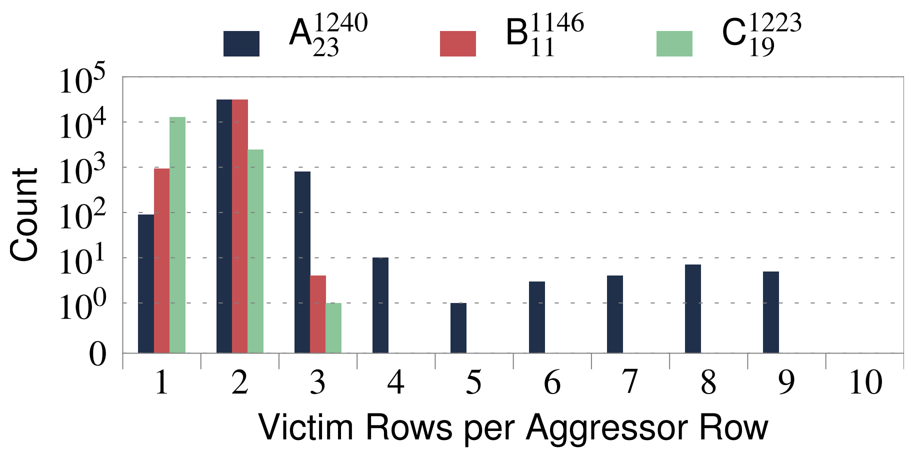


Figure 2.8. How many rows are affected by an aggressor row?

we then computed the row-address difference as follows: $VictimRow_{addr} - AggressorRow_{addr}$. The histogram of these differences is shown in Figure 2.9. It is clear from the figure that an aggressor causes errors in rows only other than itself. This is understandable since every time an aggressor is opened and closed, it also serves to replenish the charge in all of its own cells (Section 2.2.4). Since the aggressor's cells are continuously being refreshed, it is highly unlikely that they could leak enough charge to lose their data.

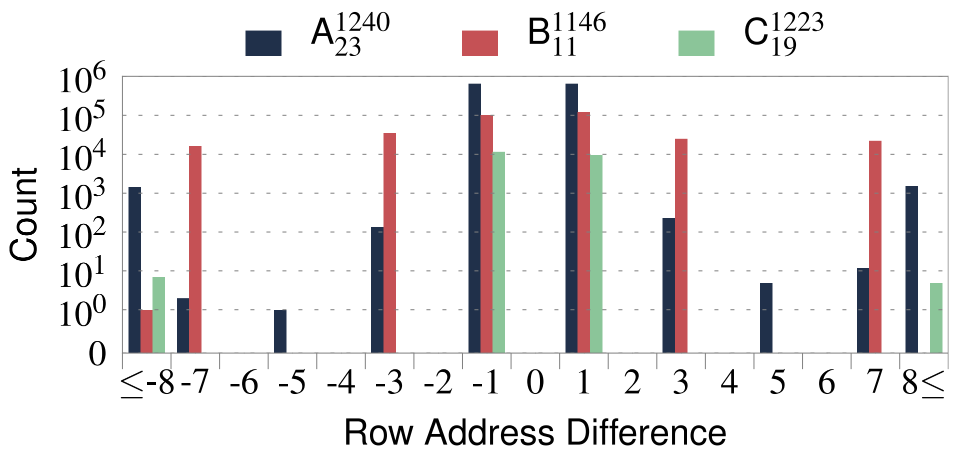


Figure 2.9. Which rows are affected by an aggressor row?

For all three modules, Figure 2.9 shows strong peaks at ± 1, suggesting that an aggressor and its victims are likely to have consecutive row-addresses, i.e., they are *logically adjacent*. Being logically adjacent, however, does not always imply that the rows are placed next to each other on the silicon die, i.e., *physically adjacent*. Although every logical row must be mapped to some physical row, it is entirely up to the DRAM manufacturer to decide how they are mapped [154]. In spite of this, we hypothesize that aggressors cause errors in their physically adjacent rows due to three reasons.

- *Reason 1.* Wordline voltage fluctuations are likely to place the greatest electrical stress on the immediately neighboring rows [110, 129].
- *Reason 2.* By definition, a row has only two immediate neighbors, which may explain why disturbance errors are localized mostly to two rows.
- *Reason 3.* Logical adjacency may highly correlate with physical adjacency, which we infer from the strong peaks at ± 1 in Figure 2.9.

However, we also see discrepancies in Figures 2.8 and 2.9, whereby an aggressor row appears to cause errors in non-adjacent rows. We hypothesize that this is due to two reasons.

- *Reason 1.* In Figure 2.8, some aggressors affect more than just two rows. This may be an irregularity caused by *re-mapped* rows. Referring back to Figure 2.2 (Section 2.2.1), the i^{th} "row" of a rank is formed by taking the i^{th} row in each chip and concatenating them. But if the row in one of the chips is faulty, the manufacturer re-maps it to a spare row (e.g., $i \rightarrow j$) [47]. In this case, the i^{th} "row" has *four* immediate neighbors: $i \pm 1^{th}$ rows in seven chips and $j \pm 1^{th}$ rows in the re-mapped chip.
- *Reason 2.* In Figure 2.9, some aggressors affect rows that are not logically-adjacent: e.g., side peaks at ± 3 and ± 7. This may be an artifact of manufacturer-dependent mapping, where some physically-adjacent rows have logical row-addresses that differ by ± 3 or ± 7 — for example, when the addresses are gray-encoded [154]. Alternatively, it could be that aggressors affect rows farther away than the immediate neighbors — a possibil-

ity that we cannot completely rule out. However, if that were the case, then it would be unlikely for the peaks to be separated by gaps at ± 2, ± 4, and ± 6.[14]

Double Aggressor Rows. Most victim cells are disturbed by only a *single* aggressor row. However, there are some victim cells that are disturbed by *two different* aggressor rows. In the first bank of the three modules, the numbers of such victim cells were 83, 2, and 0. In module A_{23}, for example, the victim cell at (row 1464, column 50466) had a '1'→'0' error when *either* row 1463 *or* row 1465 was toggled. In module B_{11}, the victim cell at (row 5907, column 32087) had a '0'→'1' error when row 5906 was toggled, whereas it had a '1'→'0' error when row 5908 was toggled. Within these two modules respectively, the same trend applies to the other victim cells with two aggressor rows. Interestingly, the two victim cells in module B_{11} with two aggressor rows were also the same cells that had errors for both runs of the test pair described in Section 2.6.1. These cells were the only cases in which we observed both '0'→'1' *and* '1'→'0' errors in the same cell. Except for such rare exceptions found only in B modules, every other victim cell had an error in just a single preferred direction, for reasons we next explain.

2.6.4. Data Pattern Dependence

Until now, we have treated all errors equally without making any distinction between the two different *directions* of errors: '0'⇆'1'. When we categorized the errors in Tables 2.3, 2.4, and 2.5 based on their direction, an interesting trend emerged. Whereas A modules did not favor one direction over the other, B and C modules heavily favored '1'→'0' errors. Averaged on a module-by-module basis, the relative fraction of '1'→'0' errors is 49.9%, 92.8%, and 97.1% for A, B, and C.[15]

The seemingly asymmetric nature of disturbance errors is related to an intrinsic prop-

[14] Figure 2.9 presents further indications of re-mapping, where some modules have non-zero values for ± 8 or beyond. Such large differences — which in some cases reach into the thousands — may be caused when a faulty row is re-mapped to a spare row that is far away, which is typically the case [47].

[15] For manufacturer C, we excluded modules with a die version of $\mathcal{B}$. Unlike other modules from the same manufacturer, these modules had errors that were evenly split between the two directions.

erty of DRAM cells called *orientation*. Depending on the implementation, some cells represent a logical value of '1' using the charged state, while other cells do so using the discharged state — these cells are referred to as *true-cells* and *anti-cells*, respectively [97]. If a true-cell loses charge, it experiences a '1'→'0' error. When we profiled two modules (B_{11} and C_{19}), we discovered that they consist mostly of true-cells by a ratio of 1000s-to-1.[16] For these two modules, the dominance of true-cells and their '1'→'0' errors imply that victim cells are most likely to *lose* charge when they are disturbed. The same conclusion also applies to A_{23}, whose address-space is divided into large swaths of true- and anti-cells that alternate every 512 rows. For this module, we found that '1'→'0' errors are dominant (>99.8%) in rows where true-cells are dominant: rows 0–511, 1024–1535, 2048–2559, $\cdots$. In contrast, '0'→'1' errors are dominant (>99.7%) in the remainder of the rows where anti-cells are dominant. Regardless of its orientation, a cell can lose charge only if it was initially charged — explaining why a given cell did not have errors in both runs of the test in Section 2.6.1. Since the two runs populate the module with inverse data patterns, a cell cannot be charged for both runs.

Table 2.8 reports the numbers of errors that were induced in three modules using four different data patterns and their inverses: Solid, RowStripe, ColStripe, and Checkered. Among them, RowStripe (even/odd rows '0's/'1's) induces the most errors for A_{23} and B_{11}, as well as the second most errors for C_{19}. In contrast, Solid (all '0's) has the fewest errors for all three modules by an order of magnitude or more. Such a large difference *cannot* be explained if the requirements for a disturbance error are only two-fold: *(i)* a victim cell is in the charged state, and *(ii)* its aggressor row is toggled. This is because the same two requirements are satisfied by all four pairs of data patterns. Instead, there must be other factors at play than just the coupling of a victim cell with an aggressor wordline. In fact, we discovered that the behavior of most victim cells is correlated with the data stored in

[16]At 70°C, we wrote all '0's to the module, disabled refreshes for six hours and read out the module. We then repeated the procedure with all '1's. A cell was deemed to be true (or anti) if its outcome was '0' (or '1') for both experiments. We could not resolve the orientation of every cell.

some other cells.[17] A victim cell may have *aggressor cell(s)* — typically residing in the aggressor row — that must be discharged for the victim to have an error. A victim cell may also have *protector cell(s)* — typically residing in either the aggressor row or the victim row — that must be charged or discharged for the victim to have a lower probability of having an error. In its generalized form, disturbance errors appear to be a complicated "N-body" phenomenon involving the interaction of multiple cells, the net result of which would only explain the differences in Table 2.8.

Module	TestBulk(*DP*) + TestBulk(∼*DP*)			
	Solid	RowStripe	ColStripe	Checkered
A_{23}	112,123	**1,318,603**	763,763	934,536
B_{11}	12,050	**320,095**	9,610	302,306
C_{19}	57	20,770	130	**29,283**

Table 2.8. Number of errors for different data patterns

2.7. Sensitivity Results

Errors are Mostly Repeatable. We subjected three modules to ten *iterations* of testing, where each iteration consists of the test pair described in Section 2.6.1. Across the ten iterations, the average numbers of errors (for only the first bank) were the following: 1.31M, 339K, and 21.0K. There were no iterations that deviated by more than $\pm 0.25\%$ from the average for all three modules. The ten iterations revealed the following numbers of unique victim cells: 1.48M, 392K, and 24.4K. Most victim cells were repeat offenders, meaning that they had an error in every iteration: 78.3%, 74.4%, and 73.2%. However, some victim cells had an error in just a single iteration: 3.14%, 4.86%, and 4.76%. This implies that an exhaustive search for every possible victim cell would require a large number of iterations, necessitating several days (or more) of continuous testing. One possible

[17] We comprehensively tested the first 32 rows in module A_{19} using hundreds of different random data patterns. Through statistical analysis on the experimental results, we were able to identify *almost certain* correlations between a victim cell and the data stored in some other cells.

way to reduce the testing time is to increase the RI beyond the standardized value of 64*ms* as we did in Figure 2.4 (Section 2.6.2). However, multiple iterations could still be required since a single iteration at RI=128*ms* does not provide 100% coverage of all the victim cells at RI=64*ms*: 99.77%, 99.87%, and 99.90%.

Victim Cells $\neq$ Weak Cells. Although the retention time of every DRAM cell is required to be greater than the 64*ms* minimum, different cells have different retention times. In this context, the cells with the shortest retention times are referred to as *weak cells* [97]. Intuitively, it would appear that the weak cells are especially vulnerable to disturbance errors since they are already leakier than others. On the contrary, we did not find any strong correlation between weak cells and victim cells. We searched for a module's weak cells by neither accessing nor refreshing a module for a generous amount of time (10 *seconds*) after having populated it with either all '0's or all '1's. If a cell was corrupted during this procedure, we considered it to be a weak cell [97]. In total, we were able to identify ~1M weak cells for each module (984K, 993K, and 1.22M), which is on par with the number of victim cells. However, only a few weak cells were also victim cells: 700, 220, and 19. Therefore, we conclude that the coupling pathway responsible for disturbance errors may be independent of the process variation responsible for weak cells.

Not Strongly Affected by Temperature. When temperature increases by 10°C, the retention time for each cell is known to decrease by almost a factor of two [81, 97]. To see whether this would drastically increase the number of errors, we ran a single iteration of the test pair for the three modules at 70±2.0°C, which is 20°C higher than our default ambient temperature. Compared to an iteration at 50°C, the number of errors did not change greatly: +10.2%, −0.553%, and +1.32%. We also ran a single iteration of the test pair for the three modules at 30±2.0°C with similar results: −14.5%, +2.71%, and −5.11%. From this we conclude that disturbance errors are not strongly influenced by temperature.

2.8. Solutions to Disturbance Errors

We examine seven solutions to tolerate, prevent, or mitigate disturbance errors. Each solution makes a different trade-off between feasibility, cost, performance, power, and reliability. Among them, we believe our seventh and last solution, called *PARA*, to be the most efficient and low-overhead. Section 2.8.1 discusses each of the first six solutions. Section 2.8.2 analyzes our seventh solution (PARA) in detail.

2.8.1. Six Potential Solutions

1. Make better chips. Manufacturers could fix the problem at the chip-level by improving circuit design. However, the problem could resurface when the process technology is upgraded. In addition, this may get worse in the future as cells become smaller and more vulnerable.

2. Correct errors. Server-grade systems employ ECC modules with extra DRAM chips, incurring a 12.5% capacity overhead. However, even such modules cannot correct multi-bit disturbance errors (Section 2.6.3). Due to their high cost, ECC modules are rarely used in consumer-grade systems.

3. Refresh all rows frequently. Disturbance errors can be eliminated for sufficiently short refresh intervals ($RI \leq RI_{th}$) as we saw in Section 2.6.2. However, frequent refreshes also degrade performance and energy-efficiency. Today's modules already spend 1.4–4.5% of their time just performing refreshes [68]. This number would increase to 11.0–35.0% if the refresh interval is shortened to 8.2*ms*, which is required by A_{20} (Table 2.3). Such a high overhead is unlikely to be acceptable for many systems.

4. Retire cells (manufacturer). Before DRAM chips are sold, the manufacturer could identify victim cells and re-map them to spare cells [47]. However, an exhaustive search for all victim cells could take several days or more (Section 2.7). In addition, if there are many victim cells, there may not be enough spare cells for all of them.

5. Retire cells (end-user). The end-users themselves could test the modules and employ system-level techniques for handling DRAM reliability problems: disable faulty addresses [3, 45, 147, 156], re-map faulty addresses to reserved addresses [119, 123], or refresh faulty addresses more frequently [98, 156]. However, the first/second approaches are ineffective when every row in the module is a victim row (Section 2.6.3). On the other hand, the third approach is inefficient since it always refreshes the victim rows more frequently — even when the module is not being accessed at all. In all three approaches, the end-user pays for the cost of identifying and storing the addresses of the aggressor/victim rows.

6. Identify "hot" rows and refresh neighbors. Perhaps the most intuitive solution is to identify frequently opened rows and refresh only their neighbors. The challenge lies in minimizing the hardware cost to identify the "hot" rows. For example, having a counter for each row would be too expensive when there are millions of rows in a system.[18] The generalized problem of identifying frequent items (from a stream of items) has been extensively studied in other domains. We applied a well-known method [78] and found that while it reduces the number of counters, it also requires expensive operations to query the counters (e.g., highly-associative search). We also analyzed approximate methods which further reduce the storage requirement: Bloom Filters [17], Morris Counters [112], and variants thereof [30, 35, 155]. These approaches, however, rely heavily on hash functions and, therefore, introduce hash collisions. Whenever *one* counter exceeds the threshold value, *many* rows are falsely flagged as being "hot," leading to a torrent of refreshes to all of their neighbors.

[18]Several patent applications propose to maintain an array of counters ("detection logic") in either the memory controller [11, 12, 39] or in the DRAM chips themselves [13, 10, 40]. If the counters are tagged with the addresses of only the most recently activated rows, their number can be significantly reduced [39].

2.8.2. Seventh Solution: PARA

Our main proposal to prevent DRAM disturbance errors is a low-overhead mechanism called *PARA* (*probabilistic adjacent row activation*). The key idea of PARA is simple: every time a row is opened and closed, one of its adjacent rows is also opened (i.e., refreshed) with some low probability. If one particular row happens to be opened and closed repeatedly, then it is statistically certain that the row's adjacent rows will eventually be opened as well. The main advantage of PARA is that it is *stateless*. PARA does not require expensive hardware data-structures to count the number of times that rows have been opened or to store the addresses of the aggressor/victim rows.

Implementation. PARA is implemented in the memory controller as follows. Whenever a row is closed, the controller flips a biased coin with a probability p of turning up heads, where $p \ll 1$. If the coin turns up heads, the controller opens one of its adjacent rows where either of the two adjacent rows are chosen with equal probability ($p/2$). Due to its probabilistic nature, PARA does *not* guarantee that the adjacent will always be refreshed in time. Hence, PARA *cannot* prevent disturbance errors with absolute certainty. However, its parameter p can be set so that disturbance errors occur at an extremely low probability — many orders of magnitude lower than the failure rates of other system components (e.g., more than 1% of hard-disk drives fail every year [126, 137]).

Error Rate. We analyze PARA's error probability by considering an adversarial access pattern that opens and closes a row just enough times (N_{th}) during a refresh interval but no more. Every time the row is closed, PARA flips a coin and refreshes a given adjacent row with probability $p/2$. Since the coin-flips are independent events, the number of refreshes to one particular adjacent row can be modeled as a random variable X that is binomially-distributed with parameters $B(N_{th}, p/2)$. An error occurs in the adjacent row only if it is never refreshed during any of the N_{th} coin-flips (i.e., X=0). Such an event has the following probability of occurring: $(1-p/2)^{N_{th}}$. When p=0.001, we evaluate this probability in Table 2.9 for different values of N_{th}. The table shows two error probabilities: one in which the adversarial access pattern is sustained for 64*ms* and the other for one *year*. Recall from Section 2.6.2 that realistic values for N_{th} in our modules are in the range of 139K–284K. For p=0.001 and N_{th}=100K, the probability of experiencing an error in one year is negligible at 9.4×10^{-14}.

Duration	N_{th}=50K	N_{th}=100K	N_{th}=200K
64*ms*	1.4×10^{-11}	1.9×10^{-22}	3.6×10^{-44}
1 *year*	6.8×10^{-3}	9.4×10^{-14}	1.8×10^{-35}

Table 2.9. Error probabilities for PARA when p=0.001

Adjacency Information. For PARA to work, the memory controller must know which rows are physically adjacent to each other. This is also true for alternative solutions based on “hot” row detection (Section 2.8.1). Without this information, rows cannot be selectively refreshed, and the only safe resort is to blindly refresh *all* rows in the same bank, incurring a large performance penalty. To enable low-overhead solutions, we argue for the manufacturers to disclose how they map logical rows onto physical rows.[19] Such a *mapping function* could possibly be as simple as specifying the bit-offset within the logical row-address that is used as the least-significant-bit of the physical row-address. Along with other metadata about the module (e.g., capacity, and bus frequency), the mapping function could be stored in a small ROM (called the *SPD*) that exists on every DRAM module [71]. The manufacturers should also disclose how they re-map faulty physical rows (Section 2.6.3). When a faulty physical row is re-mapped, the logical row that had mapped to it acquires a new set of physical neighbors. The SPD could also store the *re-mapping function*, which specifies how the logical row-addresses of those new physical neighbors can be computed. To account for the possibility of re-mapping, PARA can be configured to *(i)* have a higher value of p and *(ii)* choose a row to refresh from a wider pool of candidates, which includes the re-mapped neighbors in addition to the original neighbors.

[19]Bains et al. [13] make the same argument. As an alternative, Bains et al. [11, 12] propose a new DRAM command called “targeted refresh”. When the memory controller sends this command along with the target row address, the DRAM chip is responsible for refreshing the row and its neighbors.

Performance Overhead. Using a cycle-accurate DRAM simulator, we evaluate PARA's performance impact on 29 single-threaded workloads from SPEC CPU2006, TPC, and memory-intensive microbenchmarks (We assume a reasonable system setup [87] with a 4GHz out-of-order core and dual-channel DDR3-1600.) Due to re-mapping, we conservatively assume that a row can have up to *ten* different rows as neighbors, not just two. Correspondingly, we increase the value of p by five-fold to 0.005.[20] Averaged across all 29 benchmarks, there was only a 0.197% degradation in instruction throughput during the simulated duration of 100*ms*. In addition, the largest degradation in instruction throughput for any single benchmark was 0.745%. From this, we conclude that PARA has a small impact on performance, which we believe is justified by the *(i)* strong reliability guarantee and *(ii)* low design complexity resulting from its stateless nature.

2.9. Other Related Work

Disturbance errors are a general class of reliability problem that afflicts not only DRAM, but also other memory and storage technologies: SRAM [28, 42, 83], flash [15, 19, 20, 31, 41], and hard-disk [76, 148, 160]. Van de Goor and de Neef [153] present a collection of production tests that can be employed by DRAM manufacturers to screen faulty chips. One such test is the "hammer," where each cell is written a thousand times to verify that it does not disturb nearby cells. In 2013, one test equipment company mentioned the "row hammer" phenomenon in the context of DDR4 DRAM [103], the next generation of commodity DRAM. To our knowledge, no previous work demonstrated and characterized the phenomenon of disturbance errors in DRAM chips from the field.

[20]We do not make any special considerations for victim cells with two aggressor rows (Section 2.6.3). Although they could be disturbed by either aggressor row, they could also be refreshed by either aggressor row.

2.10. Chapter Summary

We have demonstrated, characterized, and analyzed the phenomenon of disturbance errors in modern commodity DRAM chips. These errors happen when repeated accesses to a DRAM row corrupts data stored in other rows. Based on our experimental characterization, we conclude that disturbance errors are an emerging problem likely to affect current and future computing systems. We propose several solutions, including a new stateless mechanism that provides a strong statistical guarantee against disturbance errors by probabilistically refreshing rows adjacent to an accessed row. As DRAM process technology scales down to smaller feature sizes, we hope that our findings will enable new system-level [116] approaches to enhance DRAM reliability.

Chapter 3

Subarray Parallelism: A High-Performance DRAM Architecture

The large latency of main memory is a well-known bottleneck for overall system performance. As a coping mechanism, modern processors employ numerous techniques to expose multiple requests to main memory, in an effort to overlap their latencies: e.g., out-of-order execution [150], non-blocking caches [92], prefetching, and multi-threading.

The effectiveness of such techniques, however, depends critically on whether the memory requests are actually served in parallel. For this purpose, DRAM chips are divided into several banks, each of which can be accessed independently. Nevertheless, if two memory requests go to the same bank, they must be served one after another — experiencing what is referred to as a *bank conflict*.

3.1. Bank Conflicts Exacerbate DRAM Latency

Bank conflicts have two negative consequences. First, they serialize the memory requests, and increase the effective latency of accessing main memory. As a result, processing cores are more likely to experience stalls, which would lead to reduced system performance. And to make matters worse, a memory request scheduled after a write request to the same bank incurs an additional latency called the *write-recovery penalty*. Furthermore, this penalty is expected to increase by more than 5x in the near future due to worsening process variation, which creates increasingly slow outlier DRAM cells [77].

Second, a bank conflict could cause thrashing in the bank's row-buffer. A *row-buffer*, present in each bank, effectively acts as a "cache" for the rows in the bank. Memory requests that hit in the row-buffer incur much lower latency than those that miss. In a multi-core system, requests from different applications are interleaved with each other. When such interleaved requests lead to bank conflicts, they can "evict" the row that is present in the row-buffer. As a result, requests of an application that could have otherwise hit in the row-buffer will miss in the row-buffer, significantly degrading the performance of the application (and potentially the overall system) [113, 117, 143, 167].

A solution to the bank conflict problem is to increase the number of DRAM banks in the system. While current memory subsystems theoretically allow for three ways of doing so, they all come at a significantly high cost. First, one can increase the number of banks in the DRAM chip itself. However, for a constant storage capacity, increasing the number of banks-per-chip significantly increases the DRAM die area (and thus chip cost) due to replicated decoding logic, routing, and drivers at each bank [164]. Second, one can increase the number of banks in a channel by multiplexing the channel with many memory modules, each of which is a collection of banks. Unfortunately, this increases the electrical load on the channel, causing it to run at a significantly reduced frequency [37, 38]. Third, one can add more memory channels to increase the overall bank count. Unfortunately, this increases the pin-count in the processor package, which is an expensive resource.[1] Considering both the low growth rate of pin-count and the prohibitive cost of pins in general, it is clear that increasing the number of channels is not a scalable solution.

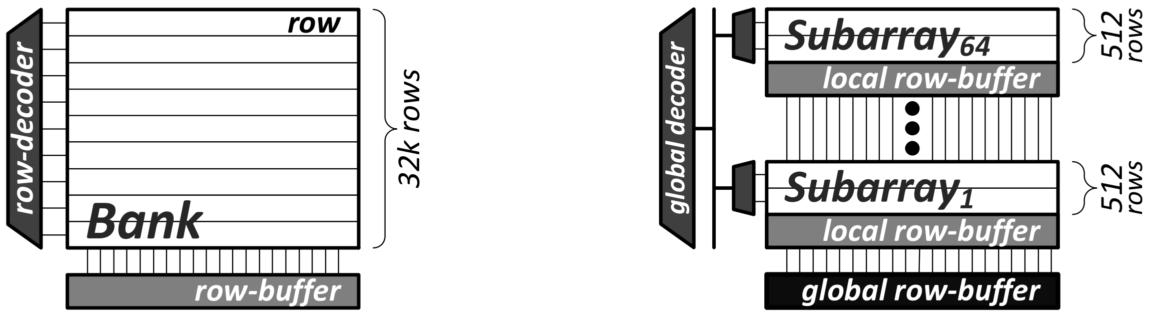
[1]Intel Sandy Bridge dedicates 264 pins for two channels [54]. IBM POWER7 dedicates 640 pins for eight channels [139].

This chapter's goal is to mitigate the detrimental effects of bank conflicts with a low-cost approach. We make two key observations that lead to our proposed mechanisms.

First, a modern DRAM bank is *not* implemented as a monolithic component with a single row-buffer. Implementing a DRAM bank as a monolithic structure requires very long wires (called bitlines), to connect the row-buffer to all the rows in the bank, which can significantly increase the access latency (Section 3.2.3). Instead, a bank consists of multiple *subarrays*, each with its own local row-buffer, as shown in Figure 3.1. Subarrays within a bank share *(i)* a global row-address decoder and *(ii)* a set of global bitlines which connect their local row-buffers to a global row-buffer.

Second, the latency of bank access consists of three major components: *(i)* opening a row containing the required data (referred to as *activation*), *(ii)* accessing the data (*read* or *write*), and *(iii)* closing the row (*precharging*). In existing systems, all three operations must be completed for one memory request before serving another request to a different row within the same bank, even if the two rows reside in different subarrays. However, this need not be the case for two reasons. First, the activation and precharging operations are mostly local to each subarray, enabling the opportunity to overlap these operations to different subarrays within the same bank. Second, if we reduce the resource sharing among subarrays, we can enable activation operations to different subarrays to be performed in parallel and, in addition, also exploit the existence of multiple local row-buffers to cache more than one row in a single bank, enabling the opportunity to improve row-buffer hit rate.

Based on these observations, our proposition in this chapter is that exposing the subarray-level internal organization of a DRAM bank to the memory controller would allow the controller to exploit the independence between subarrays within the same bank and reduce the negative impact of bank conflicts. To this end, we propose three different mechanisms for exploiting *subarray-level parallelism*. Our proposed mechanisms allow the memory controller to overlap or eliminate different latency components required to complete mul-



(a) Logical abstraction **(b)** Physical implementation

Figure 3.1. DRAM bank organization

tiple requests going to different subarrays within the same bank.

First, *SALP-1* (Subarray-Level-Parallelism-1) overlaps the latency of closing a row of one subarray with that of opening a row in a different subarray within the same bank by pipelining the two operations one after the other. SALP-1 requires *no* changes to the existing DRAM structure. Second, *SALP-2* (Subarray-Level-Parallelism-2) allows the memory controller to start opening a row in a subarray before closing the currently open row in a different subarray. This allows SALP-2 to overlap the latency of opening a row with the write-recovery period of another row in a different subarray, and further improve performance compared to SALP-1. SALP-2 requires the addition of small latches to each subarray's peripheral logic. Third, *MASA* (Multitude of Activated Subarrays) exploits the fact that each subarray has its own *local row-buffer* that can potentially "cache" the most recently accessed row in that subarray. MASA reduces hardware resource sharing between subarrays to allow the memory controller to *(i)* activate multiple subarrays in parallel to reduce request serialization, *(ii)* concurrently keep local row-buffers of multiple subarrays active to significantly improve row-buffer hit rate. In addition to the change needed by SALP-2, MASA requires only the addition of a single-bit latch to each subarray's peripheral logic as well as a new 1-bit global control signal.

This chapter makes the following contributions.

- We exploit the existence of subarrays within each DRAM bank to mitigate the effects of bank conflicts. We propose three mechanisms, SALP-1, SALP-2, and MASA, that overlap (to varying degrees) the latency of accesses to different subarrays. SALP-1 does not require any modifications to existing DRAM structure, while SALP-2 and MASA introduce small changes only to the subarrays' peripheral logic.

- We exploit the existence of local subarray row-buffers within DRAM banks to mitigate row-buffer thrashing. We propose MASA that allows multiple such subarray row-buffers to remain activated at any given point in time. We show that MASA can significantly increase row-buffer hit rate while incurring only modest implementation cost.

- We perform a thorough analysis of area and power overheads of our proposed mechanisms. MASA, the most aggressive of our proposed mechanisms, incurs a DRAM chip area overhead of 0.15% and a modest power cost of 0.56mW per each additionally activated subarray.

- We identify that *tWR* (*bank write-recovery*[2]) worsens the negative impact of bank conflicts by increasing the latency of critical read requests. We show that SALP-2 and MASA are effective at minimizing the negative effects of *tWR*.

- We evaluate our proposed mechanisms using a variety of system configurations and show that they significantly improve performance for single-core systems compared to conventional DRAM: 7%/13%/17% for SALP-1/SALP-2/MASA, respectively. Our schemes also interact positively with application-aware memory scheduling algorithms and further improve performance for multi-core systems.

3.2. Background: DRAM Organization

As shown in Figure 3.2, DRAM-based main memory systems are logically organized as a hierarchy of channels, ranks, and banks. In today's systems, *banks* are the smallest

[2]Write-recovery (explained in Section 3.2.2) is different from the *bus-turnaround penalty* (read-to-write, write-to-read), which is addressed by several prior works [27, 94, 142].

memory structures that can be accessed in parallel with respect to each other. This is referred to as *bank-level parallelism* [86, 118]. Next, a *rank* is a collection of banks across multiple DRAM chips that operate in lockstep.[3] Banks in different ranks are fully decoupled with respect to their device-level electrical operation and, consequently, offer better bank-level parallelism than banks in the same rank. Lastly, a *channel* is the collection of all banks that share a common physical link (command, address, data buses) to the processor. While banks from the same channel experience contention at the physical link, banks from different channels can be accessed completely independently of each other. Although the DRAM system offers varying degrees of parallelism at different levels in its organization, two memory requests that access the same bank must be served one after another. To understand why, let us examine the logical organization of a DRAM bank as seen by the memory controller.

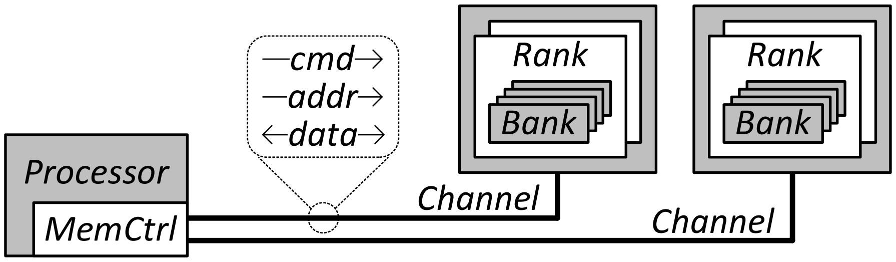


Figure 3.2. Logical hierarchy of main memory

3.2.1. Bank: Logical Organization & Operation

Figure 3.3 presents the logical organization of a DRAM bank. A DRAM bank is a two-dimensional array of capacitor-based DRAM cells. It is viewed as a collection of *rows*, each of which consists of multiple *columns*. Each bank contains a *row-buffer* which is an array of sense-amplifiers that act as latches. Spanning a bank in the column-wise direction are the *bitlines*, each of which can connect a sense-amplifier to any of the cells in the same

[3]A DRAM rank typically consists of eight DRAM chips, each of which has eight banks. Since the chips operate in lockstep, the rank has only eight independent banks, each of which is the set of the i^{th} bank across all chips.

Category	Row Cmd ↔ Row Cmd			Row Cmd ↔ Col Cmd		
Name	*tRC*	*tRAS*	*tRP*	*tRCD*	*tRTP*	*tWR**
Commands	A→A	A→P	P→A	A→R/W	R→P	W*→P
Scope	Bank	Bank	Bank	Bank	Bank	Bank
Value (ns)	$\boxed{\sim\mathbf{50}}$	~35	13-15	13-15	~7.5	$\boxed{\mathbf{15}}$

Category	Col Cmd ↔ Col Cmd			Col Cmd → Data	
Name	*tCCD*	*tRTW*†	*tWTR**	*CL*	*CWL*
Commands	R(W)→R(W)	R→W	W*→R	R→DATA	W→DATA
Scope	Channel	Rank	Rank	Bank	Bank
Value (ns)	5-7.5	11-15	~7.5	13-15	10-15

A: `ACTIVATE`– P: `PRECHARGE`– R: `READ`– W: `WRITE`

∗ Goes into effect after the last write *data*, not from the `WRITE` command

† Not explicitly specified by the DDR3 standard [64]. Defined as a function of other timing constraints.

Table 3.1. Summary of DDR3-SDRAM timing constraints [64]

column. A *wordline* (one for each row) determines whether or not the corresponding row of cells is connected to the bitlines.

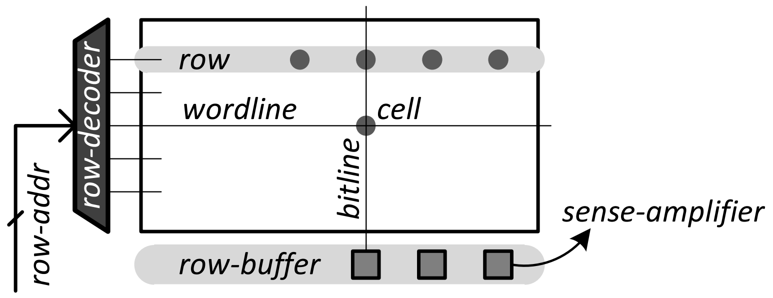


Figure 3.3. DRAM Bank: Logical organization

To serve a memory request that accesses data at a particular row and column address, the memory controller issues three commands to a bank in the order listed below. Each command triggers a specific sequence of events within the bank.

1. `ACTIVATE`: read the entire row into the row-buffer

2. `READ`/`WRITE`: access the column from the row-buffer

3. `PRECHARGE`: de-activate the row-buffer

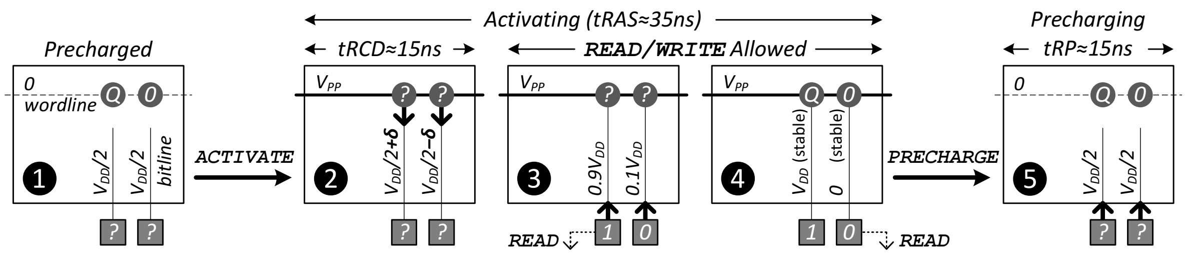


Figure 3.4. DRAM bank operation: Steps involved in serving a memory request [60] ($V_{PP} > V_{DD}$)

`ACTIVATE` **Row**. Before a DRAM row can be activated, the bank must be in the *precharged state* (State ❶, Figure 3.4). In this state, all the bitlines are maintained at a voltage-level of $\frac{1}{2}V_{DD}$. Upon receiving the `ACTIVATE` command along with a row-address, the wordline corresponding to the row is raised to a voltage of V_{PP}, connecting the row's cells to the bitlines (State ❶→❷). Subsequently, depending on whether a cell is charged (Q) or uncharged (0), the bitline voltage is slightly perturbed towards V_{DD} or 0 (State ❷). The row-buffer "senses" this perturbation and "amplifies" it in the same direction (State ❷→❸). During this period when the bitline voltages are still in transition, the cells are left in an undefined state. Finally, once the bitline voltages stabilize, cell charges are restored to their original values (State ❹). The time taken for this entire procedure is called *tRAS* (≈ 35ns).

`READ`/`WRITE` **Column**. After an `ACTIVATE`, the memory controller issues a `READ` or a `WRITE` command, along with a column address. The timing constraint between an `ACTIVATE` and a subsequent column command (`READ`/`WRITE`) is called *tRCD* (≈ 15ns). This reflects the time required for the data to be latched in the row-buffer (State ❸). If the next request to the bank also happens to access the same row, it can be served with only a column command, since the row has already been activated. As a result, this request is served more quickly than a request that requires a new row to be activated.

`PRECHARGE` **Bank**. To activate a new row, the memory controller must first take the bank back to the precharged state (State ❺). This happens in two steps. First, the wordline corresponding to the currently activated row is lowered to zero voltage, disconnecting the cells from the bitlines. Second, the bitlines are driven to a voltage of $\frac{1}{2}V_{DD}$. The time taken for this operation is called *tRP* (≈ 15ns).

3.2.2. Timing Constraints

As described above, different DRAM commands have different latencies. Undefined behavior may arise if a command is issued before the previous command is fully processed. To prevent such occurrences, the memory controller must obey a set of timing constraints while issuing commands to a bank. These constraints define when a command becomes ready to be scheduled, depending on all other commands issued before it to the same channel, rank, or bank. Table 3.1 summarizes the most important timing constraints between `ACTIVATE` (A), `PRECHARGE` (P), `READ` (R), and `WRITE` (W) commands. Among these, two timing constraints (highlighted in bold) are the critical bottlenecks for bank conflicts: tRC and tWR.

tRC. Successive `ACTIVATE`s to the same bank are limited by *tRC* (row-cycle time), which is the sum of tRAS and tRP [60]. In the worst case, when N requests all access different rows within the same bank, the bank must activate a new row and precharge it for each request. Consequently, the last request experiences a DRAM latency of $N \cdot tRC$, which can be hundreds or thousands of nanoseconds.

tWR. After issuing a `WRITE` to a bank, the bank needs additional time, called *tWR* (write-recovery latency), while its row-buffer drives the bitlines to their new voltages. A bank cannot be precharged before then – otherwise, the new data may not have been safely stored in the cells. Essentially, after a `WRITE`, the bank takes longer to reach State ❹ (Figure 3.4), thereby delaying the next request to the same bank even longer than tRC.

3.2.3. Subarrays: Physical Organization of Banks

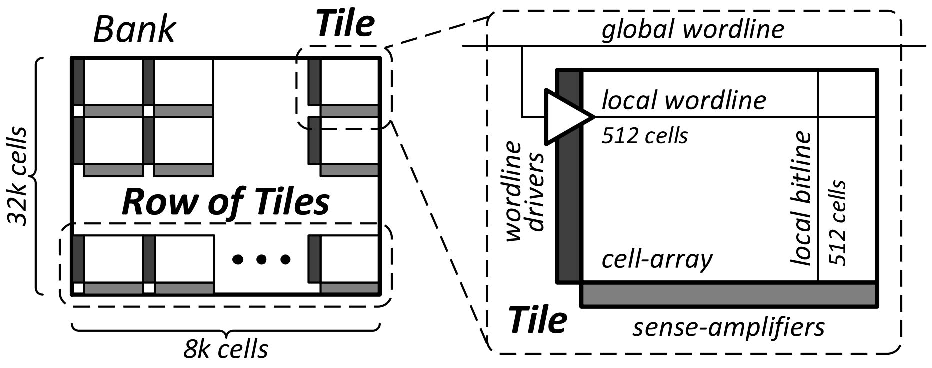
Although we have described a DRAM bank as a monolithic array of rows equipped with a single row-buffer, implementing a large bank (e.g., 32k rows and 8k cells-per-row) in this manner requires long bitlines. Due to their large parasitic capacitance, long bitlines have two disadvantages. First, they make it difficult for a DRAM cell to cause the necessary perturbation required for reliable sensing [80]. Second, a sense-amplifier takes longer to drive a long bitline to a target voltage-level, thereby increasing the latency of activation and precharging.

To avoid the disadvantages of long bitlines, as well as long wordlines, a DRAM bank is divided into a two-dimensional array of tiles [60, 80, 157], as shown in Figure 3.5a. A *tile* comprises *(i)* a cell-array, whose typical dimensions are 512 cells$\times$512 cells [157], *(ii)* sense-amplifiers, and *(iii)* wordline-drivers that strengthen the signals on the *global wordlines* before relaying them to the local wordlines.

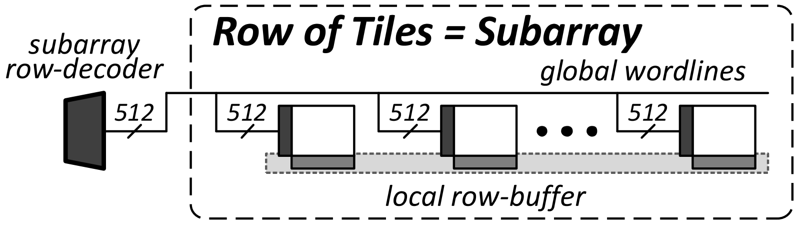
All tiles in the horizontal direction – a "row of tiles" – share the same set of global wordlines, as shown in Figure 3.5b. Therefore, these tiles are activated and precharged in lockstep. We abstract such a "row of tiles" as a single entity that we refer to as a *subarray*.[4] More specifically, a subarray is a collection of cells that share a *local row-buffer* (all sense-amplifiers in the horizontal direction) and a *subarray row-decoder* [60].

As shown in Figure 3.6, all subarray row-decoders in a bank are driven by the shared *global row-address latch* [60]. The latch holds a partially pre-decoded row-address (from the global row-decoder) that is routed by the *global address-bus* to all subarray row-decoders, where the remainder of the decoding is performed. A partially pre-decoded row-address allows subarray row-decoders to remain small and simple without incurring the large global routing overhead of a fully pre-decoded row-address [60]. All subarrays in a bank also share a *global row-buffer* [60, 82, 111] that can be connected to any one of the local row-buffers through a set of *global bitlines* [60]. The purpose of the global row-buffer is to sense the perturbations caused by the local row-buffer on the global bitlines and to amplify the perturbations before relaying them to the I/O drivers. Without a global row-buffer, the local row-buffers will take a long time to drive their values on the global

[4]We use the term subarray to refer to a *single* "row of tiles" (alternatively, a block [93]). Others have used the term subarray to refer to *(i)* an individual tile [152, 157], *(ii)* a single "row of tiles" [164], or *(iii)* multiple "rows of tiles" [111].



(a) A DRAM bank is divided into tiles



(b) Subarray: A row of tiles that operate in lockstep.

Figure 3.5. A DRAM bank consists of tiles and subarrays

bitlines, thereby significantly increasing the access latency.[5]

Although all subarrays within a bank share some global structures (e.g., the global row-address latch and the global bitlines), some DRAM operations are completely local to a subarray or use the global structures minimally. For example, precharging is completely local to a subarray and does not use any of the shared structures, whereas activation uses only the global row-address latch to drive the corresponding wordline.

Unfortunately, existing DRAMs cannot fully exploit the independence between different subarrays for two main reasons. First, *only one* row can be activated (i.e., only one wordline can be raised) within each bank at a time. This is because the global row-address latch, which determines which wordline within the bank is raised, is shared by all subarrays. Second, although each subarray has its own local row-buffer, *only one* subarray can be activated at a time. This is because all local row-buffers are connected to the global

[5]Better known as *main* [60] or *I/O* [82, 111] *sense-amplifiers*, the global row-buffer lies between the local row-buffers and the I/O driver. It is narrower than a local row-buffer; column-selection logic (not shown in Figure 3.6) multiplexes the wide outputs of the local row-buffer onto the global row-buffer.

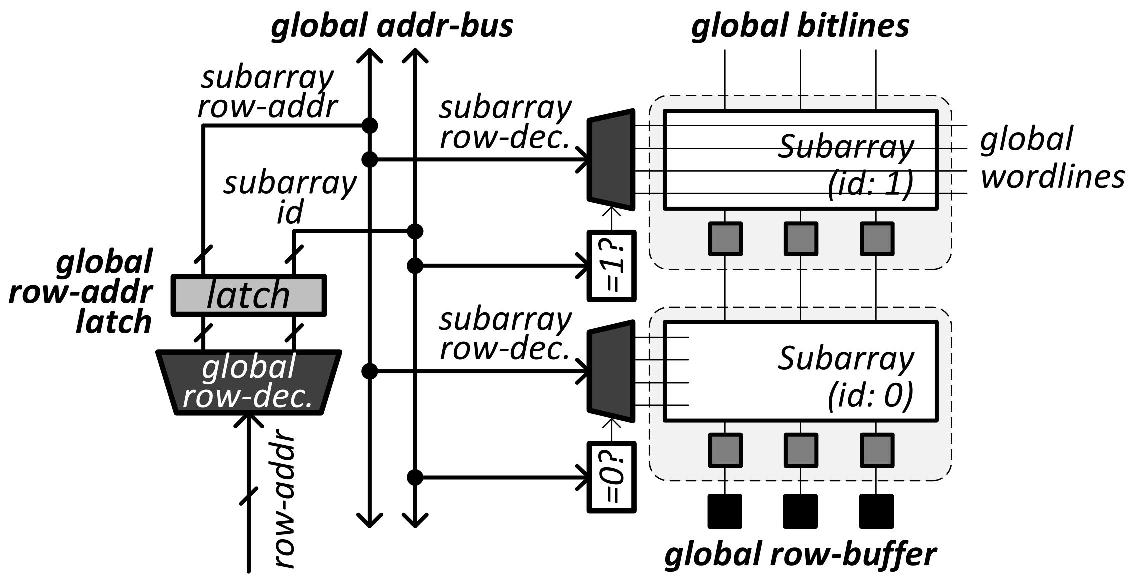


Figure 3.6. DRAM Bank: Physical organization

row-buffer by a *single* set of global bitlines. If multiple subarrays were allowed to be activated[6] at the same time when a column command is issued, all of their row-buffers would attempt to drive the global bitlines, leading to a short-circuit.

Our goal in this chapter is to reduce the performance impact of bank conflicts by exploiting the existence of subarrays to enable their parallel access and to allow multiple activated local row-buffers within a bank, using low cost mechanisms.

3.3. Motivation

To understand the benefits of exploiting the subarray-organization of DRAM banks, let us consider the two examples shown in Figure 3.7. The first example (top) presents the timeline of four memory requests being served at the same bank in a subarray-oblivious baseline.[7] The first two requests are write requests to two rows in different subarrays. The next two requests are read requests to the same two rows, respectively. This example highlights three key problems in the operation of the baseline system. First, successive

[6]We use the phrases "subarray is activated (precharged)" and "row-buffer is activated (precharged)" interchangeably as they denote the same phenomenon.
[7]This timeline (as well as other timelines we will show) is for illustration purposes and does not incorporate all DRAM timing constraints.

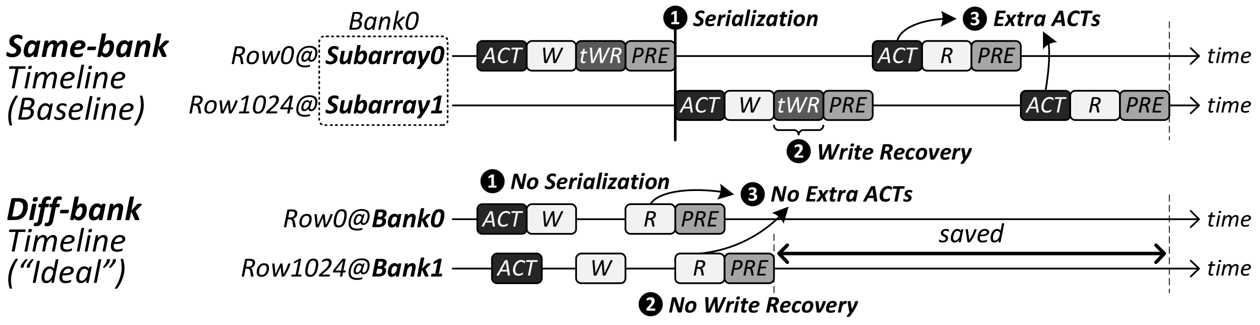


Figure 3.7. Service timeline of four requests to two different rows. The rows are in the same bank (top) or in different banks (bottom).

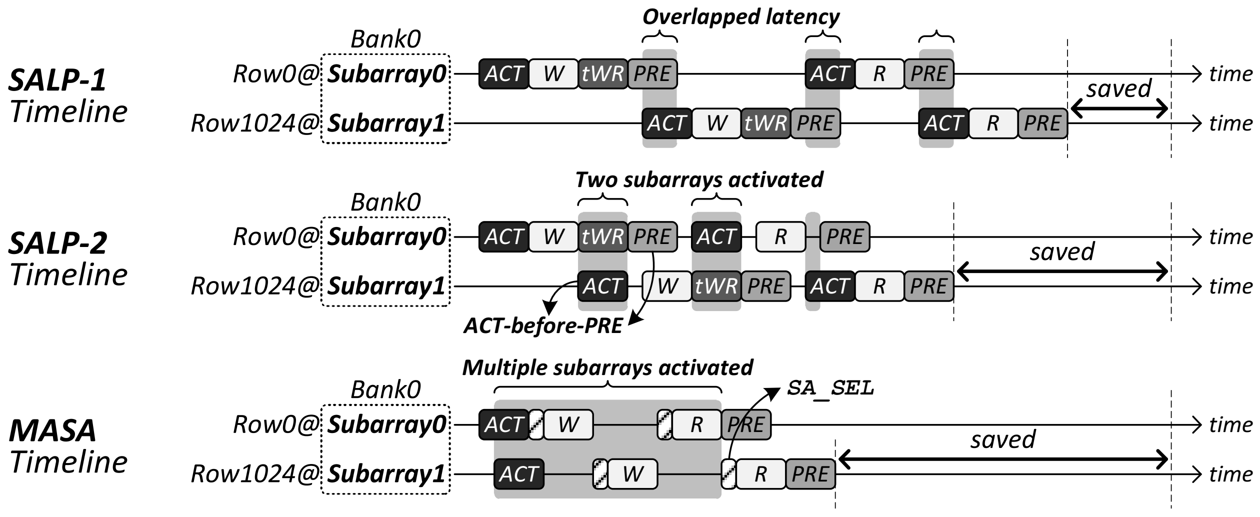


Figure 3.8. Service timeline of four requests to two different rows. The rows are in the same bank, but in different subarrays.

requests are completely serialized. This is in spite of the fact that they are to different subarrays and could potentially have been partially parallelized. Second, requests that immediately follow a `WRITE` incur the additional write-recovery latency (Section 3.2.2). Although this constraint is completely local to a subarray, it delays a subsequent request even to a different subarray. Third, both rows are activated twice, once for each of their two requests. After serving a request from a row, the memory controller is forced to de-activate the row since the subsequent request is to a different row within the same bank. This significantly increases the overall service time of the four requests.

The second example (bottom) in Figure 3.7 presents the timeline of serving the four requests when the two rows belong to different *banks*, instead of to different subarrays within the same bank. In this case, the overall service time is significantly reduced due to three reasons. First, rows in different banks can be activated in parallel, overlapping a large portion of their access latencies. Second, the write-recovery latency is local to a bank and hence, does not delay a subsequent request to another bank. In our example, since consecutive requests to the same bank access the same row, they are also not delayed by the write-recovery latency. Third, since the row-buffers of the two banks are completely independent, requests do not evict each other's rows from the row-buffers. This eliminates the need for extra `ACTIVATE`s for the last two requests, further reducing the overall service time. However, as we described in Section 3.1, increasing the number of banks in the system significantly increases the system cost.

In this chapter, we contend that most of the performance benefits of having multiple banks can be achieved at a significantly lower cost by exploiting the potential parallelism offered by subarrays within a bank. To this end, we propose three mechanisms that exploit the existence of subarrays with little or no change to the existing DRAM designs.

3.4. Overview of Proposed Mechanisms

We call our three proposed schemes *SALP-1*, *SALP-2* and *MASA*. As shown in Figure 3.8, each scheme is a successive refinement over the preceding scheme such that the performance benefits of the most sophisticated scheme, MASA, subsumes those of SALP-1 and SALP-2. We explain the key ideas of each scheme below.

3.4.1. SALP-1: Subarray-Level-Parallelism-1

The key observation behind SALP-1 is that precharging and activation are mostly local to a subarray. SALP-1 exploits this observation to overlap the precharging of one subarray with the activation of another subarray. In contrast, existing systems always serialize precharging and activation to the same bank, conservatively provisioning for when they are to the same subarray. SALP-1 requires *no modifications* to existing DRAM structure.

It only requires reinterpretation of an existing timing constraint (tRP) and, potentially, the addition of a new timing constraint (explained in Section 3.5.1). Figure 3.8 (top) shows the performance benefit of SALP-1.

3.4.2. SALP-2: Subarray-Level-Parallelism-2

While SALP-1 pipelines the precharging and activation of different subarrays, the relative ordering between the two commands is still preserved. This is because existing DRAM banks do not allow two subarrays to be activated at the same time. As a result, the write-recovery latency (Section 3.2.2) of an activated subarray not only delays a `PRECHARGE` to itself, but also delays a subsequent `ACTIVATE` to another subarray. Based on the observation that the write-recovery latency is also local to a subarray, SALP-2 (our second mechanism) issues the `ACTIVATE` to another subarray *before* the `PRECHARGE` to the currently activated subarray. As a result, SALP-2 can overlap the write-recovery of the currently activated subarray with the activation of another subarray, further reducing the service time compared to SALP-1 (Figure 3.8, middle).

However, as highlighted in the figure, SALP-2 requires two subarrays to remain activated at the same time. This is not possible in existing DRAM banks as the global row-address latch, which determines the wordline in the bank that is raised, is shared by all the subarrays. In Section 3.5.2, we will show how to enable SALP-2 by eliminating this sharing.

3.4.3. MASA: Multitude of Activated Subarrays

Although SALP-2 allows two subarrays within a bank to be activated, it requires the controller to precharge one of them before issuing a column command (e.g., `READ`) to the bank. This is because when a bank receives a column command, all activated subarrays in the bank will connect their local row-buffers to the global bitlines. If more than one subarray is activated, this will result in a short circuit. As a result, SALP-2 cannot allow multiple subarrays to concurrently remain activated and serve column commands.

The key idea of MASA (our third mechanism) is to allow *multiple* subarrays to be activated at the same time, while allowing the memory controller to *designate* exactly one of the activated subarrays to drive the global bitlines during the next column command. MASA has two advantages over SALP-2. First, MASA overlaps the activation of different subarrays within a bank. Just before issuing a column command to any of the activated subarrays, the memory controller *designates* one particular subarray whose row-buffer should serve the column command. Second, MASA eliminates extra `ACTIVATE`s to the same row, thereby mitigating row-buffer thrashing. This is because the local row-buffers of multiple subarrays can remain activated at the same time without experiencing collisions on the global bitlines. As a result, MASA further improves performance compared to SALP-2 (Figure 3.8, bottom).

As indicated in the figure, to designate one of the multiple activated subarrays, the controller needs a new command, `SA_SEL` (*subarray-select*). In addition to the changes required by SALP-2, MASA requires a single-bit latch per subarray to denote whether a subarray is *designated* or not (Section 3.5.3).

3.5. Implementation

Our three proposed mechanisms assume that the memory controller is aware of the existence of subarrays (to be described in Section 3.5.4) and can determine which subarray a particular request accesses. All three mechanisms require reinterpretation of existing DRAM timing constraints or addition of new ones. SALP-2 and MASA also require small, non-intrusive modifications to the DRAM chip. In this section, we describe the changes required by each mechanism in detail.

3.5.1. SALP-1: Relaxing tRP

As previously described, SALP-1 overlaps the precharging of one subarray with the subsequent activation of another subarray. However, by doing so, SALP-1 violates the timing constraint *tRP* (row-precharge time) imposed between consecutive `PRECHARGE` and `ACTIVATE` commands to the same bank. The reason why tRP exists is to ensure that a previously activated subarray (Subarray X in Figure 3.9) has fully reached the precharged state before it can again be activated. Existing DRAM banks provide that guarantee by conservatively delaying an `ACTIVATE` to any subarray, even to a subarray that is not the one being precharged. But, for a subarray that is *already* in the precharged state (Subarray Y in Figure 3.9), it is safe to activate it while another subarray is being precharged. So, as long as consecutive `PRECHARGE` and `ACTIVATE` commands are to different subarrays, the `ACTIVATE` can be issued before tRP has been satisfied.[8]

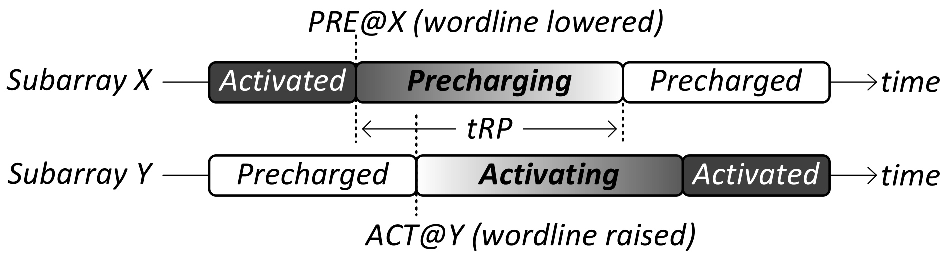


Figure 3.9. Relaxing tRP between two different subarrays.

Limitation of SALP-1. SALP-1 *cannot* overlap the write-recovery of one subarray with the activation of another subarray. This is because both write-recovery and activation require their corresponding wordline to remain raised for the entire duration of the corresponding operation. However, in existing DRAM banks, the global row-address latch determines the unique wordline within the bank that is raised (Section 3.2.3). Since this latch is shared across all subarrays, it is not possible to have two raised wordlines within a bank, even if they are in different subarrays. SALP-2 addresses this issue by adding

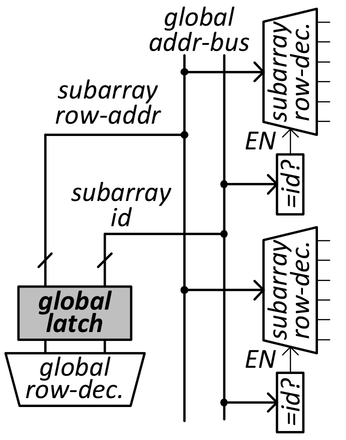
[8]We assume that it is valid to issue the two commands in consecutive DRAM cycles. Depending on vendor-specific microarchitecture, an additional precharge-to-activate timing constraint *tPA* ($<$ tRP) may be required.

row-address latches to each subarray.

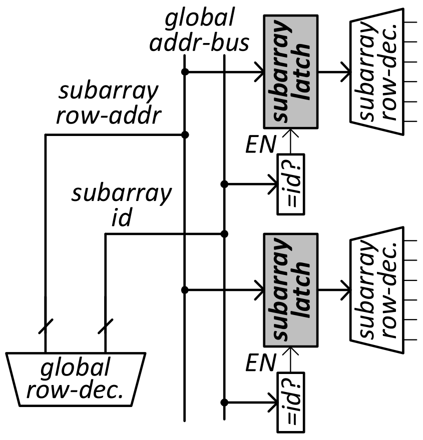
3.5.2. SALP-2: Per-Subarray Row-Address Latches

The goal of SALP-2 is to further improve performance compared to SALP-1 by overlapping the write-recovery latency of one subarray with the activation of another subarray. For this purpose, we propose two changes to the DRAM chip: *(i)* latched subarray row-decoding and *(ii)* selective precharging.

Latched Subarray Row-Decoding. The key idea of *latched subarray row-decoding* (LSRD) is to push the global row-address latch to individual subarrays such that each subarray has its own row-address latch, as shown in Figure 3.10. When an `ACTIVATE` is issued to a subarray, the subarray row-address is stored in the latch. This latch feeds the subarray row-decoder, which in turn drives the corresponding wordline within the subarray. Figure 3.11 shows the timeline of subarray activation with and without LSRD. Without LSRD, the global row-address bus is utilized by the subarray until it is precharged. This prevents the controller from activating another subarray. In contrast, with LSRD, the global address-bus is utilized only until the row-address is stored in the corresponding subarray's latch. From that point on, the latch drives the wordline, freeing the global address-bus to be used by another subarray.

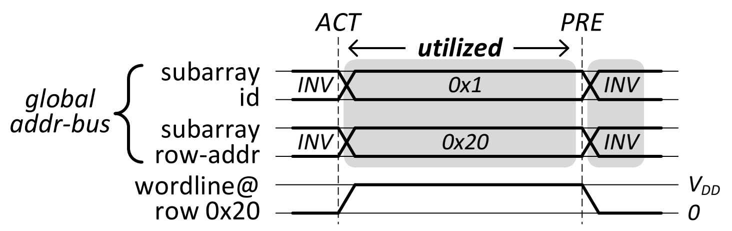


(a) Global row-address latch

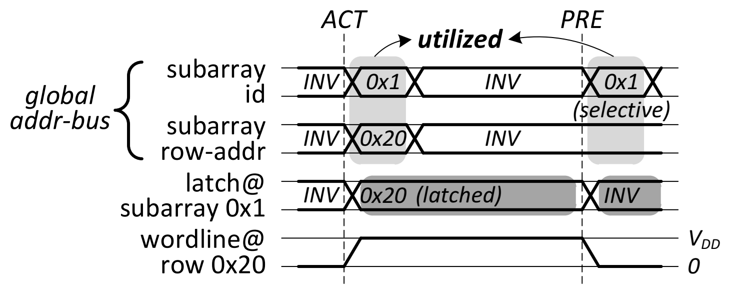


(b) Per-subarray row-address latch

Figure 3.10. SALP-2: Latched Subarray Row-Decoding



(a) Baseline: global row-address latch & global precharging



(b) SALP-2: Subarray row-address latch & selective precharging

Figure 3.11. Activating/precharging wordline-0x20 of subarray-0x1.

Selective Precharging. Since existing DRAMs do not allow a bank to have more than one raised wordline, a PRECHARGE is designed to lower *all* wordlines within a bank to zero voltage. In fact, the memory controller does not even specify a row address when it issues a PRECHARGE. A bank lowers all wordlines by broadcasting an INV (invalid) value on the global row-address bus.[9] However, when there are two activated subarrays (each with a raised wordline) SALP-2 needs to be able to *selectively precharge* only one of the subarrays. To achieve this, we require that PRECHARGEs be issued with the corresponding subarray ID. When a bank receives a PRECHARGE to a subarray, it places the subarray ID and INV (for the subarray row-address) on the global row-address bus. This ensures that only that specific subarray is precharged. Selective precharging requires the memory controller to remember the ID of the subarray to be precharged. This requires modest storage overhead at the memory controller – one subarray ID per bank.

Timing Constraints. Although SALP-2 allows two activated subarrays, no column command can be issued during that time. This is because a column command electrically

[9]When each subarray receives the INV values for both subarray ID and subarray row-address, it lowers all its wordlines and precharges all its bitlines.

connects the row-buffers of all activated subarrays to the global bitlines – leading to a short-circuit between the row-buffers. To avoid such hazards on the global bitlines, SALP-2 must wait for a column command to be processed before it can activate another subarray in the same bank. Hence, we introduce two new timing constraints for SALP-2: *tRA* (read-to-activate) and *tWA* (write-to-activate).

Limitation of SALP-2. As described above, SALP-2 requires a bank to have exactly one activated subarray when a column command is received. Therefore, SALP-2 cannot address the row-buffer thrashing problem.

3.5.3. MASA: Designating an Activated Subarray

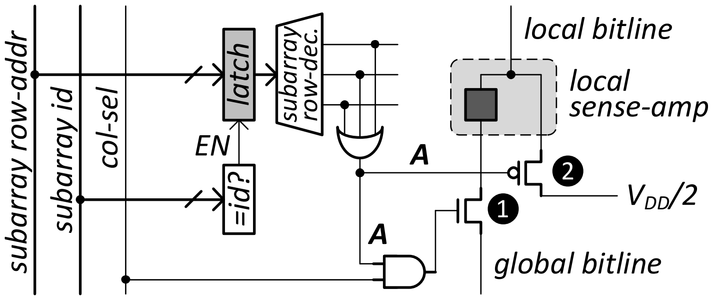
The key idea behind MASA is to allow multiple activated subarrays, but to ensure that only a single subarray's row-buffer is connected to the global bitlines on a column command. To achieve this, we propose the following changes to the DRAM microarchitecture in addition to those required by SALP-2: *(i)* addition of a *designated-bit latch* to each subarray, *(ii)* introduction of a new DRAM command, `SA_SEL` (subarray-select), and *(iii)* routing of a new global wire (subarray-select).

Designated-Bit Latch. In SALP-2 (and existing DRAM), an activated subarray's local sense-amplifiers are connected to the global bitlines on a column command. The connection between each sense-amplifier and the corresponding global bitline is established when an access transistor, ❶ in Figure 3.12a, is switched on. All such access transistors (one for each sense-amplifier) within a subarray are controlled by the same 1-bit signal, called *activated* (**A** in figure), that is raised only when the subarray has a raised wordline.[10] As a result, it is *not* possible for a subarray to be activated while at the same time be disconnected from the global bitlines on a column command.

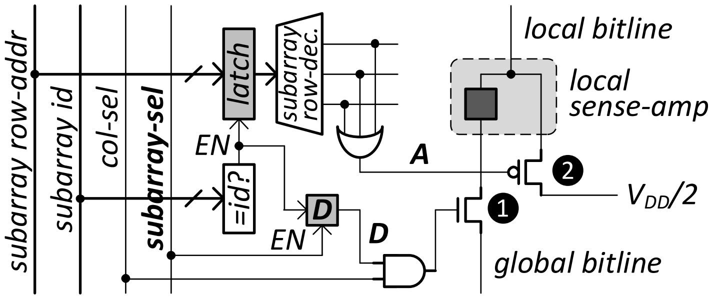
To enable MASA, we propose to decouple the control of the access transistor from the wordlines, as shown in Figure 3.12b. To this end, we propose a separate 1-bit signal, called

[10]The *activated* signal can be abstracted as a logical *OR* across all wordlines in the subarray, as shown in Figure 3.12a. The exact implementation of the signal is microarchitecture-specific.

designated (**D** in figure), to control the transistor independently of the wordlines. This signal is driven by a *designated-bit latch*, which must be set by the memory controller in order to enable a subarray's row-buffer to be connected to the global bitlines. To access data from one particular activated subarray, the memory controller sets the designated-bit latch of the subarray and clears the designated-bit latch of all other subarrays. As a result, MASA allows multiple subarrays to be activated within a bank while ensuring that one subarray (the *designated* one) can at the same time serve column commands. Note that MASA still requires the *activated* signal to control the precharge transistors ❷ that determine whether or not the row-buffer is in the precharged state (i.e., connecting the local bitlines to $\frac{1}{2}V_{DD}$).



(a) SALP-2: **A**ctivated subarray is connected to global bitlines



(b) MASA: **D**esignated subarray is connected to global bitlines

Figure 3.12. MASA: Designated-bit latch and subarray-select signal

Subarray-Select Command. To allow the memory controller to selectively set and clear the designated-bit of any subarray, MASA requires a new DRAM command, which we call `SA_SEL` (subarray-select). To set the designated-bit of a particular subarray, the controller issues a `SA_SEL` along with the row-address that corresponds to the raised wordline within the subarray. Upon receiving this command, the bank sets the designated-bit for only the subarray and clears the designated-bits of all other subarrays. After this operation, all subsequent column commands are served by the designated subarray.

To update the designated-bit latch of each subarray, MASA requires a new global control signal that acts as a strobe for the latch. We call this signal *subarray-select*. When a bank receives the `SA_SEL` command, it places the corresponding subarray ID and subarray row-address on the global address-bus and briefly raises the subarray-select signal. At this point, the subarray whose ID matches the ID on the global address-bus will set its designated-bit, while all other subarrays will clear their designated-bit. Note that `ACTIVATE` also sets the designated-bit for the subarray it activates, as it expects the subarray to serve all subsequent column commands. In fact, from the memory controller's perspective, `SA_SEL` is the same as `ACTIVATE`, except that for `SA_SEL`, the supplied row-address corresponds to a wordline that is already raised.

Timing Constraints. Since designated-bits determine which activated subarray will serve a column command, they should not be updated (by `ACTIVATE/SA_SEL`) while a column command is in progress. For this purpose, we introduce two timing constraints called tRA (read-to-activate/select) and tWA (write-to-activate/select). These are the same timing constraints introduced by SALP-2.

Additional Storage at the Controller. To support MASA, the memory controller must track the status of all subarrays within each bank. A subarray's status represents *(i)* whether the subarray is activated, *(ii)* if so, which wordline within the subarray is raised, and *(iii)* whether the subarray is designated to serve column commands. For the system configurations we evaluate (Section 3.8), maintaining this information incurs a storage overhead of less than 256 bytes at the memory controller.

While MASA overlaps multiple `ACTIVATE`s to the same bank, it must still obey timing constraints such as tFAW and tRRD that limit the rate at which `ACTIVATE`s are issued to the entire DRAM chip. We evaluate the power and area overhead of our three proposed mechanisms in Section 3.6.

3.5.4. Exposing Subarrays to the Memory Controller

For the memory controller to employ our proposed schemes, it requires the following three pieces of information: *(i)* the number of subarrays per bank, *(ii)* whether the DRAM supports SALP-1, SALP-2 and/or MASA, and *(iii)* the values for the timing constraints tRA and tWA. Since these parameters are heavily dependent on vendor-specific microarchitecture and process technology, they may be difficult to standardize. Therefore, we describe an alternate way of exposing these parameters to the memory controller.

Serial Presence Detect. Multiple DRAM chips are assembled together on a circuit board to form a DRAM module. On every DRAM module lies a separate 256-byte EEPROM, called the *serial presence detect* (SPD), which contains information about both the chips and the module, such as timing, capacity, organization, etc. [66]. At system boot time, the SPD is read by the BIOS, so that the memory controller can correctly issue commands to the DRAM module. In the SPD, more than a hundred extra bytes are set aside for use by the manufacturer and the end-user [66]. This storage is more than sufficient to store subarray-related parameters required by the controller.

Number of Subarrays per Bank. The number of subarrays within a bank is expected to increase for larger capacity DRAM chips that have more rows. However, certain manufacturing constraints may prevent all subarrays from being accessed in parallel. To increase DRAM yield, every subarray is provisioned with a few spare rows that can replace faulty rows [60, 80]. If a faulty row in one subarray is mapped to a spare row in another subarray, then the two subarrays can no longer be accessed in parallel. To strike a tradeoff between high yield and the number of subarrays that can be accessed in parallel, spare rows in each subarray can be restricted to replace faulty rows only within a subset of the other subarrays. With this guarantee, the memory controller can still apply our mechanisms to different subarray groups. In our evaluations (Section 3.9.2), we show that just having 8 subarray groups can provide significant performance improvements. From now on, we refer to an independently accessible subarray group as a "subarray."

3.6. Power & Area Overhead

Of our three proposed schemes, SALP-1 does not incur any additional area or power overhead since it does not make any modifications to the DRAM structure. On the other hand, SALP-2 and MASA require subarray row-address latches that minimally increase area and power. MASA also consumes additional static power due to multiple activated subarrays and additional dynamic power due to extra `SA_SEL` commands. We analyze these overheads in this section.

3.6.1. Additional Latches

SALP-2 and MASA add a subarray row-address latch to each subarray. While MASA also requires an additional single-bit latch for the designated-bit, its area and power overheads are insignificant compared to the subarray row-address latches. In most of our evaluations, we assume 8 subarrays-per-bank and 8 banks-per-chip. As a result, a chip requires a total of 64 row-address latches, where each latch stores the 40-bit partially pre-decoded row-address.[11] Scaling the area from a previously proposed latch design [90] to 55nm process technology, each row-address latch occupies an area of 42.9μm^2. Overall, this amounts to a 0.15% area overhead compared to a 2Gb DRAM chip fabricated using 55nm technology (die area = 73mm^2 [128]). Similarly, normalizing the latch power consumption to 55nm technology and 1.5V operating voltage, a 40-bit latch consumes 72.2μW additional power for each `ACTIVATE`. This is negligible compared to the activation power, 51.2mW (calculated using DRAM models [104, 128, 157]).

[11]A 2Gb DRAM chip with 32k rows has a 15-bit row-address. We assume 3:8 pre-decoding, which yields a 40-bit partially pre-decoded row-address.

3.6.2. Multiple Activated Subarrays

To estimate the additional static power consumption of multiple activated subarrays, we compute the difference in the maximum current between the cases when *all* banks are activated (I_{DD3N}, 35mA) and when *no* bank is activated (I_{DD2N}, 32mA) [107]. For a DDR3 chip which has 8 banks and operates at 1.5V, each activated local row-buffer consumes at most 0.56mW additional static power in the steady state. This is small compared to the baseline static power of 48mW per DRAM chip.

3.6.3. Additional `SA_SEL` Commands

To switch between multiple activated subarrays, MASA issues additional `SA_SEL` commands. Although `SA_SEL` is the same as `ACTIVATE` from the memory controller's perspective (Section 3.5.3), internally, `SA_SEL` does not involve the subarray core, i.e., a subarray's cells. Therefore, we estimate the dynamic power of `SA_SEL` by subtracting the subarray core's power from the dynamic power of `ACTIVATE`, where the subarray core's power is the sum of the wordline and row-buffer power during activation [128]. Based on our analysis using DRAM modeling tools [104, 128, 157], we estimate the power consumption of `SA_SEL` to be 49.6% of `ACTIVATE`. MASA also requires a global subarray-select wire in the DRAM chip. However, compared to the large amount of global routing that is already present within a bank (40 bits of partially pre-decoded row-address and 1024 bits of fully decoded column-address), the overhead of one additional wire is negligible.

3.6.4. Comparison to Expensive Alternatives

As a comparison, we present the overhead incurred by two alternative approaches that can mitigate bank conflicts: *(i)* increasing the number of DRAM banks and *(ii)* adding an SRAM cache inside the DRAM chip.

More Banks. To add more banks, per-bank circuit components such as the global decoders and I/O-sense amplifiers must be replicated [60]. This leads to significant increase

Processor	1-16 cores, 5.3GHz, 3-wide issue, 8 MSHRs, 128-entry instruction window
Last-Level Cache	64B cache-line, 16-way associative, 512kB private cache-slice per core
Memory Controller	64/64-entry read/write request queues per controller, FR-FCFS scheduler [130, 172], writes are scheduled in batches [27, 94, 142]
Memory	Timing: DDR3-1066 (8-8-8) [107], tRA (4tCK), tWA (14tCK) Organization (default in bold): **1**-8 channels, **1**-8 ranks-per-channel, **8**-64 banks-per-rank, 1-**8**-128 subarrays-per-bank

Table 3.2. Configuration of simulated system

in DRAM chip area. Using the DRAM area model from Rambus [128, 157], we estimate that increasing the number of banks from 8 to 16, 32, and 64, increases the chip area by 5.2%, 15.5%, and 36.3%, respectively. Larger chips also consume more static power.

Additional SRAM Cache. Adding a separate SRAM cache within the DRAM chip (such "Cached DRAM" proposals are discussed in Section 3.7), can achieve similar benefits as utilizing multiple row-buffers across subarrays. However, this increases the DRAM chip area and, consequently, its static power consumption. We calculate the chip area penalty for adding SRAM caches using CACTI-D [149]. An SRAM cache that has a size of 8 Kbits (same as a row-buffer), 64 Kbits, and 512 Kbits increases DRAM chip area by 0.6%, 5.0%, and 38.8%, respectively. These figures do not include the additional routing logic that is required between the I/O sense-amplifiers and the SRAM cache.

3.7. Related Work

In this chapter, we propose three schemes that exploit the existence of subarrays within DRAM banks to mitigate the negative effects of bank conflicts. Prior works proposed increasing the performance and energy-efficiency of DRAM through approaches such as DRAM module reorganization, changes to DRAM chip design, and memory controller optimizations.

DRAM Module Reorganization. Threaded Memory Module [158], Multicore DIMM [5], and Mini-Rank [171] are all techniques that partition a DRAM rank (and the DRAM data-bus) into multiple rank-subsets [4], each of which can be operated independently. Although partitioning a DRAM rank into smaller rank-subsets increases parallelism, it narrows the data-bus of each rank-subset, incurring longer latencies to transfer a 64 byte cache-line. Fore example, having 8 mini-ranks increases the data-transfer latency by 8 times (to 60 ns, assuming DDR3-1066) for all memory accesses. In contrast, our schemes increase parallelism without increasing latency. Furthermore, having many rank-subsets requires a correspondingly large number of DRAM chips to compose a DRAM rank, an assumption that does not hold in mobile DRAM systems where a rank may consist of as few as two chips [106]. However, since the parallelism exposed by rank-subsetting is orthogonal to our schemes, rank-subsetting can be combined with our schemes to further improve performance.

Changes to DRAM Design. Cached DRAM organizations, which have been widely proposed [34, 44, 46, 49, 79, 121, 135, 159, 170] augment DRAM chips with an additional SRAM cache that can store recently accessed data. Although such organizations reduce memory access latency in a manner similar to MASA, they come at increased chip area and design complexity (as Section 3.6.4 showed). Furthermore, cached DRAM only provides parallelism when accesses hit in the SRAM cache, while serializing cache misses that access the same DRAM bank. In contrast, our schemes parallelize DRAM bank accesses while incurring significantly lower area and logic complexity.

Since a large portion of the DRAM latency is spent driving the local bitlines [109], Fujitsu's FCRAM and Micron's RLDRAM proposed to implement shorter local bitlines (i.e., fewer cells per bitline) that are quickly drivable due to their lower capacitances. However, this significantly increases the DRAM die size (30-40% for FCRAM [136], 40-80% for RLDRAM [80]) because the large area of sense-amplifiers is amortized over a smaller number of cells.

A patent by Qimonda [125] proposed the high-level notion of separately addressable sub-banks, but it lacks concrete mechanisms for exploiting the independence between sub-banks. In the context of embedded DRAM, Yamauchi et al. proposed the Hierarchical Multi-Bank (HMB) [164] that parallelizes accesses to different subarrays in a fine-grained manner. However, their scheme adds complex logic to all subarrays. For example, each subarray requires a timer that automatically precharges a subarray after an access. As a result, HMB cannot take advantage of multiple row-buffers.

Although only a small fraction of the row is needed to serve a memory request, a DRAM bank wastes power by always activating an entire row. To mitigate this "overfetch" problem and save power, Udipi et al. [152] proposed two techniques (SBA and SSA).[12] In SBA, global wordlines are segmented and controlled separately so that tiles in the horizontal direction are not activated in lockstep, but selectively. However, this increases DRAM chip area by 12-100% [152]. SSA combines SBA with chip-granularity rank-subsetting to achieve even higher energy savings. But, both SBA and SSA increase DRAM latency, more significantly so for SSA (due to rank-subsetting).

A DRAM chip experiences bubbles in the data-bus, called the bus-turnaround penalty (*tWTR* and *tRTW* in Table 3.1), when transitioning from serving a write request to a read request, and vice versa [27, 94, 142]. During the bus-turnaround penalty, Chatterjee et al. [27] proposed to internally "prefetch" data for subsequent read requests into extra registers that are added to the DRAM chip.

An IBM patent [89] proposed latched row-decoding to activate multiple wordlines in a DRAM bank simultaneously, in order to expedite the testing of DRAM chips by checking for defects in multiple rows at the same time.

Memory Controller Optimizations. To reduce bank conflicts and increase row-buffer locality, Zhang et al. proposed to randomize the bank address of memory requests by XOR hashing [169]. Sudan et al. proposed to improve row-buffer locality by placing frequently referenced data together in the same row [143]. Both proposals can be combined with our schemes to further improve parallelism and row-buffer locality.

[12] Udipi et al. use the term subarray to refer to an individual tile.

Prior works have also proposed memory scheduling algorithms (e.g., [33, 58, 85, 86, 114, 117, 118, 122]) that prioritize certain favorable requests in the memory controller to improve system performance and/or fairness. Subarrays expose more parallelism to the memory controller, increasing the controller's flexibility to schedule requests.

3.8. Evaluation Methodology

We developed a cycle-accurate DDR3-SDRAM simulator that we validated against Micron's Verilog behavioral model [108] and DRAMSim2 [131]. We use this memory simulator as part of a cycle-level in-house x86 multi-core simulator, whose front-end is based on Pin [100]. We calculate DRAM dynamic energy consumption by associating an energy cost with each DRAM command, derived using the tools [104, 128, 157] and the methodology as explained in Section 3.6.[13]

Unless otherwise specified, our default system configuration comprises a single-core processor with a memory subsystem that has 1 channel, 1 rank-per-channel (RPC), 8 banks-per-rank (BPR), and 8 subarrays-per-bank (SPB). We also perform detailed sensitivity studies where we vary the numbers of cores, channels, ranks, banks, and subarrays. More detail on the simulated system configuration is provided in Table 3.2.

We use *line-interleaving* to map the physical address space onto the DRAM hierarchy (channels, ranks, banks, etc.). In line-interleaving, small chunks of the physical address space (often the size of a cache-line) are striped across different banks, ranks, and channels. Line-interleaving is utilized to maximize the amount of memory-level parallelism and is employed in systems such as Intel Nehalem [55], Sandy Bridge [54], Sun OpenSPARC T1 [144], and IBM POWER7 [139]. We use the *closed-row policy* in which the memory controller precharges a bank when there are no more outstanding requests to the activated row of that bank. The closed-row policy is often used in conjunction with

[13]We consider dynamic energy dissipated by only the DRAM chip itself and do not include dynamic energy dissipated at the channel (which differs on a motherboard-by-motherboard basis).

line-interleaving since row-buffer locality is expected to be low. Additionally, we also show results for *row-interleaving* and the *open-row policy* in Section 3.9.3.

We use 32 benchmarks from SPEC CPU2006, TPC [151], and STREAM [141], in addition to a random-access microbenchmark similar in behavior to HPCC RandomAccess [48]. We form multi-core workloads by randomly choosing from only the benchmarks that access memory at least once every 1000 instructions. We simulate all benchmarks for 100 million instructions. For multi-core evaluations, we ensure that even the slowest core executes 100 million instructions, while other cores still exert pressure on the memory subsystem. To measure performance, we use instruction throughput for single-core systems and *weighted speedup* [140] for multi-core systems. We report results that are averaged across all 32 benchmarks for single-core evaluations and averaged across 16 different workloads for each multi-core system configuration.

3.9. Results

3.9.1. Individual Benchmarks (Single-Core)

Figure 3.13 shows the performance improvement of SALP-1, SALP-2, and MASA on a system with 8 subarrays-per-bank over a subarray-oblivious baseline. The figure also shows the performance improvement of an "Ideal" scheme which is the subarray-oblivious baseline with 8 times as many banks (this represents a system where all subarrays are fully independent). We draw two conclusions. First, SALP-1, SALP-2 and MASA consistently perform better than baseline for all benchmarks. On average, they improve performance by 6.6%, 13.4%, and 16.7%, respectively. Second, MASA captures most of the benefit of the "Ideal," which improves performance by 19.6% compared to baseline.

The difference in performance improvement across benchmarks can be explained by a combination of factors related to their individual memory access behavior. First, subarray-level parallelism in general is most beneficial for memory-intensive benchmarks that frequently access memory (benchmarks located towards the right of Figure 3.13). By increas-

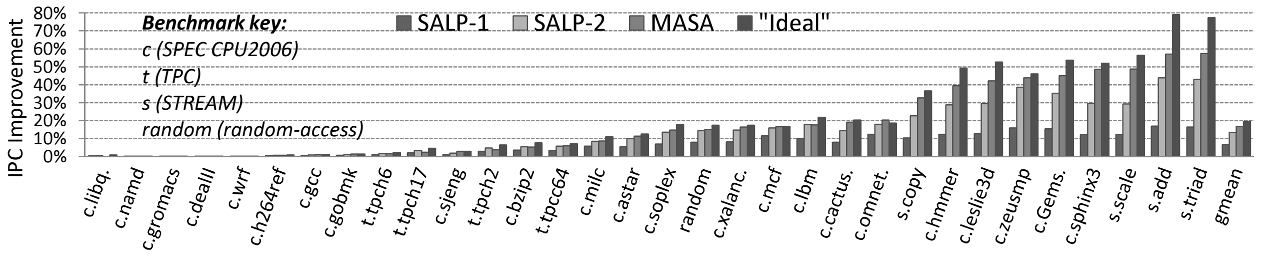


Figure 3.13. IPC improvement over the conventional subarray-oblivious baseline

ing the memory throughput for such applications, subarray-level parallelism significantly alleviates their memory bottleneck. The average memory-intensity of the rightmost applications (i.e., those that gain >5% performance with SALP-1) is 18.4 MPKI (last-level cache misses per kilo-instruction), compared to 1.14 MPKI of the leftmost applications.

Second, the advantage of SALP-2 is large for applications that are write-intensive. For such applications, SALP-2 can overlap the long write-recovery latency with the activation of a subsequent access. In Figure 3.13, the three applications (that improve more than 38% with SALP-2) are among both the most memory-intensive (>25 MPKI) and the most write-intensive (>15 WMPKI).

Third, MASA is beneficial for applications that experience frequent bank conflicts. For such applications, MASA parallelizes accesses to different subarrays by concurrently activating multiple subarrays (`ACTIVATE`) and allowing the application to switch between the activated subarrays at low cost (`SA_SEL`). Therefore, the subarray-level parallelism offered by MASA can be gauged by the `SA_SEL`-to-`ACTIVATE` ratio. For the nine applications that benefit more than 30% from MASA, on average, one `SA_SEL` was issued for every two `ACTIVATEs`, compared to one-in-seventeen for all the other applications. For a few benchmarks, MASA performs slightly worse than SALP-2. The baseline scheduling algorithm used with MASA tries to overlap as many `ACTIVATEs` as possible and, in the process, inadvertently delays the column command of the most critical request which slightly degrades performance for these benchmarks.[14]

[14]For one benchmark, MASA performs slightly better than the "Ideal" due to interactions with the scheduler.

3.9.2. Sensitivity to Number of Subarrays

With more subarrays, there is greater opportunity to exploit subarray-level parallelism and, correspondingly, the improvements provided by our schemes also increase. As the number of subarrays-per-bank is swept from 1 to 128, Figure 3.14 plots the IPC improvement, average read latency,[15] and memory-level parallelism[16] of our three schemes (averaged across 32 benchmarks) compared to the subarray-oblivious baseline.

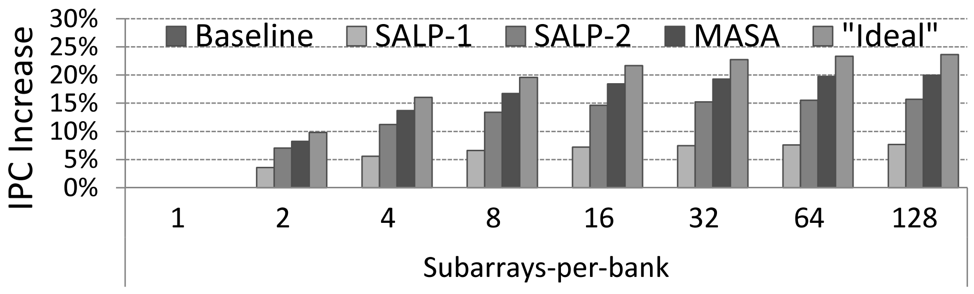
Figure 3.14a shows that SALP-1, SALP-2, and MASA consistently improve IPC as the number of subarrays-per-bank increases. But, the gains are diminishing because most of the bank conflicts are parallelized for even a modest number of subarrays. Just 8 subarrays-per-bank captures more than 80% of the IPC improvement provided by the same mechanism with 128 subarrays-per-bank. The performance improvements of SALP-1, SALP-2, and MASA are a direct result of reduced memory access latency and increased memory-level parallelism, as shown in Figures 3.14b and 3.14c, respectively. These improvements are two-sides of the same coin: by increasing the parallelism across subarrays, our mechanisms are able to overlap the latencies of multiple memory requests to reduce the average memory access latency.

3.9.3. Sensitivity to System Configuration

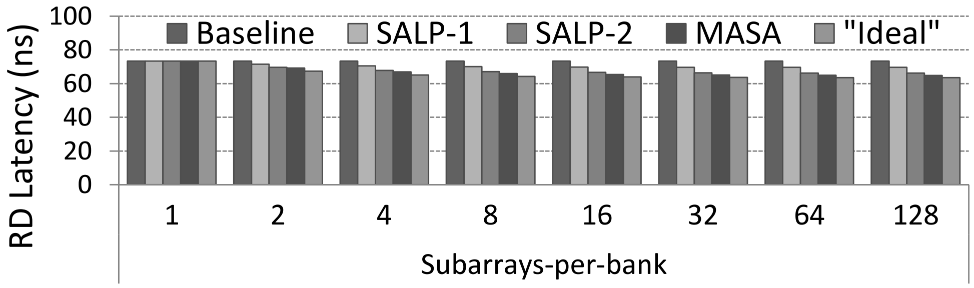
Mapping and Row Policy. In *row-interleaving*, as opposed to line-interleaving, a contiguous chunk of the physical address space is mapped to each DRAM row. Row-interleaving is commonly used in conjunction with the *open-row policy* so that a row is never eagerly closed – a row is left open in the row-buffer until another row needs to be accessed. Figure 3.15 shows the results (averaged over 32 benchmarks) of employing our three schemes on a row-interleaved, open-row system.

[15]Average memory latency for read requests, which includes: *(i)* queuing delay at the controller, *(ii)* bank access latency, and *(iii)* data-transfer latency.

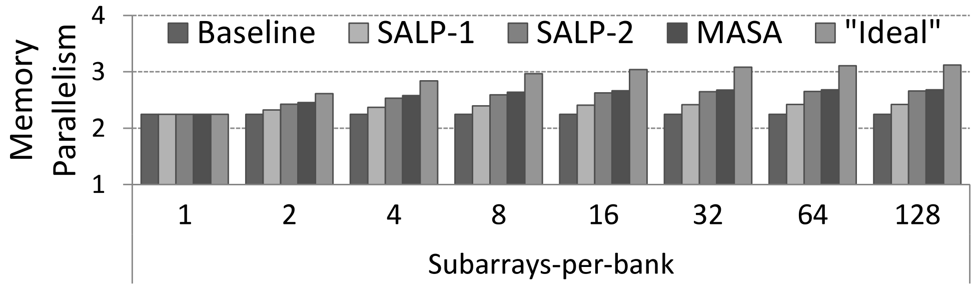
[16]The average number of requests that are *being served*, given that there is at least one such request. A request is defined as *being served* from when the first command is issued on its behalf until its data-transfer has completed.



(a) IPC improvement



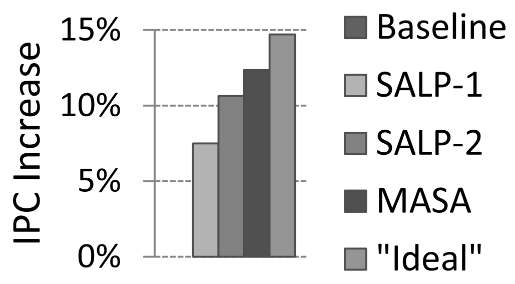
(b) Average read latency



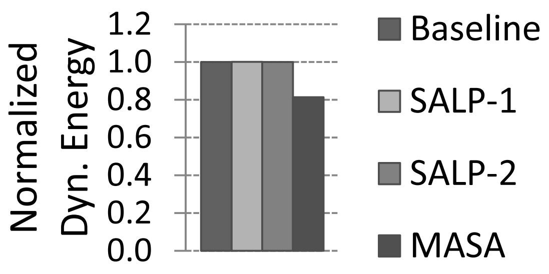
(c) Memory-level parallelism

Figure 3.14. Sensitivity to number of subarrays-per-bank

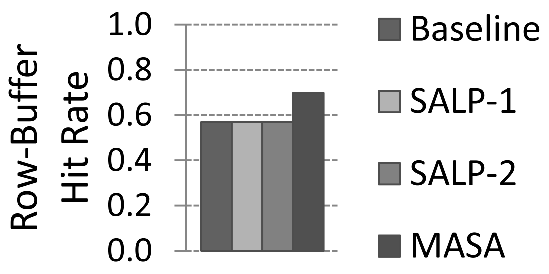
As shown in Figure 3.15a, the IPC improvements of SALP-1, SALP-2, and MASA are 7.5%, 10.6%, and 12.3%, where MASA performs nearly as well as the "Ideal" (14.7%). However, the gains are lower than compared to a line-interleaved, closed-row system. This is because the subarray-oblivious baseline performs better on a row-interleaved, open-row system (due to row-buffer locality), thereby leaving less headroom for our schemes to improve performance. MASA also improves DRAM energy-efficiency in a row-interleaved system. Figure 3.15b shows that MASA decreases DRAM dynamic energy consumption by 18.6%. Since MASA allows multiple row-buffers to remain activated, it increases the row-buffer hit rate by 12.8%, as shown in Figure 3.15c. This is clear from Figure 3.15d, which shows that 50.1% of the `ACTIVATEs` issued in the baseline are converted to `SA_SELs` in MASA.



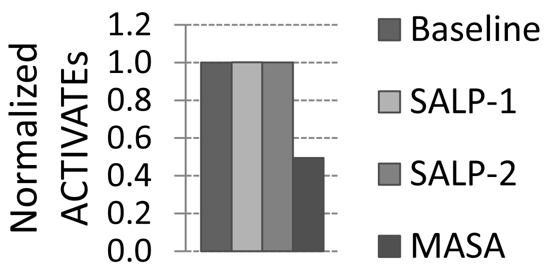
(a) IPC improvement



(b) Dynamic DRAM energy



(c) Row-buffer hit rate



(d) Number of issued `ACTIVATEs`

Figure 3.15. Row-interleaving and open-row policy.

Number of Channels, Ranks, Banks. Even for highly provisioned systems with unrealistically large numbers of channels, ranks, and banks, exploiting subarray-level parallelism improves performance significantly, as shown in Figure 3.16. This is because even such systems cannot completely remove all bank conflicts due to the well-known birthday paradox: even if there were 365 banks (very difficult to implement), with just 23 concurrent memory requests, the probability of a bank conflict between any two requests is more than 50% (for 64 banks, only 10 requests are required). Therefore, exploiting subarray-level parallelism still provides performance benefits. For example, while an 8-channel baseline system provides more than enough memory bandwidth (<4% data-bus utilization), MASA reduces access latency by parallelizing bank conflicts, and improves performance by 8.6% over the baseline.

As more ranks/banks are added to the same channel, increased contention on the data-bus is likely to be the performance limiter. That is why adding more ranks/banks does not provide as large benefits as adding more channels (Figure 3.16).[17] Ideally, for the highest performance, one would increase the numbers of all three: channels/ranks/banks. However, as explained in Section 3.1, adding more channels is very expensive, whereas the number of ranks-/banks-per-channel is limited to a low number in modern high-frequency DRAM systems. Therefore, exploiting subarray-level parallelism is a cost-effective way of achieving the performance of many ranks/banks and, as a result, extracting the most performance from a given number of channels.

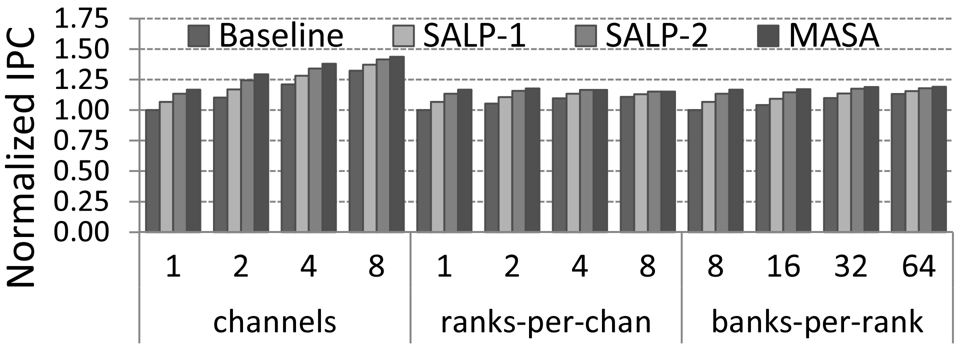


Figure 3.16. Memory configuration sweep (line-interleaved, closed-row). IPC normalized to: 1-channel, 1-RPC, 8-BPR, 8-SPB.

Number of Cores. As shown in Figure 3.17, our schemes improve performance of 8-core and 16-core systems with the FR-FCFS memory scheduler [130, 172]. However, previous studies have shown that destructive memory interference among applications due to FR-FCFS scheduling can severely degrade system performance [113, 117]. Therefore, to exploit the full potential of subarray-level parallelism, the scheduler should resolve bank conflicts in an application-aware manner. To study this effect, we evaluate our schemes with TCM [86], a state-of-the-art scheduler that mitigates inter-application interference. As shown in Figure 3.17, TCM outperforms FR-FCFS by 3.7%/12.3% on 8-core/16-core systems. When employed with the TCM scheduler, SALP-1/SALP-2/MASA further improve performance by 3.9%/5.9%/7.4% on the 8-core system and by 2.5%/3.9%/8.0% on

[17]Having more ranks (as opposed to having just more banks) aggravates data-bus contention by introducing bubbles in the data-bus due to *tRTRS* (rank-to-rank switch penalty).

the 16-core system. We also observe similar trends for systems using row-interleaving and the open-row policy (not shown due to space constraints). We believe that further performance improvements are possible by designing memory request scheduling algorithms that are both application-aware and subarray-aware.

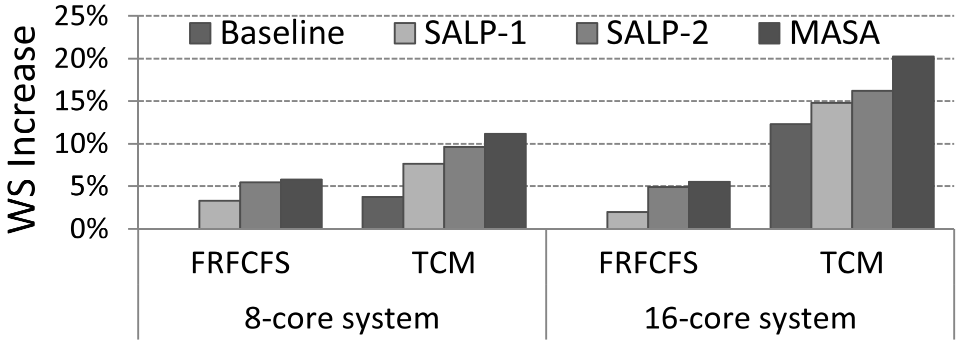


Figure 3.17. Multi-core weighted speedup improvement. Configuration: 2-channel, 2-RPC, line-interleaved, closed-row policy.

3.10. Chapter Summary

We introduced new techniques that exploit the existence of subarrays within a DRAM bank to mitigate the performance impact of bank conflicts. Our mechanisms are built on the key observation that subarrays within a DRAM bank operate largely independently and have their own row-buffers. Hence, the latencies of accesses to different subarrays within the same bank can potentially be overlapped to a large degree. We introduce three schemes that take advantage of this fact and progressively increase the independence of operation of subarrays by making small modifications to the DRAM chip. Our most sophisticated scheme, MASA, enables *(i)* multiple subarrays to be accessed in parallel, and *(ii)* multiple row-buffers to remain activated at the same time in different subarrays, thereby improving both memory-level parallelism and row-buffer locality. We show that our schemes significantly improve system performance on both single-core and multi-core systems on a variety of workloads while incurring little (<0.15%) or no area overhead in the DRAM chip. Our techniques can also improve memory energy efficiency. We conclude that exploiting subarray-level parallelism in a DRAM bank can be a promising and cost-effective method for overcoming the negative effects of DRAM bank conflicts, without paying the large cost of increasing the number of banks in the DRAM system.

Chapter 4

Ramulator: A Fast and Extensible DRAM Simulator

In recent years, we have witnessed a flurry of new proposals for DRAM interfaces and organizations. As listed in Table 4.1, some were evolutionary upgrades to existing standards (e.g., DDR4, LPDDR4), while some were pioneering implementations of die-stacking (e.g., WIO, HMC, HBM), and still others were academic research projects in experimental stages (e.g., Udipi et al. [152], Kim et al. [87]).

Segment	*DRAM Standards & Architectures*
Commodity	DDR3 (2007) [62]; DDR4 (2012) [69]
Low-Power	LPDDR3 (2012) [67]; LPDDR4 (2014) [72]
Graphics	GDDR5 (2009) [63]
Performance	eDRAM [102, 120]; RLDRAM3 (2011) [105]
3D-Stacked	WIO (2011) [65]; WIO2 (2014) [73]; MCDRAM (2015) [61]; HBM (2013) [70]; HMC1.0 (2013) [51]; HMC1.1 (2014) [52]
Academic	SBA/SSA (2010) [152]; Staged Reads (2012) [27]; RAIDR (2012) [98]; SALP (2012) [87]; TL-DRAM (2013) [96]; RowClone (2013) [138]; Half-DRAM (2014) [168]; Row-Buffer Decoupling (2014) [124]; SARP (2014) [24]; AL-DRAM (2015) [95]

Table 4.1. Landscape of DRAM-based memory

At the forefront of such innovations should be *DRAM simulators*, the software tool with which to evaluate the strengths and weaknesses of each new proposal. However, DRAM simulators have been lagging behind the rapid-fire changes to DRAM. For example, two of the most popular simulators (DRAMSim2 [131] and USIMM [26]) provide support for only one or two DRAM standards (DDR2 and/or DDR3), as listed in Table 4.2. Although these simulators are well suited for their intended standard(s), they were not explicitly designed to support a wide variety of standards with different organization and behavior. Instead, the simulators are implemented in a way that the specific details of one standard are integrated tightly into their codebase. As a result, researchers — especially those who are not intimately familiar with the details of an existing simulator — may find it cumbersome to implement and evaluate new standards on such simulators.

Type	*Simulator*	*DRAM Standards*
Standalone	DRAMSim2 (2011) [131]	DDR2, DDR3
	USIMM (2012) [26]	DDR3
	DrSim (2012) [74]	DDR2, DDR3, LPDDR2
	NVMain (2012) [127]	DDR3, LPDDR3, LPDDR4
Integrated	GPGPU-Sim (2009) [14]	GDDR3, GDDR5
	McSimA+ (2013) [6]	DDR3
	gem5 (2014) [43]	DDR3,* LPDDR3,* WIO*

**Not cycle-accurate* [43].

Table 4.2. Survey of popular DRAM simulators

The lack of an easy-to-extend DRAM simulator is an impediment to both industrial evaluation and academic research. Ultimately, it hinders the speed at which different points in the DRAM design space can be explored and studied. As a solution, we propose *Ramulator*, a fast and versatile DRAM simulator that treats extensibility as a first-class citizen. Ramulator is based on the important observation that DRAM can be abstracted as a *hierarchy* of state-machines, where the *behavior* of each state-machine — as well as the aforementioned hierarchy itself — is dictated by the DRAM standard in question. From any given DRAM standard, Ramulator extracts the full specification for the hierarchy and behavior, which is then entirely consolidated into just a single class (e.g., `DDR3.h/cpp`). On the other hand, Ramulator also provides a standard-agnostic state-machine (i.e., `DRAM.h`), which is capable of being paired with any standard (e.g., `DDR3.h/cpp` or `DDR4.h/cpp`) to take on its particular hierarchy and behavior. In essence, Ramulator enables the flexibility to reconfigure DRAM for different standards at compile-time, instead of laboriously hardcoding different configurations of DRAM for different standards.

The distinguishing feature of Ramulator lies in its modular design. More specifically, Ramulator decouples the logic for querying/updating the state-machines from the implementation specifics of any particular DRAM standard. As far as we know, such decoupling has not been achieved in previous DRAM simulators. Internally, Ramulator is structured around a collection of lookup-tables (Section 4.1.3), which are computationally inexpensive to query and update. This allows Ramulator to have the shortest runtime, outperforming other standalone simulators, shown in Table 4.2, by 2.5× (Section 4.3.2). Below, we summarize the key features of Ramulator, as well as its major contributions.

- Ramulator is an extensible DRAM simulator providing cycle-accurate performance models for a wide variety of standards: DDR3/4, LPDDR3/4, GDDR5, WIO1/2, HBM, SALP, AL-DRAM, TL-DRAM, RowClone, and SARP. Ramulator's modular design naturally lends itself to being augmented with additional standards. For some of the standards, Ramulator is capable of reporting power consumption by relying on DRAMPower [22] as the backend.

- Ramulator is portable and easy to use. It is equipped with a simple memory controller which exposes an external API for sending and receiving memory requests. Ramulator is available in two different formats: one for standalone usage and the other for integrated usage with gem5 [16]. Ramulator is written in C++11 and is released under the permissive BSD-license [2].

4.1. Ramulator: High-Level Design

Without loss of generality, we describe the high-level design of Ramulator through a case-study of modeling the widespread DDR3 standard. Throughout this section, we assume a working knowledge of DDR3, otherwise referring the reader to literature [62]. In Section 4.1.1, we explain how Ramulator employs a reconfigurable tree for modeling the *hierarchy* of DDR3. In Section 4.1.2, we describe the tree's nodes, which are reconfigurable state-machines for modeling the *behavior* of DDR3. Finally, Section 4.1.3 provides a closer look at the state-machines, revealing some of their implementation details.

4.1.1. Hierarchy of State-Machines

In Code 1 (left), we present the `DRAM` class, which is Ramulator's generalized template for building a hierarchy (i.e., tree) of state-machines (i.e., nodes). An instance of the `DRAM` class is a node in a tree of many other nodes, as is evident from its pointers to its parent node and children nodes in Code 1 (left, lines 4–6). Importantly, for the sake of modeling DDR3, we specialize the `DRAM` class for the `DDR3` class, which is shown in Code 1 (right). An instance of the resulting specialized class (`DRAM<DDR3>`) is then able to assume one of the five *levels* that are defined by the `DDR3` class.

```
 1  // DRAM.h
 2  template <typename T>
 3  class DRAM {
 4      DRAM<T>* parent;
 5      vector<DRAM<T>*> children;
 6      T::Level level;
 7      int index;
 8
 9      // more code...
10  };
```

```
 1  // DDR3.h/cpp
 2  class DDR3 {
 3      enum class Level {
 4          Channel, Rank,
 5          Bank, Row,
 6          Column, MAX
 7      };
 8
 9    // more code...
10  };
```

Code 7. Ramulator's generalized template and its specialization

In Figure 4.1, we visualize a fully instantiated tree, consisting of nodes at the channel, rank, and bank levels.[1] Instead of having a separate class for each level (`DDR3_Channel`, `DDR3_Rank`, `DDR3_Bank`), Ramulator simply treats a level as just another property of a node — a property that can be easily reassigned to accommodate different hierarchies with different levels. Ramulator also provides a memory controller (not shown in the figure) that interacts with the tree through only the root node (i.e., channel). Whenever the memory controller initiates a query or an operation, it results in a traversal down the tree, touching only the relevant nodes during the process. This, and more, will be explained next.

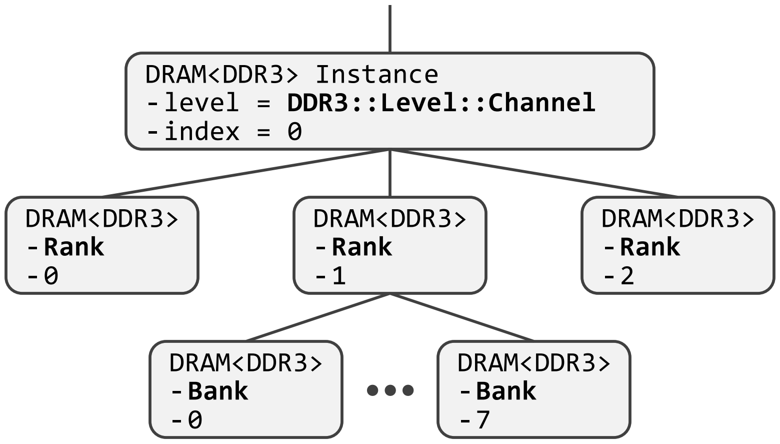


Figure 4.1. Tree of DDR3 state-machines

4.1.2. Behavior of State-Machines

States. Generally speaking, a state-machine maintains a set of states, whose transitions are triggered by an external input. In Ramulator, each state-machine (i.e., node) maintains two types of states as shown in Code 2 (top, lines 5–6): *status* and *horizon*. First, *status* is the node's state proper, which can assume one of the statuses defined by the `DDR3` class in Code 2 (bottom). The node may transition into another status when it receives one of the *commands* defined by the `DDR3` class. Second, *horizon* is a lookup-table for the earliest time when each command can be received by the node. Its purpose is to prevent a node from making premature transitions between statuses, thereby honoring

[1]Due to their sheer number (tens of thousands), nodes at or below the row level are *not* instantiated. Instead, their bookkeeping is relegated to their parent — in DDR3's particular case, the bank.

DDR3 *timing parameters* (to be explained later). We purposely neglected to mention a third state called *leaf_status*, because it is merely an optimization artifact — leaf_status is a sparsely populated hash-table used by a bank to track the status of its rows (i.e., leaf nodes) instead of instantiating them.

Functions. Code 2 (top, lines 9–11) also shows three functions that are exposed at each node: *decode*, *check*, and *update*. These functions are recursively defined, meaning that an invocation at the root node (by the memory controller) causes these functions to walk down the tree. In the following, we explain how the memory controller relies on these three functions to serve a memory request — in this particular example, a read request.

1. `decode()`: The ultimate goal of a read request is to read from DRAM, which is accomplished by a read command. Depending on the status of the tree, however, it may not be possible to issue the read command: e.g., the rank is powered-down or the bank is closed. For a given command to a given address,[2] the decode function returns a "prerequisite" command that must be issued before it, if any exists: e.g., power-up or activate command.

2. `check()`: Even if there are no prerequisites, it doesn't mean that the read command can be issued right away: e.g., the bank may not be ready if it was activated just recently. For a given command to a given address, the check function returns whether or not the command can be issued right *now* (i.e., current cycle).

3. `update()`: If the check is passed, there is nothing preventing the memory controller from issuing the read command. For a given command to a given address, the update function triggers the necessary modifications to the status/horizon (of the affected nodes) to signify the command's issuance at the current cycle. In Ramulator, invoking the update function *is* issuing a command.

[2]An *address* is an array of node indices specifying a path down the tree.

```
// DRAM.h
template <typename T>
class DRAM {
    // states (queried/updated by functions below)
    T::Status status;
    long horizon[T::Command::MAX];
    map<int, T::Status> leaf_status;  // for bank only

    // functions (recursively traverses down tree)
    T::Command decode(T::Command cmd, int addr[]);
    bool check(T::Command cmd, int addr[], long now);
    void update(T::Command cmd, int addr[], long now);
};
```

```
// DDR3.h/cpp
class DDR3 {
    enum class Status {Open, Closed, ..., MAX};
    enum class Command {ACT, PRE, RD, WR, ..., MAX};
};
```

Code 8. Specifying the DDR3 state-machines: states and functions

4.1.3. A Closer Look at a State-Machine

So far, we have described the role of the three functions without describing how they exactly perform their role. To preserve the standard-agnostic nature of the `DRAM` class, the three functions defer most of their work to the `DDR3` class, which supplies them with all of the standard-dependent information in the form of three lookup-tables: *(i) prerequisite*, *(ii) timing*, and *(iii) transition*. Within these tables are encoded the DDR3 standard, providing answers to the following three questions: *(i)* which commands must be preceded by which other commands at which levels/statuses? *(ii)* which timing parameters at which levels apply between which commands? *(iii)* which commands to which levels trigger which status transitions?

Decode. Due to space limitations, we cannot go into detail about all three lookup-tables. However, Code 3 (bottom) does provide a glimpse of only the first lookup-table, called *prerequisite*, which is consulted inside the decode function as shown in Code 3 (top).

In brief, prerequisite is a two-dimensional array of *lambdas* (a C++11 construct), which is indexed using the *(i)* level in the hierarchy at which the *(ii)* command is being decoded. As a concrete example, Code 3 (bottom, lines 7–13) shows how one of its entries is defined, which happens to be for *(i)* the rank-level and *(ii)* the refresh command. The entry is a lambda, whose sole argument is a pointer to the rank-level node that is trying to decode the refresh command. If any of the node's children (i.e., banks) are open, the lambda returns the precharge-all command (i.e., `PREA`, line 11), which would close all the banks and pave the way for a subsequent refresh command. Otherwise, the lambda returns the refresh command itself (i.e., `REF`, line 12), signaling that no other command need be issued before it. Either way, the command has been successfully decoded at that particular level, and there is no need to recurse further down the tree. However, that may not always be the case. For example, the only reason why the rank-level node was asked to decode the refresh command was because its parent (i.e., channel) did not have enough information to do so, forcing it to invoke the decode function at its child (i.e., rank). When a command cannot be decoded at a level, the lambda returns a sentinel value (i.e., `MAX`), indicating that the recursion should continue on down the tree, until the command is eventually decoded by a different lambda at a lower level (or until the recursion stops at the lowest-level).

```
// DRAM.h
template <typename T>
class DRAM {
    T::Command decode(T::Command cmd, int addr[]) {
        if (prereq[level][cmd]) {
            // consult lookup-table to decode command
            T::Command p = prereq[level][cmd](this);
            if (p != T::Command::MAX)
                return p;  // decoded successfully
        }

        if (children.size() == 0)  // lowest-level
            return cmd; // decoded successfully

        // use addr[] to identify target child...
        // invoke decode() at the target child...
    }
};
```

```
// DDR3.h/cpp
class DDR3 {
    // declare 2D lookup-table of lambdas
    function<Command(DRAM<DDR3>*)>
        prereq[Level::MAX][Command::MAX];

    // populate an entry in the table
    prereq[Level::Rank][Command::REF] =
        [] (DRAM<DDR3>* node) -> Command {
            for (auto bank : node->children)
                if (bank->status == Status::Open)
                    return Command::PREA;
            return Command::REF;
        };

    // populate other entries...
};
```

Code 9. The lookup-table for `decode()`: `prereq`

Check & Update. In addition to prerequisite, the `DDR3` class also provides two other lookup-tables: *transition* and *timing*. As is apparent from their names, they encode the status transitions and the timing parameters, respectively. Similar to *prerequisite*, these two are also indexed using some combination of levels, commands, and/or statuses. When a command is issued, the update function consults *both* lookup-tables to modify *both* the status (via lookups into transition) and the horizon (via lookups into timing) for all of the affected nodes in the tree. In contrast, the check function does *not* consult any of the lookup-tables in the `DDR3` class. Instead, it consults only the horizon, the localized lookup-table that is embedded inside the `DRAM` class itself. More specifically, the check function simply verifies whether the following condition holds true for every node affected by a command: `horizon[cmd]` $\leq$ `now`. This ensures that the time, as of right now, is already past the earliest time at which the command can be issued. The check function relies on the update function for keeping the horizon lookup-table up-to-date. As a result, the check function is able to remain computationally inexpensive — it simply looks up a horizon value and compares it against the current time. For performance reasons, we deliberately optimized the check function to be lightweight, because it could be invoked many times each cycle — the memory controller typically has more than one memory request whose scheduling eligibility must be determined. In contrast, the update function is invoked at most once-per-cycle and can afford to be more expensive. The implementation details of the update function, as well as that of other components, can be found in the source code.

4.2. Extensibility of Ramulator

Ramulator's extensibility is a natural result of its fully-decoupled design: Ramulator provides a generalized skeleton of DRAM (i.e., `DRAM.h`) that is capable of being infused with the specifics of an arbitrary DRAM standard (e.g., `DDR3.h/cpp`). To demonstrate the extensibility of Ramulator, we describe how easy it was to add support for DDR4: *(i)* copy `DDR3.h/cpp` to `DDR4.h/cpp`, *(ii)* add `BankGroup` as an item in `DDR4::Level`, and *(iii)* add or edit 20 entries in the lookup-tables — 1 in prerequisite, 2 in transition, and 17 in timing. Although there were some other changes that were also required (e.g., speed-bins), only tens of lines of code were modified in total — giving a general idea about the ease at which

Ramulator is extended. As far as Ramulator is concerned, the difference between any two DRAM standards is simply a matter of the difference in their lookup-tables, whose entries are populated in a disciplined and localized manner. This is in contrast to existing simulators, which require the programmer to chase down each of the hardcoded for-loops and if-conditions that are likely scattered across the codebase.

In addition, Ramulator also provides a single, unified memory controller that is compatible with all of the standards that are supported by Ramulator (Table 4.2). Internally, the memory controller maintains three queues of memory requests: *read*, *write*, and *maintenance*. Whereas the read/write queues are populated by demand memory requests (`read`, `write`) generated by an external source of memory traffic, the maintenance queue is populated by other types of memory requests (`refresh`, `powerdown`, `selfrefresh`) generated internally by the memory controller as they are needed. To serve a memory request in any of the queues, the memory controller interacts with the tree of DRAM state-machines using the three functions described in Section 4.1.2 (i.e., decode, check, and update). The memory controller also supports several different scheduling policies that determine the priority between requests from different queues, as well as those from the same queue.

4.3. Validation & Evaluation

As a simulator for the memory controller and the DRAM system, Ramulator must be supplied with a stream of memory requests from an external source of memory traffic. For this purpose, Ramulator exposes a simple software interface that consists of two functions: one for receiving a request into the controller, and the other for returning a request after it has been served. To be precise, the second function is a callback that is bundled inside the request. Using this interface, Ramulator provides two different modes of operation: *(i)* standalone mode where it is fed a memory trace or an instruction trace, and *(ii)* integrated mode where it is fed memory requests from an execution-driven engine (e.g., gem5 [16]). In this section, we present the results from operating Ramulator in standalone-mode, where we validate its correctness (Section 4.3.1), compare its performance with other DRAM simulators (Section 4.3.2), and conduct a cross-sectional study of contemporary DRAM standards (Section 4.3.3). Directions for conducting the experiments are included the source code release [2].

4.3.1. Validating the Correctness of Ramulator

Ramulator must simulate any given stream of memory requests using a legal sequence of DRAM commands, honoring the status transitions and the timing parameters of a standard (e.g, DDR3). To validate this behavior, we created a synthetic memory trace that would stress-test Ramulator under a wide variety of command interleavings. More specifically, the trace contains 10M memory requests, the majority of which are reads and writes (9:1 ratio) to a mixture of random and sequential addresses (10:1 ratio), and the minority of which are refreshes, power-downs, and self-refreshes.[3] While this trace was fed into Ramulator as fast as possible (without overflowing the controller's request buffer), we collected a timestamped log of every command that was issued by Ramulator. We then used this trace as part of an RTL simulation by feeding it into Micron's DDR3 Verilog model [108] — a reference implementation of DDR3. Throughout the entire duration of the RTL simulation ($\sim$$10$ hours), no violations were ever reported, indicating that Ramulator's DDR3 command sequence is indeed legal.[4] Due to the lack of corresponding Verilog models, however, we could not employ the same methodology to validate other standards. Nevertheless, we are reasonably confident in their correctness, because we implemented them by making careful modifications to Ramulator's DDR3 model, modifications that were expressed succinctly in just a few lines of code — minimizing the risk of human error, as well as making it easy to double-check. In fact, the ease of validation is another advantage of Ramulator, arising from its clean and modular design.

[3]We exclude maintenance-related requests which are not supported by Ramulator or other simulators: e.g., ZQ calibration and mode-register set.

[4]This verifies that Ramulator does not issue commands too early. However, the Verilog model does not allow us to verify whether Ramulator issues commands too late.

4.3.2. Measuring the Performance of Ramulator

In Table 4.3, we quantitatively compare Ramulator with four other standalone simulators using the same experimental setup. All five were configured to simulate DDR3-1600[5] for two different memory traces, *Random* and *Stream*, comprising 100M memory requests (read:write=9:1) to random and sequential addresses, respectively. For each simulator, Table 4.3 presents four metrics: *(i)* simulated clock cycles, *(ii)* simulation runtime, *(iii)* simulated request throughput, and *(iv)* maximum memory consumption. From the table, we make three observations. First, all five simulators yield roughly the same number of simulated clock cycles, where the slight discrepancies are caused by the differences in how their memory controllers make scheduling decisions (e.g., when to issue reads vs. writes). Second, Ramulator has the shortest simulation runtime (i.e., the highest simulated request throughput), taking only 752/249 seconds to simulate the two traces — a 2.5×/3.0× speedup compared to the next fastest simulator. Third, Ramulator consumes only a small amount of memory while it executes (2.1MB). We conclude that Ramulator provides superior performance and efficiency, as well as the greatest extensibility.

Simulator (clang -O3)	*Cycles (10^6)*		*Runtime (sec.)*		*Req/sec (10^3)*		*Memory (MB)*
	Random	*Stream*	*Random*	*Stream*	*Random*	*Stream*	
Ramulator	652	411	752	249	133	402	2.1
DRAMSim2	645	413	2,030	876	49	114	1.2
USIMM	661	409	1,880	750	53	133	4.5
DrSim	647	406	18,109	12,984	6	8	1.6
NVMain	666	413	6,881	5,023	15	20	4,230.0

Table 4.3. Comparison of five simulators using two traces

4.3.3. Cross-Sectional Study of DRAM Standards

With its integrated support for many different DRAM standards — some of which (e.g., LPDDR4, WIO2) have never been modeled before in academia — Ramulator unlocks the

[5]Single rank, 800Mhz, 11-11-11, row-interleaved, FR-FCFS [130], open-row policy.

ability to perform a comparative study across them. In particular, we examine nine different standards (Table 4.4), whose configurations (e.g., timing) were set to reasonable values. Instead of memory traces, we collected instruction traces from 22 SPEC2006 benchmarks,[6] which were fed into a simplistic "CPU" model that comes with Ramulator.[7]

Standard	*Rate (MT/s)*	*Timing (CL-RCD-RP)*	*Data-Bus (Width×Chan.)*	*Rank-per-Chan*	*BW (GB/s)*
DDR3	1,600	11-11-11	64-bit × 1	1	11.9
DDR4	2,400	16-16-16	64-bit × 1	1	17.9
SALP†	1,600	11-11-11	64-bit × 1	1	11.9
LPDDR3	1,600	12-15-15	64-bit × 1	1	11.9
LPDDR4	2,400	22-22-22	32-bit × 2*	1	17.9
GDDR5 [53]	6,000	18-18-18	64-bit × 1	1	44.7
HBM	1,000	7-7-7	128-bit × 8*	1	119.2
WIO	266	7-7-7	128-bit × 4*	1	15.9
WIO2	1,066	9-10-10	128-bit × 8*	1	127.2

†*MASA [87] on top of DDR3 with 8 subarrays-per-bank.*
**More than one channel is built into these particular standards.*

Table 4.4. Configuration of nine DRAM standards used in study

Figure 4.2 contains the violin plots and geometric means of the normalized IPC compared to the DDR3 baseline. We make several broad observations. First, newly upgraded standards (e.g., DDR4) perform better than their older counterparts (e.g., DDR3). Second, standards for embedded systems (i.e., LPDDRx, WIOx) have lower performance because they are optimized to consume less power. Third, standards for graphics systems (i.e., GDDR5, HBM) provide a large amount of bandwidth, leading to higher average performance than DDR3 even for our non-graphics benchmarks. Fourth, a recent academic proposal, SALP, provides significant performance improvement (e.g., higher than that of WIO2) by reducing the serialization effects of bank conflicts without increasing peak bandwidth. These observations are only a small sampling of the analyses that are enabled by Ramulator.

[6]perlbench, bwaves, gamess, povray, calculix, tonto were unavailable for trace collection.

[7]3.2GHz, 4-wide issue, 128-entry ROB, no instruction-dependency, one cycle for non-DRAM instructions, instruction trace is pre-filtered through a 512KB cache, memory controller has 32/32 entries in its read/write request buffers.

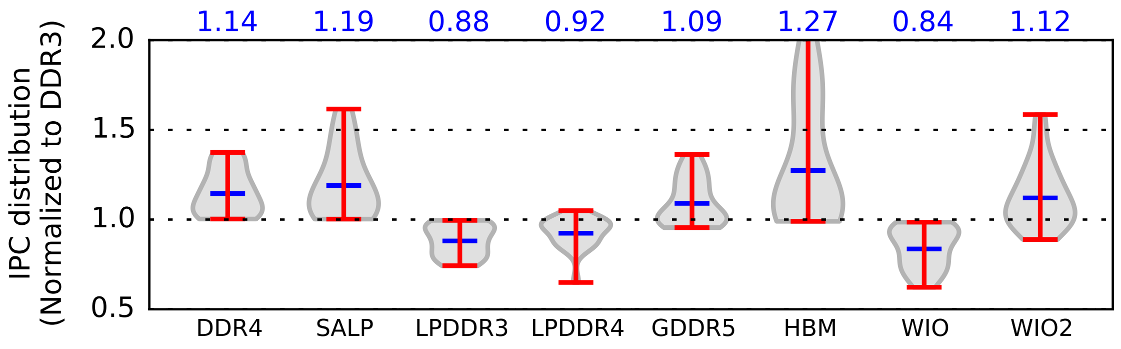


Figure 4.2. Performance comparison of DRAM standards

4.4. Chapter Summary

In this chapter, we introduced *Ramulator*, a fast and cycle-accurate simulation tool for current and future DRAM systems. We demonstrated Ramulator's advantage in efficiency and extensibility, as well as its comprehensive support for DRAM standards. We hope that Ramulator would facilitate DRAM research in an era when main memory is undergoing rapid changes [77, 116].

Chapter 5

Conclusion & Future Work

For the last four decades, it has been the sustained success of DRAM scaling that has allowed computing systems to enjoy larger and faster main memory at lower cost. Recently, however, the advantages provided by DRAM scaling have started to become offset by its disadvantages, mainly in the form of deteriorating reliability and performance. This is because, at reduced sizes, DRAM cells are significantly more vulnerable to coupling effects and process variation. Unlike in the past, these problems are too costly to be solved by employing techniques in the domain of circuits/devices alone. In this thesis, we showed the effectiveness of taking an architectural approach to enhance DRAM scaling. First, we demonstrated the widespread existence of a new reliability problem — disturbance errors — in recent DRAM chips, and proposed to prevent them through a collaborative effort between the DRAM controller and the DRAM chips. Second, we highlighted a latency serialization bottleneck in DRAM chips, and proposed to alleviate it by making small and non-intrusive modifications to the DRAM architecture that increase the parallelism of its underlying subarrays. Lastly, we developed a DRAM simulator, called Ramulator, that accelerates the design space exploration of DRAM architecture with its high simulation speed and ease of extensibility. Our architectural approach, combined with the benefits of traditional circuits/devices scaling, provides a more sustainable roadmap for DRAM-based main memory.

5.1. Future Work

As DRAM process technology fast approaches its limit, this thesis contends that computer architects must play a greater role in defining and building the next generation of memory systems. Treating the memory system simply as "a bag of commodity DRAM chips" — as it has been done in the past — is no longer a viable approach. In fact, several disruptive changes to the memory system have already been set in motion. For example, the shortcomings of commodity DRAM in providing adequate bandwidth and energy-efficiency are driving the industry toward *3D die-stacking* (e.g., HMC, WIO2, HBM, MC-DRAM), especially for graphics and embedded systems. Also, the projected erosion in DRAM's cost-per-bit is sparking renewed interest in cheaper *non-volatile alternatives* (e.g., resistive memory, phase change memory), despite their inferior latency and endurance characteristics. The nature of these and other emerging technologies is such that they create new opportunities to provision the memory system with a rich set of features and capabilities, while also presenting new challenges that must be overcome.

Hardware Reliability & Security. At advanced technology nodes, hardware failures will become more commonplace, some of which may be diagnosed only after they have been released into the wild, as was the case with disturbance errors in DRAM [84]. This opens up the possibility of zero-day hardware vulnerabilities that could undermine system integrity in unpredictable ways. Moreover, such vulnerabilities would have far-reaching consequences since they exploit a systemic weakness in the process technology itself, thereby affecting millions of logic and memory chips that have already been deployed in the field. From this, we identify two research topics. First, we plan to devise new testing methodologies that can expose emerging failure modes in the hardware without *a priori* knowledge about their symptoms. Second, I plan to develop intelligent hardware controllers that can be reconfigured on-the-fly (i.e., a hardware "patch") in response to newly discovered threats.

Reformulating the Memory Hierarchy. Traditionally, the memory hierarchy consisted of well-defined tiers (e.g., cache, main memory, storage) that were clearly distinguished from each other with regard to capacity, performance, function, and physical medium. But now, the boundaries between them are blurring. Embedded DRAM (eDRAM) and die-stacked DRAM are bridging the gap between cache and main memory, while non-volatile memory is poised to do the same between main memory and storage. What was once a clear-cut, black-and-white hierarchy is rapidly becoming an ambiguous one with many shades of gray. This calls for a broad re-evaluation of the trade-offs that have been assumed at each tier.

We identify three research topics in particular. First, the hierarchy must expose new software primitives that allow the application to specify the tier from which it allocates memory. Such primitives should be descriptive, so that the application can fully express the attributes of the memory it desires (e.g., high vs. low bandwidth, volatile vs. non-volatile). Alternatively, the memory system could make this decision on behalf of the application, based on information gathered from its past behavior. Second, the hierarchy must assign appropriate roles for each tier, which may change fluidly depending on the system's budget and composition. For example, the low capacity of die-stacked DRAM may suffice as main memory in low-end systems, but not in high-end systems where it is more befitting as the last-level cache. Third, the hierarchy must gracefully manage all the implications of introducing non-volatility into main memory. This includes, but is not limited to, reconciling two disparate namespaces (addresses vs. files), bootup/recovery from inconsistent memory states, and vulnerability to remanence attacks on sensitive data.

Bibliography

[1] Memtest86+ v4.20. `http://www.memtest.org`.

[2] Ramulator source code. `https://github.com/CMU-SAFARI/ramulator`.

[3] The GNU GRUB Manual, 2012.

[4] J. H. Ahn, N. P. Jouppi, C. Kozyrakis, J. Leverich, and R. S. Schreiber. Improving System Energy Efficiency with Memory Rank Subsetting. *ACM TACO*, Mar. 2012.

[5] J. H. Ahn, J. Leverich, R. Schreiber, and N. P. Jouppi. Multicore DIMM: An Energy Efficient Memory Module with Independently Controlled DRAMs. *IEEE CAL*, 2009.

[6] J. H. Ahn, S. Li, O. Seongil, and N. Jouppi. McSimA+: A Manycore Simulator with Application-Level+ Simulation and Detailed Microarchitecture Modeling. In *ISPASS*, 2013.

[7] Z. Al-Ars. *DRAM Fault Analaysis and Test Generation*. PhD thesis, TU Delft, 2005.

[8] Z. Al-Ars, S. Hamdioui, A. van de Goor, G. Gaydadjiev, and J. Vollrath. DRAM-Specific Space of Memory Tests. In *ITC*, 2006.

[9] AMD. BKDG for AMD Family 15h Models 10h-1Fh Processors, 2013.

[10] K. Bains and J. Halbert. Distributed Row Hammer Tracking. US Patent App. 13/631,781, Apr. 3 2014.

[11] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield. Row Hammer Refresh Command. US Patent App. 13/539,415, Jan. 2 2014.

[12] K. Bains, J. Halbert, C. Mozak, T. Schoenborn, and Z. Greenfield. Row Hammer Refresh Command. US Patent App. 14/068,677, Feb. 27 2014.

[13] K. Bains, J. Halbert, S. Sah, and Z. Greenfield. Method, Apparatus and System for Providing a Memory Refresh. US Patent App. 13/625,741, Mar. 27 2014.

[14] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *ISPASS*, 2009.

[15] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to Flash Memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[16] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, May 2011.

[17] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[18] D. Burger and T. M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News*, June 1997.

[19] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis. In *DATE*, pages 521–526, 2012.

[20] Y. Cai, O. Mutlu, E. F. Haratsch, and K. Mai. Program Interference in MLC NAND Flash Memory: Characterization, Modeling and Mitigation. In *ICCD*, 2013.

[21] S. Y. Cha. DRAM and Future Commodity Memories. In *VLSI Technology Short Course*, 2011.

[22] K. Chandrasekar, C. Weis, Y. Li, B. Akesson, N. Wehn, and K. Goossens. DRAMPower: Open-Source DRAM Power & Energy Estimation Tool. `http://www.drampower.info`, 2012.

[23] N. Chandrasekaran, S. Hues, S. Lu, D. Li, and C. Biship. Characterization and Metrology Challenges for Emerging Memory Technology Landscape. In *Frontiers of Characterization and Metrology for Nanoelectronics*, 2013.

[24] K. Chang, D. Lee, Z. Chishti, C. Wilkerson, A. Alameldeen, Y. Kim, and O. Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *HPCA*, 2014.

[25] M.-T. Chao, H.-Y. Yang, R.-F. Huang, S.-C. Lin, and C.-Y. Chin. Fault Models for Embedded-DRAM Macros. In *DAC*, 2009.

[26] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the Utah SImulated Memory Module. *UUCS-12-002, University of Utah*, Feb. 2012.

[27] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads. In *HPCA*, 2012.

[28] Q. Chen, H. Mahmoodi, S. Bhunia, and K. Roy. Modeling and Testing of SRAM for New Failure Mechanisms Due to Process Variations in Nanoscale CMOS. In *VLSI Test Symposium*, 2005.

[29] P.-F. Chia, S.-J. Wen, and S. Baeg. New DRAM HCI Qualification Method Emphasizing on Repeated Memory Access. In *Integrated Reliability Workshop*, 2010.

[30] S. Cohen and Y. Matias. Spectral Bloom Filters. In *SIGMOD*, 2003.

[31] J. Cooke. The Inconvenient Truths of NAND Flash Memory. In *Flash Memory Summit*, 2007.

[32] DRAMeXchange. TrendForce: 3Q13 Global DRAM Revenue Rises by 9%, Samsung Shows Most Noticeable Growth, Nov. 12, 2013.

[33] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel Application Memory Scheduling. In *MICRO*, 2011.

[34] Enhanced Memory Systems. Enhanced SDRAM SM2604, 2002.

[35] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *Transactions on Networking*, 8(3), June 2000.

[36] J. A. Fifield and H. L. Kalter. Crosstalk-Shielded-Bit-Line DRAM. US Patent 5,010,524, Apr. 23, 1991.

[37] H. Fredriksson and C. Svensson. Improvement potential and equalization example for multidrop DRAM memory buses. *IEEE Transactions on Advanced Packaging*, 2009.

[38] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling. In *HPCA*, 2007.

[39] Z. Greenfield, K. Bains, T. Schoenborn, C. Mozak, and J. Halbert. Row Hammer Condition Monitoring. US Patent App. 13/539,417, Jan. 2, 2014.

[40] Z. Greenfield, J. Halbert, and K. Bains. Method, Apparatus and System for Determining a Count of Accesses to a Row of Memory. US Patent App. 13/626,479, Mar. 27 2014.

[41] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing Flash Memory: Anomalies, Observations, and Applications. In *MICRO*, 2009.

[42] Z. Guo, A. Carlson, L.-T. Pang, K. T. Duong, T.-J. K. Liu, and B. Nikolic. Large-Scale SRAM Variability Characterization in 45 nm CMOS. *Journal of Solid-State Circuits*, 44(11):3174–3192, 2009.

[43] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A. Udipi. Simulating DRAM Controllers for Future System Architecture Exploration. In *ISPASS*, 2014.

[44] C. A. Hart. CDRAM in a unified memory architecture. In *Compcon*, 1994.

[45] D. Henderson and J. Mitchell. *IBM POWER7 System RAS*, Dec. 2012.

[46] H. Hidaka, Y. Matsuda, M. Asakura, and K. Fujishima. The Cache DRAM Architecture: A DRAM with an On-Chip Cache Memory. *IEEE Micro*, Mar. 1990.

[47] M. Horiguchi and K. Itoh. *Nanoscale Memory Repair*. Springer, 2011.

[48] HPCC. RandomAccess. `http://icl.cs.utk.edu/hpcc/`.

[49] W.-C. Hsu and J. E. Smith. Performance of cached DRAM organizations in vector supercomputers. In *ISCA*, 1993.

[50] R.-F. Huang, H.-Y. Yang, M. Chao, and S.-C. Lin. Alternate Hammering Test for Application-Specific DRAMs and an Industrial Case Study. In *DAC*, 2012.

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

[52] Hybrid Memory Cube Consortium. *HMC Specification 1.1*, Feb. 2014.

[53] Hynix. *GDDR5 SGRAM H5GQ1H24AFR*, Nov. 2009.

[54] Intel. 2nd Gen. Intel Core Processor Family Desktop Datasheet, 2011.

[55] Intel. Intel Core Desktop Processor Series Datasheet, 2011.

[56] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, 2012.

[57] Intel. 4th Generation Intel Core Processor Family Desktop Datasheet, 2013.

[58] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self Optimizing Memory Controllers: A Reinforcement Learning Approach. In *ISCA*, 2008.

[59] K. Itoh. Semiconductor Memory. US Patent 4,044,340, Apr. 23, 1977.

[60] K. Itoh. *VLSI Memory Chip Design*. Springer, 2001.

[61] James Reinders. Knights Corner: Your Path to Knights Landing, Sept. 17, 2014.

[62] JEDEC. *JESD79-3 DDR3 SDRAM Standard*, June 2007.

[63] JEDEC. *JESD212 GDDR5 SGRAM*, Dec. 2009.

[64] JEDEC. Standard No. 79-3E. DDR3 SDRAM Specification, 2010.

[65] JEDEC. *JESD229 Wide I/O Single Data Rate (Wide/IO SDR)*, Dec. 2011.

[66] JEDEC. Standard No. 21-C. Annex K: Serial Presence Detect (SPD) for DDR3 SDRAM Modules, 2011.

[67] JEDEC. *JESD209-3 Low Power Double Data Rate 3 (LPDDR3)*, May 2012.

[68] JEDEC. *JESD79-3F DDR3 SDRAM Standard*, July 2012.

[69] JEDEC. *JESD79-4 DDR4 SDRAM*, Sept. 2012.

[70] JEDEC. *JESD235 High Bandwidth Memory (HBM) DRAM*, Oct. 2013.

[71] JEDEC. *JESD-21C (4.1.2.11) Serial Presence Detect (SPD) for DDR3 SDRAM Modules*, Feb. 2014.

[72] JEDEC. *JESD209-4 Low Power Double Data Rate 3 (LPDDR4)*, Aug. 2014.

[73] JEDEC. *JESD229-2 Wide I/O 2 (WideIO2)*, Aug. 2014.

[74] M. K. Jeong, D. H. Yoon, and M. Erez. DrSim: A Platform for Flexible DRAM System Research. `http://lph.ece.utexas.edu/public/DrSim`, 2012.

[75] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee, and M. Erez. Balancing DRAM Locality and Parallelism in Shared Memory CMP Systems. In *HPCA*, 2012.

[76] W. Jiang, G. Khera, R. Wood, M. Williams, N. Smith, and Y. Ikeda. Cross-Track Noise Profile Measurement for Adjacent-Track Interference Study and Write-Current Optimization in Perpendicular Recording. *Journal of Applied Physics*, 93(10):6754–6756, 2003.

[77] U. Kang, H. soo Yu, C. Park, H. Zheng, J. Halbert, K. Bains, S. Jang, and J. S. Choi. Co-Architecting Controllers and DRAM to Enhance DRAM Process Scaling. In *The Memory Forum (Co-located with ISCA)*, 2014.

[78] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *Transactions on Database Systems*, 28(1), Mar. 2003.

[79] G. Kedem and R. P. Koganti. WCDRAM: A Fully Associative Integrated Cached-DRAM with Wide Cache Lines. *CS-1997-03, Duke*, 1997.

[80] B. Keeth, R. J. Baker, B. Johnson, and F. Lin. *DRAM Circuit Design. Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2007.

[81] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu. The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study. In *SIGMETRICS*, 2014.

[82] R. Kho, D. Boursin, M. Brox, P. Gregorius, H. Hoenigschmid, B. Kho, S. Kieser, D. Kehrer, M. Kuzmenka, U. Moeller, P. Petkov, M. Plan, M. Richter, I. Russell, K. Schiller, R. Schneider, K. Swaminathan, B. Weber, J. Weber, I. Bormann, F. Funfrock, M. Gjukic, W. Spirkl, H. Steffens, J. Weller, and T. Hein. 75nm 7Gb/s/pin 1Gb GDDR5 Graphics Memory Device with Bandwidth-Improvement Techniques. In *ISSCC*, 2009.

[83] D. Kim, V. Chandra, R. Aitken, D. Blaauw, and D. Sylvester. Variation-Aware Static and Dynamic Writability Analysis for Voltage-Scaled Bit-Interleaved 8-T SRAMs. In *ISLPED*, 2011.

[84] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[85] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. In *HPCA*, 2010.

[86] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior. In *MICRO*, 2010.

[87] Y. Kim, V. Seshadri, D. Lee, J. Liu, and O. Mutlu. A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM. In *ISCA*, 2012.

[88] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE CAL*, 2015.

[89] T. Kirihata. Latched Row Decoder for a Random Access Memory. U.S. patent number 5615164, 1997.

[90] B.-S. Kong, S.-S. Kim, and Y.-H. Jun. Conditional-Capture Flip-Flop for Statistical Power Reduction. *IEEE JSSC*, 2001.

[91] Y. Konishi, M. Kumanoya, H. Yamasaki, K. Dosaka, and T. Yoshihara. Analysis of Coupling Noise between Adjacent Bit Lines in Megabit DRAMs. *IEEE Journal of Solid-State Circuits*, 24(1):35–42, 1989.

[92] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *ISCA*, 1981.

[93] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA*, 2009.

[94] C. J. Lee, V. Narasiman, E. Ebrahimi, O. Mutlu, and Y. N. Patt. DRAM-Aware Last-Level Cache Writeback: Reducing Write-Caused Interference in Memory Systems. *TR-HPS-2010-002, UT Austin*, 2010.

[95] D. Lee, Y. Kim, G. Pekhimenko, S. Khan, V. Seshadri, K. Chang, and O. Mutlu. Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case. In *HPCA*, 2015.

[96] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu. Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture. In *HPCA*, 2013.

[97] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu. An Experimental Study of Data Retention Behavior in Modern DRAM Devices: Implications for Retention Time Profiling Mechanisms. In *ISCA*, 2013.

[98] J. Liu, B. Jaiyen, R. Veras, and O. Mutlu. RAIDR: Retention-Aware Intelligent DRAM Refresh. In *ISCA*, 2012.

[99] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *PACT*, 2012.

[100] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *PLDI*, 2005.

[101] J. A. Mandelman, R. H. Dennard, G. B. Bronner, J. K. DeBrosse, R. Divakaruni, Y. Li, and C. J. Radens. Challenges and Future Directions for the Scaling of Dynamic Random-Access Memory (DRAM). *IBM Journal of Research and Development*, 46(2.3):187–212, Mar. 2002.

[102] M. Meterelliyoz, F. Al-amoody, U. Arslan, F. Hamzaoglu, L. Hood, M. Lal, J. Miller, A. Ramasundar, D. Soltman, W. Ifar, Y. Wang, and K. Zhang. 2nd Generation Embedded DRAM with 4X Lower Self Refresh Power in 22nm Tri-Gate CMOS Technology. In *VLSI Symposium*, 2014.

[103] M. Micheletti. Tuning DDR4 for Power and Performance. In *MemCon*, 2013.

[104] Micron. DDR3 SDRAM System-Power Calculator, 2010.

[105] Micron. Micron Announces Sample Availability for Its Third-Generation RLDRAM(R) Memory. `http://investors.micron.com/releasedetail.cfm?ReleaseID=581168`, May 26, 2011.

[106] Micron. 2Gb: x16, x32 Mobile LPDDR2 SDRAM, 2012.

[107] Micron. 2Gb: x4, x8, x16, DDR3 SDRAM, 2012.

[108] Micron. DDR3 SDRAM Verilog model, 2012.

[109] M. J. Miller. Bandwidth Engine Serial Memory Chip Breaks 2 Billion Accesses/sec. In *HotChips*, 2011.

[110] D.-S. Min, D.-I. Seo, J. You, S. Cho, D. Chin, and Y. E. Park. Wordline Coupling Noise Reduction Techniques for Scaled DRAMs. In *Symposium on VLSI Circuits*, 1990.

[111] Y. Moon, Y.-H. Cho, H.-B. Lee, B.-H. Jeong, S.-H. Hyun, B.-C. Kim, I.-C. Jeong, S.-Y. Seo, J.-H. Shin, S.-W. Choi, H.-S. Song, J.-H. Choi, K.-H. Kyung, Y.-H. Jun, and K. Kim. 1.2V 1.6Gb/s 56nm 6F2 4Gb DDR3 SDRAM with Hybrid-I/O Sense Amplifier and Segmented Sub-Array Architecture. In *ISSCC*, 2009.

[112] R. Morris. Counting Large Numbers of Events in Small Registers. *Communications of the ACM*, 21(10):840–842, Oct. 1978.

[113] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *USENIX SS*, 2007.

[114] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda. Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning. In *MICRO*, 2011.

[115] C. H. Museum. Oral History of Joel Karp (Interviewed by Gardner Hendrie), Mar. 2003.

[116] O. Mutlu. Memory Scaling: A Systems Architecture Perspective. In *MemCon*, 2013.

[117] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *MICRO*, 2007.

[118] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ISCA*, 2008.

[119] P. J. Nair, D.-H. Kim, and M. K. Qureshi. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *ISCA*, 2013.

[120] S. Narasimha, P. Chang, C. Ortolland, D. Fried, E. Engbrecht, K. Nummy, P. Parries, T. Ando, M. Aquilino, N. Arnold, R. Bolam, J. Cai, M. Chudzik, B. Cipriany, G. Costrini, M. Dai, J. Dechene, C. Dewan, B. Engel, M. Gribelyuk, D. Guo, G. Han, N. Habib, J. Holt, D. Ioannou, B. Jagannathan, D. Jaeger, J. Johnson, W. Kong, J. Koshy, R. Krishnan, A. Kumar, M. Kumar, J. Lee, X. Li, C. Lin, B. Linder, S. Lucarini, N. Lustig, P. McLaughlin, K. Onishi, V. Ontalus, R. Robison, C. Sheraw, M. Stoker, A. Thomas, G. Wang, R. Wise, L. Zhuang, G. Freeman, J. Gill, E. Maciejewski, R. Malik, J. Norum, and P. Agnello. 22nm High-Performance SOI Technology Featuring Dual-Embedded Stressors, Epi-Plate High-K Deep-Trench Embedded DRAM and Self-Aligned Via 15LM BEOL. In *IEDM*, 2012.

[121] NEC. Virtual Channel SDRAM uPD4565421, 1999.

[122] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO*, 2006.

[123] C. Nibby, R. Goldin, and T. Andrews. Remap Method and Apparatus for a Memory System Which Uses Partially Good Memory Devices. US Patent 4,527,251, July 2 1985.

[124] S. O, Y. H. Son, N. S. Kim, and J. H. Ahn. Row-Buffer Decoupling: A Case for Low-latency DRAM Microarchitecture. In *ISCA*, 2014.

[125] J.-h. Oh. Semiconductor Memory Having a Bank with Sub-Banks. U.S. patent number 7782703, 2010.

[126] E. Pinheiro, W. Weber, and L. Barroso. Failure Trends in a Large Disk Drive Population. In *FAST*, 2007.

[127] M. Poremba and Y. Xie. NVMain: An Architectural-Level Main Memory Simulator for Emerging Non-volatile Memories. In *ISVLSI*, 2012.

[128] Rambus. DRAM Power Model, 2010.

[129] M. Redeker, B. F. Cockburn, and D. G. Elliott. An Investigation into Crosstalk Noise in DRAM Structures. In *MTDT*, 2002.

[130] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory Access Scheduling. In *ISCA*, 2000.

[131] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE CAL*, 2011.

[132] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003.

[133] K. Saino, S. Horiba, S. Uchiyama, Y. Takaishi, M. Takenaka, T. Uchida, Y. Takada, K. Koyama, H. Miyake, and C. Hu. Impact of Gate-Induced Drain Leakage Current on the Tail Distribution of DRAM Data Retention Time. In *IEDM*, pages 837–840, 2000.

[134] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer Design: An Introduction*. Chapter 8, p. 58. Morgan Kaufmann, 2009.

[135] R. H. Sartore, K. J. Mobley, D. G. Carrigan, and O. F. Jones. Enhanced DRAM with Embedded Registers. U.S. patent number 5887272, 1999.

[136] Y. Sato, T. Suzuki, T. Aikawa, S. Fujioka, W. Fujieda, H. Kobayashi, H. Ikeda, T. Nagasawa, A. Funyu, Y. Fuji, K. Kawasaki, M. Yamazaki, and M. Taguchi. Fast Cycle RAM (FCRAM); A 20-ns Random Row Access, Pipe-Lined Operating DRAM. In *Symposium on VLSI Circuits*, 1998.

[137] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST*, 2007.

[138] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data. In *MICRO*, 2013.

[139] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 Multicore Server Processor. *IBM Journal Res. Dev.*, May. 2011.

[140] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor. In *ASPLOS*, 2000.

[141] STREAM Benchmark. `http://www.streambench.org/`.

[142] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-Level Cache Policies. In *ISCA*, 2010.

[143] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis. Micro-pages: Increasing DRAM efficiency with locality-aware data placement. In *ASPLOS*, 2010.

[144] Sun Microsystems. OpenSPARC T1 Microarch. Specification, 2006.

[145] N. Suzuki, H. Kim, D. de Niz, B. Andersson, L. Wrage, M. Klein, and R. Rajkumar. Coordinated Bank and Cache Coloring for Temporal Protection of Memory Accesses. In *ICESS*, 2013.

[146] A. Tanabe, T. Takeshima, H. Koike, Y. Aimoto, M. Takada, T. Ishijima, N. Kasai, H. Hada, K. Shibahara, T. Kunio, T. Tanigawa, T. Saeki, M. Sakao, H. Miyamoto, H. Nozue, S. Ohya, T. Murotani, K. Koyama, and T. Okuda. A 30-ns 64-Mb DRAM with Built-In Self-Test and Self-Repair Function. *IEEE Journal of Solid-State Circuits*, 27(11):1525–1533, 1992.

[147] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. In *DSN*, 2006.

[148] Y. Tang, X. Che, H. J. Lee, and J.-G. Zhu. Understanding Adjacent Track Erasure in Discrete Track Media. *Transactions on Magnetics*, 44(12):4780–4783, 2008.

[149] S. Thoziyoor, J. H. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi. A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies. In *ISCA*, 2008.

[150] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal Res. Dev.*, Jan. 1967.

[151] TPC. `http://www.tpc.org/`.

[152] A. N. Udipi, N. Muralimanohar, N. Chatterjee, R. Balasubramonian, A. Davis, and N. P. Jouppi. Rethinking DRAM Design and Organization for Energy-Constrained Multi-Cores. In *ISCA*, 2010.

[153] A. J. van de Goor and J. de Neef. Industrial Evaluation of DRAM Tests. In *DATE*, 1999.

[154] A. J. van de Goor and I. Schanstra. Address and Data Scrambling: Causes and Impact on Memory Tests. In *DELTA*, 2002.

[155] B. Van Durme and A. Lall. Probabilistic Counting with Randomized Storage. In *IJCAI*, 2009.

[156] R. Venkatesan, S. Herr, and E. Rotenberg. Retention-Aware Placement in DRAM (RAPID): Software Methods for Quasi-Non-Volatile DRAM. In *HPCA*, pages 155–165, 2006.

[157] T. Vogelsang. Understanding the Energy Consumption of Dynamic Random Access Memories. In *MICRO*, 2010.

[158] F. Ware and C. Hampel. Improving Power and Data Efficiency with Threaded Memory Modules. In *ICCD*, 2006.

[159] W. A. Wong and J.-L. Baer. DRAM caching. *CSE-97-03-04, UW*, 1997.

[160] R. Wood, M. Williams, A. Kavcic, and J. Miles. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *Transactions on Magnetics*, 45(2):917–923, 2009.

[161] Xilinx. *Virtex-6 FPGA Integrated Block for PCI Express*, Mar. 2011.

[162] Xilinx. *ML605 Hardware User Guide*, Oct. 2012.

[163] Xilinx. *Virtex-6 FPGA Memory Interface Solutions*, Mar. 2013.

[164] T. Yamauchi, L. Hammond, and K. Olukotun. The Hierarchical Multi-Bank DRAM: A High-Performance Architecture for Memory Integrated with Processors. In *Advanced Research in VLSI*, 1997.

[165] J. H. Yoon, H. C. Hunter, and G. A. Tressler. Flash & DRAM Si Scaling Challenges, Emerging Non-Volatile Memory Technology Enablement — Implications to Enterprise Storage and Server Compute Systems. In *Flash Memory Summit*, 2013.

[166] T. Yoshihara, H. Hidaka, Y. Matsuda, and K. Fujishima. A Twisted Bit Line Technique for Multi-Mb DRAMs. In *ISSCC*, 1988.

[167] G. L. Yuan, A. Bakhoda, and T. M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *MICRO*, 2009.

[168] T. Zhang, K. Chen, C. Xu, G. Sun, T. Wang, and Y. Xie. Half-DRAM: A High-Bandwidth and Low-Power DRAM Architecture from the Rethinking of Fine-Grained Activation. In *ISCA*, 2014.

[169] Z. Zhang, Z. Zhu, and X. Zhang. A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality. In *MICRO*, 2000.

[170] Z. Zhang, Z. Zhu, and X. Zhang. Cached DRAM for ILP Processor Memory Access Latency Reduction. *IEEE Micro*, Jul. 2001.

[171] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-Rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency. In *MICRO*, 2008.

[172] W. K. Zuravleff and T. Robinson. Controller for a Synchronous DRAM that Maximizes Throughput by Allowing Memory Requests and Commands to be Issued Out of Order. U.S. patent number 5630096, 1997.