WORKLOAD-ADAPTATION IN MEMORY CONTROLLERS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2015

By
Mohsen Ghasempour
School of Computer Science

Contents

List of Tables

List of Figures

Acronyms

AAI Average Activation Intervals. 136

ARMOR A Run-time Memory hot-row detectOR. 15

DCA Dynamic Counter Allocator. 126

DIMM Dual In-line Memory Module. 26

DRAM Dynamic Random Access Memory. 14

DReAM Dynamic Re-arrangement of Address Mapping. 15

EAMS Estimated Address-Mapping Scheme. 95

ECC Error Correcting Codes. 42

FCFS First Come First Served. 41

FPGAs Field-Programmable Gate Arrays. 14

FR-FCFS First Ready First Come First Served. 41

GMEAN Geometric Mean. 70

GPUs Graphics Processing Units. 14

HAPPY Hybrid Address-based Page PolicY. 14

HMC Hybrid-Memory Cubes. 22

HTW Hot Time Window. 125

IPC Instruction Per Clock cycle. 53

JEDEC Joint Electron Device Engineering Council. 20

LLC Last-Level Cache. 26

MC Mistake Counter. 63

MF Memory Footprint. 68

MPKI LLC Misses Per Kilo Instructions. 50

MSC Memory Channel Storage. 94

MT Migration Table. 96

NaCl Native Client. 119

PAMS Pre-defined Address-Mapping Scheme. 95

PARA Probabilistic Adjacent Row Activation. 123

PCM Phase-Changed Memory. 46

RI Refresh Interval. 124

ROI Region Of Interest. 93

SDRAM Synchronous Dynamic Random Access Memory. 37

SKG Synthetic Kernel Generator. 49

ST Swap Table. 96

TC Timeout Counter. 63

TR Timeout Register. 63

TRR Target Row Refresh. 46

TSRF Time-based Shift Register Filter. 126

TSV Through-Silicon Via. 46

USIMM Utah Simulated Memory Module. 51

Abstract

WORKLOAD-ADAPTATION IN MEMORY CONTROLLERS
Mohsen Ghasempour
A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2015

Advanced development in processor design, increasing the heterogeneity of computer system, by involving Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) and custom accelerators, and increasing the number of cores and threads in such systems puts extra pressure on the main memory, demanding a higher performance. Current computing trends are putting ever more pressure on main memory. In modern computer systems, this is generally Dynamic Random Access Memory (DRAM) which consists of a multi-level access hierarchy (e.g. Rank, Bank, Row etc.). This heterogeneity of structure implies different access latencies (and power consumption), resulting in performance differences according to memory access patterns. DRAM controllers manage access and satisfy the timing constraints and now employ complex scheduling and prediction algorithms to mitigate the effect on performance. This complexity can limit the scalability of a controller with the size of memory, while maintaining performance. The focus of this PhD thesis is to improve performance, reliability and scalability (with respect to memory size) of DRAM controllers. To this end, it covers three significant contributors to the performance and reliability of a memory controller: 'Address Mapping', 'Page Closure Policies' and 'Reliability Monitoring'. A detailed DRAM simulator is used as an evaluation platform throughout this work. The following contributions are presented in this thesis.

Hybrid Address-based Page PolicY (HAPPY): Memory controllers have used static page-closure policies to decide whether a row should be left open (*open-page policy*) or closed immediately (*close-page policy*) after use. The appropriate choice can reduce the average memory latency. Since access patterns are dynamic, static page policies cannot guarantee to deliver optimum execution time. *Hybrid* page policies can cover dynamic scenarios and are now implemented in state-of-the-art processors. These switch between open-page and close-page policies by monitoring the access

pattern of row hits/conflicts and predicting future behaviour. Unfortunately, as the size of DRAM memory increases, fine-grain tracking and analysis of accesses does not remain practical. HAPPY proposes a compact, memory address-based encoding technique which can maintain or improve page closure predictor performance while reducing the hardware overhead. As a case study, HAPPY is integrated, with a state-of-the-art monitor – the Intel-adaptive open-page policy predictor employed by the Intel Xeon X5650 – and a traditional Hybrid page policy. The experimental results show that using the HAPPY encoding applied to the Intel-adaptive page closure policy can reduce the hardware overhead by 5× for the evaluated 64 GB memory (up to 40× for a 512 GB memory) while maintaining the prediction accuracy.

Dynamic Re-arrangement of Address Mapping (DReAM): The initial location of data in DRAMs is determined and controlled by the 'address-mapping' and even modern memory controllers use a fixed and runtime-agnostic address-mapping. On the other hand, the memory access pattern seen at the memory interface level will be dynamically changed at run-time. This dynamic nature of memory access pattern and the fixed behaviour of address mapping process in DRAM controllers, implied by using a fixed address-mapping scheme, means that DRAM performance cannot be exploited efficiently. DReAM is a novel hardware technique that can detect a workload-specific address mapping at run-time based on the application access pattern. The experimental results show that DReAM outperforms the best evaluated baseline address mapping by 5%, on average, and up to 28% across all the workloads.

A Run-time Memory hot-row detectOR (ARMOR): DRAM needs refreshing to avoid data loss. Data can also be corrupted within a refresh interval by crosstalk caused by repeated accesses to neighbouring rows; this is the *row hammer* effect and is perceived as a potentially serious reliability and security threat. ARMOR is a novel technique which improves memory reliability by detecting which rows are potentially being "hammered" within the memory controller, which can then insert extra refresh operations. It can detect (and thus prevent) row hammer errors with minimal execution time overhead and hardware requirements. Alternatively by adding buffers inside the memory controller to cache such hammered rows, execution times are reduced with small hardware costs. The ARMOR technique is now the basis of a patent applicant and under process for commercial exploitation.

As a final step of this PhD thesis, an adaptive memory controller was developed integrating HAPPY, DReAM and ARMOR into a standard memory controller. The performance and the implementation cost of such an adaptive memory controller were compared against a state-of-the-art memory controller, as a baseline. The experimental results show that the adaptive memory controller outperforms the baseline, on average by 18%, and up to 35% for some workloads, while requiring around 6 KB-900 KB more storage than the baseline to support a wide range of memory sizes (from 4 GB up to 512 GB).

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First and foremost, I would like to thank my supervisors, Dr. Mikel Luján and Dr. Jim Garside, for their mentorship, dedication, and support throughout my PhD study. Mikel has not only been a constant source of excellent, technically sound advice, but also a joy to work with. He has always been a great source of motivation and my PhD study would not be possible without his financial and educational support. Jim has always been patient and a supportive supervisor spending a significant amount of time and effort supervising me. His technical advice improved my research quality and his excellent comments and feedback helped me improving my writing significantly.

I have learned a lot from my fellow graduate students, friends and colleagues at Advanced Processor Technologies (APT) group in the school of computer science at The University of Manchester. Specially, I would like to thank John Mawer for his excellent technical comments and feedback on my ideas. Also, I would like to thank him for accepting to be one of my thesis readers and appreciate his feedback to improve the content of this PhD thesis. I want to thank my collaborator Dr. Aamer Jaleel, principal research scientist at Nvidia, whose advice had an important role to develop the evaluation methodology of this PhD thesis. I would like to thank Professor Stephen Furber and Professor George Constantinides for taking the time to be my thesis examiners.

On a more personal note, I would like to thank my family in IRAN for their unwavering support and encouragement. I would like to thank my parents for their support, both emotionally and financially, and encouraging me to pursue a PhD. Finally, I want to thank my wife, Farideh, for her love, support, patience and faith. Finishing this PhD thesis was not possible without her unwavering support and encouragement.

Chapter 1

Introduction

At least as early as in 1995 researchers have postulated that the performance improvement of computers will cease 'soon' [WM95]. They reasoned that, at some point, processors would become much faster than memory and thus the program execution time would depend on the memory performance and how fast main memory can feed the data required by a processor. They referred to this situation as "hitting the memory wall" [WM95].

For decades, continuous advancement in device technology has improved processors' performance by increasing the clock frequency. However, as Moore's law is coming to an end, the scaling of transistors is no longer a reliable solution for further improvements of future processors. Therefore, the processor industry shifted to multicore systems in the past decade to take advantage of thread-level parallelism to keep the efficiency of modern processors up.

Nowadays, as was postulated, system performance is not normally processor-bound. In this situation, the main bottleneck is shifted toward the memory hierarchy and especially the main memory. Figure 1.1 presents a typical memory hierarchy. Typically, the main memory is populated with DRAM modules that, in general, operate at a lower frequency than the processor. In this situation, it is crucial to utilise the maximum bandwidth offered by DRAMs to minimise the degradation of overall system performance.

DRAM stands for Dynamic RAM (Random Accessed Memory). RAM refers to a memory device that allows a data block to be read or written in almost the same amount of time regardless of the address of the block. Despite this, a DRAM is not truly randomly accessible. Depending on the order of different memory accesses to the data blocks, the access latency might vary significantly. Due to the internal structure

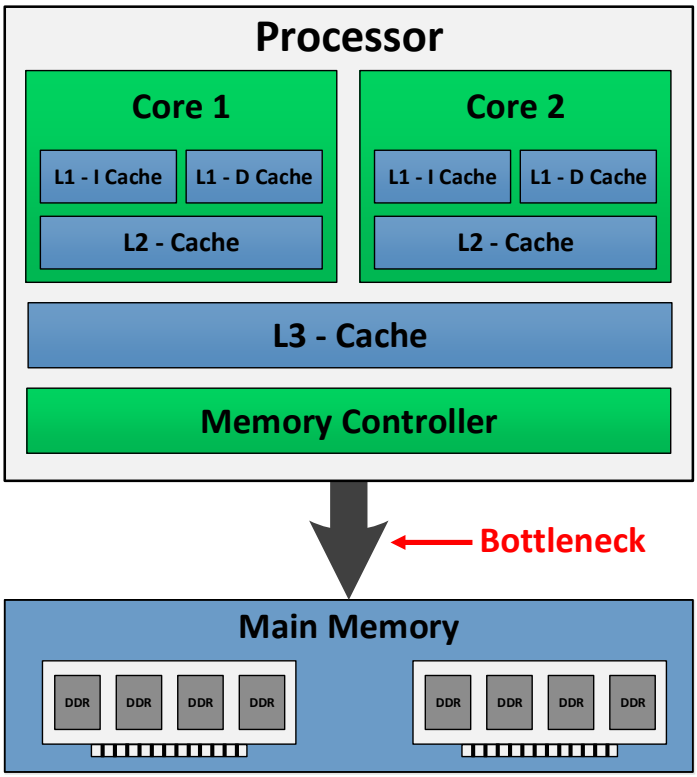


Figure 1.1: Memory hierarchy.

of DRAMs, described in Chapter 2, the performance of DRAMs is dependent on the application behaviour and memory access patterns. Moreover, the high level of parallelism in multicore systems increases the level of randomness of the memory access patterns that makes the DRAMs' performance even more unpredictable.

The sensitivity of DRAMs to memory access patterns at run-time affects all the efficiency aspects of the memory system from *Performance* to *Power* including *Reliability*. In the next section, memory challenges are discussed in more detail.

1.1 Memory Challenges

At a high level, a DRAM device is organised in a 3D structure consisting of multiple Banks, Rows and Columns (Figure 1.2). To access a DRAM module specific timing constraints complying with Joint Electron Device Engineering Council (JEDEC) standard [SPE09] have to be respected. For instance, two consecutive accesses to different rows within the same bank (i.e. *Page Miss*) must be delayed allowing DRAMs to close

the previous opened row and activate a new row. On the other hand, consecutive accesses to the same row within the bank delivers lower access latency than the previous example since the target row is prepared to be accessed by the first memory access (i.e. *Page Hit*).

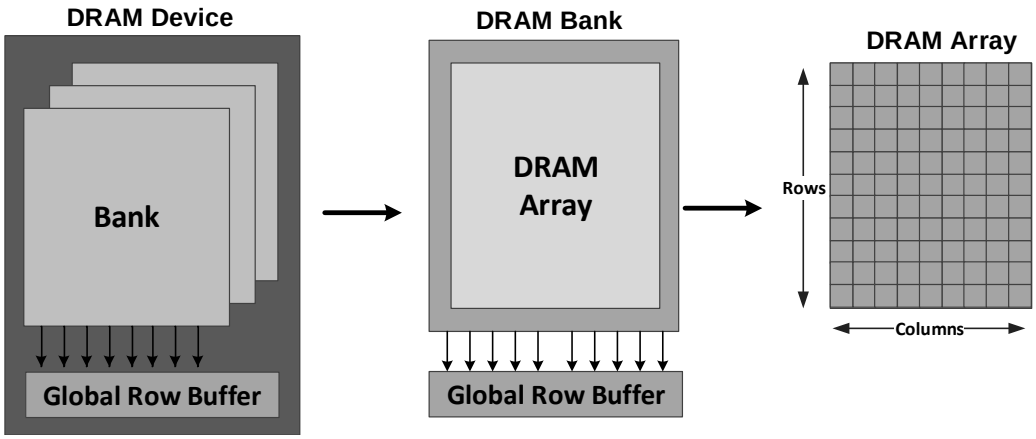


Figure 1.2: Internal structure of a DRAM device.

DRAM controllers are devised to take care of these timing constraints while issuing desired commands (e.g. read/write commands) to a DRAM module. However, the sensitivity of DRAM modules to the memory access pattern has demanded a memory controller with more capabilities than traditional memory controllers. Thus, memory controllers have started to employ algorithms to predict memory access patterns and re-order memory requests to minimise the memory access latency.

Performance, power and reliability of DRAMs depend upon the application run-time behaviour (order of memory requests, timing interval between each request etc.). For instance, as will be discussed in chapter 2, *Page Misses* are one of the most harmful phenomena that affect the efficiency aspects of a DRAM. The most obvious drawback of a Page-Miss is increasing the memory latency, and consequently, degrading the performance. Also, a Page-Miss increases the power consumption as it imposes extra DRAM operations, precharging the previously opened row and activating a new row. Moreover, depending on the occurrence pattern of Page-Misses, they might affect the reliability of DRAMs by corrupting the data in the specific DRAM cells (i.e. *the RowHammer* phenomenon [KDK+14]). Considering this phenomenon in DRAMs and its dependence on the memory access behaviour, the focus of designing modern memory controllers is to detect and reduce the possibility of page-miss occurrence at run time.

Although it is not possible to know the future access pattern, it is possible to make a prediction based on past behaviour. In principle, gathering more (statistical) information may enable better prediction. A complete access trace over the size of DRAM is infeasible as it would require storage of the same order as the memory itself. The problem is to discern what can be collected at a sensible cost – in energy and processor die area – to give 'adequate' prediction accuracy.

Probably the most challenging issue while designing a memory controller is considering that the memory controller is a part of the processor die area. This means that increasing the complexity of memory controllers will increase the overall processor area which may not be economical. This constrains the accuracy and scalability of scheduling and prediction algorithm implemented. On the other hand, the scalability of emerging modern memory technologies such as DDR4 [Mic14b, Mic14a, LCC+07, SNS+13] or Hybrid-Memory Cubes (HMC) [Mic14d, Mic14e, Paw11] demands new designs of future memory controllers. For instance, HMC, introduced by Micron, is a 3D stacked multibank DRAM that provides 15× more bandwidth than conventional DDR3 modules and requires 70% less energy and 90% less space than existing modern memory technologies [Mic14d]. Moreover, the 3D structure of this memory system makes it extremely scalable in comparison with normal DRAM systems.

Overall, considering the demands for a more scalable and efficient memory controller on future computer architectures motivated this PhD thesis to improve the performance of key components of a traditional memory controller.

1.2 Contributions

The focus of this PhD thesis is to improve the workload adaptivity of DRAM controllers to improve the *Performance* and *Reliability* of memory systems. To this end, a set of contributions is presented to redesign key components of a traditional DRAM controller. DRAM controllers leverage various page closure policy predictors (to leave a page open or close a page in advance) to mitigate the performance and power degradation imposed by *Page-Misses*. However, designing an efficient page closure policy predictor requires extra cost of implementation (i.e. area and power) and increases the complexity of DRAM controllers. This thesis proposes an efficient and *scalable* encoding-based technique which can improve the performance of DRAMs' page closure predictors with a lower cost of implementation in comparison with state-of-the-art techniques. As a case study, HAPPY was integrated with a state-of-the-art monitor –

the Intel-adaptive open-page policy predictor employed by the Intel Xeon X5650 CPU [Int] – and a traditional Hybrid page policy. These are evaluated intensively across wide range of workload mixes consisting of single-thread and multi-thread applications. The experimental results show that using the HAPPY encoding applied to the Intel-adaptive page policy can reduce the hardware cost by 5× for the evaluated 64 GB memory (up to 40× for a 512 GB memory) while maintaining the prediction accuracy. Effectively, HAPPY can achieve a similar (or slightly better) performance to existing high performance industry and academic predictors at much smaller hardware overhead.

Data placement in DRAMs, determined by how the physical addresses are mapped across the 3D space of channel, bank and row, has a significant effect on the performance. Memory controllers use fixed and runtime-agnostic address mappings to translate the physical address requested by a processor to the internal structure of DRAMs. However, a fixed address mapping scheme cannot guarantee to deliver optimum data placement for a range of workloads. This thesis presents DReAM (Dynamic Rearrangement of Address Mapping), a novel hardware technique based on approximating the entropy of each memory address bit for a set of memory requests, to generate a workload specific address mapping at run-time. To re-arrange the address mapping dynamically at run-time DReAM needs to support the online-data migration imposed by changing the address-mapping scheme. DReAM investigates different hardware-based solutions for data-migration with different levels of complication. The proposed solutions were evaluated over a wide range of mapping-sensitive and mapping-insensitive workload mixes. The experimental results show that the on-the-fly detected workload specific address mapping scheme produced by DReAM improves the performance of the memory system in comparison with the best static baseline address mapping evaluated in this PhD thesis. Overall, DReAM outperforms the permutation-based address-mapping scheme (the best evaluated baseline) by 5%, on average, and up to 28% across all the workloads. DReAM is complementary to existing schedulers in memory controllers and is the first on-the-fly mechanism capable of generating workload specific address-mappings without requiring the running applications to stop.

Due to the volatile nature of DRAM memory controllers need to issue refresh commands at specific time intervals to avoid losing stored data. Another not so well known means of losing or corrupting stored data within a refresh interval is to have a sequence of memory requests requiring a DRAM row to be activated very frequently. This can corrupt the data in the rows that are physically adjacent to the accessed row; the *Row*

Hammer effect. The data corruption is due to electrical disturbance and segmented memory or page protection cannot guarantee isolation of two or more programs when they are using memory mapped to adjacent physical rows. Moreover, ECC modules are not very efficient in this situation since they cannot detect multi-bit errors. This thesis proposes ARMOR which is a novel hardware technique that improves memory controllers by detecting which specific rows are at risk of being "hammered" without interfering with the processors. In addition, ARMOR can detect and prevent row hammer errors with minimal performance overhead and hardware requirements. The experimental results over a wide range of memory intensive workloads show that ARMOR incurs virtually no execution time overhead for the workloads. When buffers are added to the memory controller to capture such hammered rows, the execution times can also be improved with small hardware costs.

Finally, this thesis integrates *HAPPY*, *DReAM* and *ARMOR* as a part of a uniform system to develop an 'adaptive' memory controller. Figure 1.3 presents a high level overview of such a memory controller. The performance of this adaptive memory controller is compared against a state-of-the-art memory controller configured with the best parameters achieved in each Chapter.

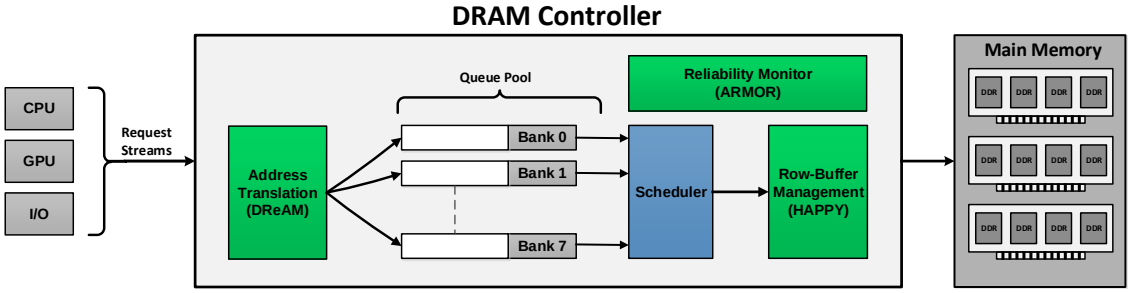


Figure 1.3: An abstract view of the developed adaptive memory controller.

1.3 Thesis Organisation

The rest of this thesis is organised as follows. Chapter 2 provides general background and motivation of this PhD thesis. Chapter 3 presents the experimental methodology. The three techniques outlined in Section 1.2 each have a chapter dedicated to them; these are largely intended to be standalone studies of each technique. Chapter 4 presents HAPPY, a hybrid address-based page closure policy to mitigate the effect of *Page-Misses* in DRAMs. Chapter 5 proposes DReAM, a highly efficient technique that

dynamically changes the address-mapping scheme of DRAMs at run-time. Chapter 6 describes ARMOR, an accurate hot-row detector scheme to improve the reliability in DRAMs. Chapter 7 integrates HAPPY, DReAM and ARMOR to develop an adaptive memory controller and investigates the overall performance improvement of the memory system taking advantage of all three techniques proposed. Finally, Chapter 8, concludes this PhD thesis and discusses future work.

Chapter 2

Background and Motivation

2.1 Overview of DRAMs

DRAMs play a significant role as the main memory of modern computer architectures. Performance, power and reliability of computer systems are highly affected by DRAM architectures. A typical DRAM module, as can be found in a general purpose computer, is pictured in figure 2.1. More specifically, this figure depicts a Dual In-line Memory Module (DIMM) that consists of multiple DRAM devices (i.e. black rectangles) plus corresponding circuitry. In general, due to the internal structure of DRAM devices, the performance of these memory systems depends on the application behaviour. This means that, depending on the internal state of DRAMs and the memory access pattern, the response time of the overall memory system can vary significantly. This phenomenon makes the overall performance of computer systems unpredictable.

Figure 2.1: A memory module [eTe14].

Figure 2.2 presents a high level overview of a typical computer systems' memory hierarchy. Typically, modern processors consist of multiple cores. Each core has its own private caches (in figure 2.2 L1 and L2 are private) and shares the Last-Level

Cache (LLC) with other cores in the system. The next level of memory hierarchy is the main memory consisting of DRAM modules which are connected to the processor through a *Memory Controller*.

This figure provides some insight into the memory latency expected for different levels of memory hierarchy. Typically, accessing off-chip memory (DRAMs) is a very expensive process due to the high latency of IOs. In this situation, the dynamic behaviour of the DRAM modules can impose an extra latency on the off-chip memory accesses which degrades the overall system performance.

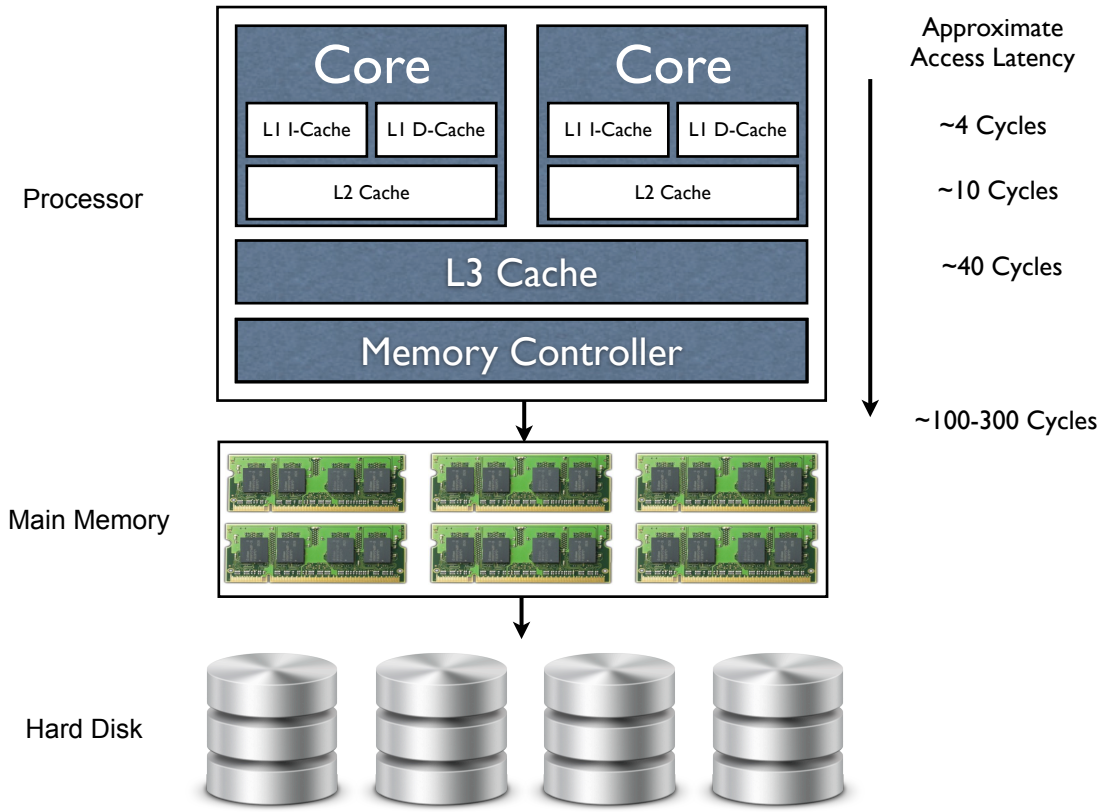


Figure 2.2: A typical computer system memory hierarchy.

To get a better insight into the main reason behind this dynamic behaviour of DRAM systems, the internal organisation of DRAMs will be discussed in more detail in the following sections.

2.2 DRAMs: Basic Circuits and Architecture

In this section, basic DRAM architecture and organisation will be discussed in detail.

2.2.1 DRAM Cell

A DRAM cell is the smallest storage unit of a DRAM device. A schematic representation is shown in figure 2.3. It consists of a capacitor connected to a bit-line via a transistor, which is controlled by a word-line. When the word-line is asserted the capacitor is connected to the bit-line. If the bit-line is driven when word-line is asserted then the value will be stored on the capacitor, a write operation.

To read the cell the bit-line is first precharged, by changing the line to a known driver voltage and then disconnecting the driver. When the word line is asserted if both the bit-line and the capacitor are at the same voltage then the bit-line potential will remain unchanged; if, however, they are different charge sharing will occur resulting in a detectable change of voltage on the bit-line. The presence or absence of this change indicates the state of the bit. It should be noted that the charge sharing will also cause the capacitor's voltage to change, so after reading, the voltage has to be restored by driving the cell back to its original value. Furthermore, over time the charge on the capacitor will leak away, so it needs to be refreshed regularly hence the name dynamic memory.

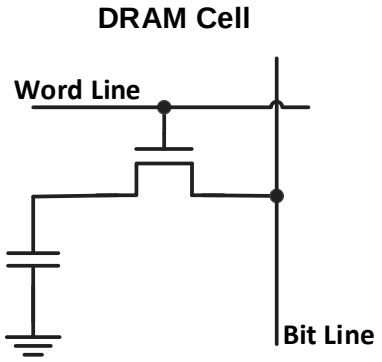


Figure 2.3: A basic DRAM cell structure.

2.2.2 DRAM Array

A DRAM array consists of DRAM cells populated in 'Rows' and 'Columns'. Figure 2.4 presents an abstract structure of a DRAM array. As this figure depicts, word-lines are connected to all the transistors located in a horizontal line (a DRAM Row). This means that raising the word-line will activate all the transistors in one row. Similarly, bit-lines are connected to all the DRAM cells located in a vertical line (a DRAM Column). This implies that, at any given time, only one row can be activated to avoid collision of capacitors' charges. Rows and Columns are separately addressable and some physical address bits are used to address a specific row – and others a column – within a DRAM device. In the literature a DRAM's row may also be referred to as a *DRAM Page*.

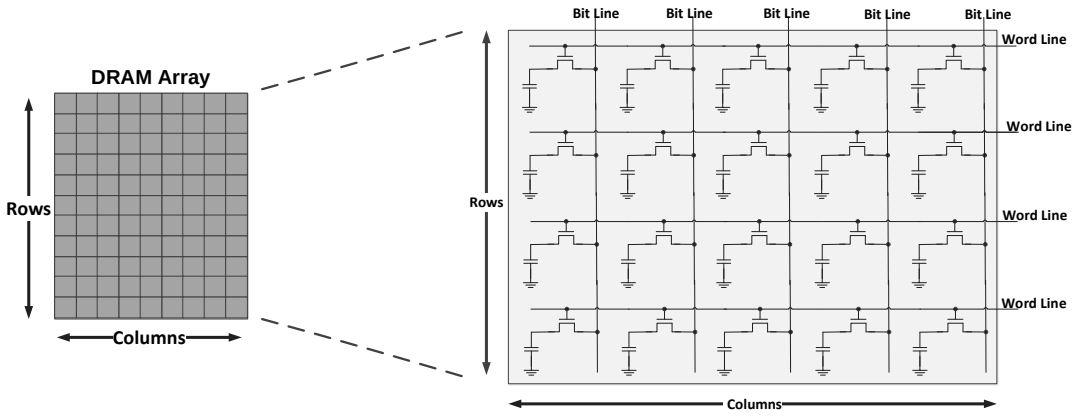


Figure 2.4: A basic DRAM array structure.

2.2.3 DRAM Bank

A DRAM Bank comprises a DRAM array plus a Sense Amplifier (or Row Buffer). During normal DRAM's operation, a row and column decoder are used to decode the accessed row and column addresses to internal row and column IDs. In the first step, a DRAM device will bring the entire accessed row to the sense amplifier which amplifies the signals and holds the row's state in a latch. Then the desired data can be accessed by decoding the column address and a read or write operation can be carried out. Figure 2.5 depicts an overview of a DRAM bank organisation. Typically, DRAM banks have a relatively narrow data bus (e.g. 4-16 bit) which means for each read/write request only part of the requested data can be provided by each bank. This is because of implementation limitations and challenges such as maximum current limitation etc.

Moreover, since accesses from the sense amplifier are faster than from the DRAM array, it is usual to burst data serially from the sense amplifier for a specific size such as a cache-line size. This improves the bandwidth but mandates that adjacent addresses are in the same row.

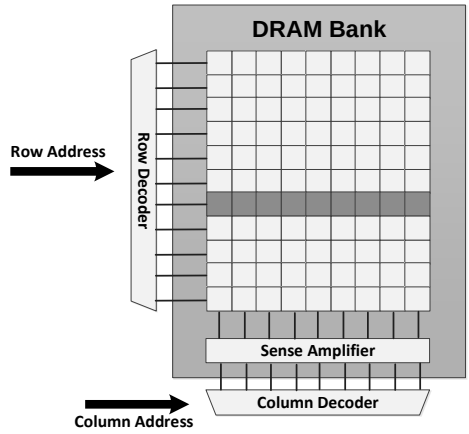


Figure 2.5: A DRAM bank structure.

2.2.4 DRAM Device

A DRAM Device (black rectangle in figure 2.1) comprises multiple DRAM banks which can be accessed in parallel. However, since the existing banks within a DRAM device share resources such as the communication bus, address decoders etc. only one bank at a time can transfer data to the output world. This means that the transfer rate of each device is limited to the transfer rate of a DRAM bank which, as discussed, is about 4-16 bit wide (depending on DRAM model).

2.3 DRAMs: Memory Access Protocol

To access a DRAM a set of processes, manifested by particular commands, must be used. Each of these has its own timing constraints. Table 2.1 presents some of these timing parameters [JNW10]. In general, a combination of a specific set of commands along with the timing constraints depicted in Table 2.1 construct the DRAM access protocol which will be discussed in this section.

Timing Parameters	Description
T_{Burst}	Data burst duration. The time period that a data burst occupies on the data bus. Typically 4 or 8 beats of data. In DDR SDRAM, 4 beats occupy 2 full clock cycles.
T_{CAS}	Column Access Strobe latency. The time interval between column access command and the start of data return by the DRAM device(s). Also known as T_{CL}.
T_{CCD}	Column-to-Column Delay. The minimum column command timing, determined by internal burst (prefetch) length.
T_{CMD}	Command transport duration. The time period that a command occupies on the command bus as it is transported from the DRAM controller to the DRAM devices.
T_{CWD}	Column Write Delay. The time interval between issuance of the column-write command and placement of the first data on the data bus by the DRAM controller.
T_{FAW}	Four (row) bank Activation Window. A rolling time-frame in which a maximum of four-bank activation can be engaged. Limits peak current profile in DDR2 and DDR3 devices with more than 4 banks.
T_{RAS}	Row Access Strobe. The time interval between row access command and data restoration in a DRAM array. A DRAM bank cannot be precharged until at least T_{RAS} time after the previous bank activation.
T_{RC}	Row Cycle. The minimum time interval between accesses to different rows in a bank.
T_{RCD}	Row to Column command Delay. The time interval between row access and data being ready at sense amplifiers.
T_{RFC}	Refresh Cycle time. The minimum time interval between Refresh and Activation commands.
T_{RP}	Row Precharge. The time interval that it takes for a DRAM array to be precharged for another row access.
T_{RRD}	Row activation to Row activation Delay. The minimum time interval between two row activation commands to the same DRAM device. Limits peak current profile.
T_{RTP}	Read to Precharge. The time interval between a read and a precharge command.
T_{WR}	Write Recovery time. The minimum time interval between the end of a write data burst and the start of a precharge command. Allows sense amplifiers to restore data to cells.
T_{WTR}	Write To Read delay time. The minimum time interval between the end of a write data burst and the start of a column-read command. Allows I/O gating to overdrive sense amplifiers before the read command starts.

Table 2.1: Summary timing constraints of DRAM [JNW10].

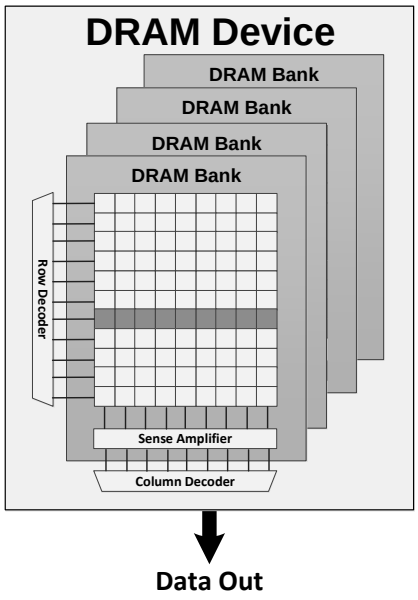


Figure 2.6: A DRAM device structure.

2.3.1 Basic DRAM Commands

In this section, five basic and crucial DRAM commands will be discussed; *Row Access*, *Column-Read*, *Column-Write*, *Precharge* and *Refresh*.

Row Access Command

This is an initialisation command that needs to be issued to move data from the DRAM's cells in the DRAM array to the sense amplifiers (Row-Buffer) and to restore data back to the DRAM array (as part of the same command). It is also known as an *Activation Command*, which is accompanied by a subset of the address bits to select the row. Considering the main operation of this command, there are two main timing parameters associated with it: *Row Column Delay* (T_{RCD}) and *Row Access Strobe latency* (T_{RAS}). T_{RCD} is the time it takes to move data from DRAM cells to the sense amplifier. After T_{RCD} the data is ready for read and write operations. In this situation, although data is ready to access in the Row-Buffer it is not fully restored back to the DRAM cells. T_{RAS} specifies the delay required to restore data to the DRAM array from the row access command. The sense amplifiers cannot be precharged until T_{RAS} has passed. Figure 2.7 presents the associated timing parameters to the row access command.

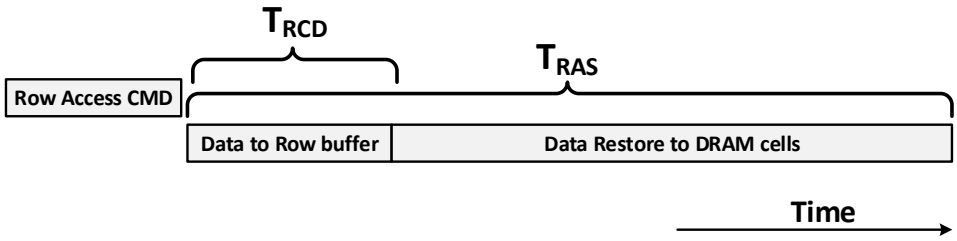


Figure 2.7: Row access command timing.

Column-Read Command

This command transfers selected data from the sense amplifiers over the data bus to the memory controller. There are three timing parameters associated with this command; *Column Access Strobe latency* (T_{CAS}), *Column-to-Column Delay* (T_{CCD}) and *Data burst duration* (T_{Burst}). T_{CAS}, also known as T_{CL}, is the time it takes after a column read command is issued for the DRAM device to place data on the data bus. Internally, DRAM devices move data in a short and continuous burst. The duration of these bursts is called T_{CCD}. Similarly, the data will be placed on the data bus in bursts but with a longer burst period which is called T_{Burst}. Figure 2.8 depicts the associated timing parameters to the column read command.

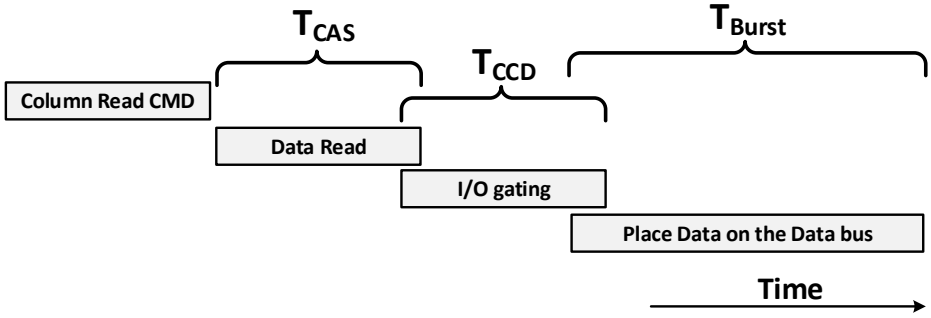


Figure 2.8: Column read command timing.

Column-Write Command

The column-write command is fairly similar to the column-read command with the difference that the direction of data movement is reversed. This command moves data from the memory controller to the sense amplifier. One specific timing parameter associated with this command is *Column Write Delay* (T_{CWD}). This specifies the time between issuance of a column-write command and placing of the data on the data bus

by the memory controller. Moreover, there are two other timing parameters that correspond to the column write command in specific situations; *Write Recovery* (T_{WR}) and *Write-To-Read* turnaround (T_{WTR}). The T_{WR} is the time it takes to transfer the written data to the DRAM array. This timing must be respected in the case of a precharge command that follows the column write command. In addition, since both read and write operation use a common data bus there is a timing constraint that must be taken into consideration when changing the direction of the I/Os. T_{WTR} is the time that it takes to release the I/O resources by the write command. This timing must be respected in the case of a read command that follows the write command. Figure 2.9 shows the corresponding timing parameters to the column write command.

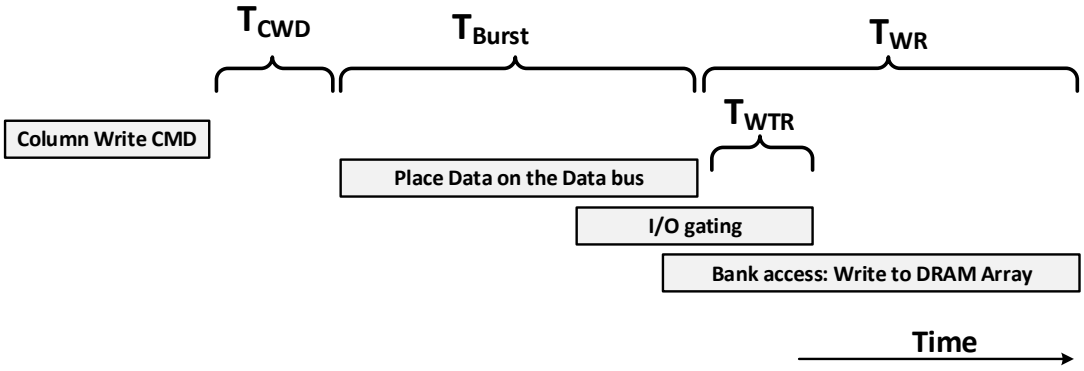


Figure 2.9: Column write command timing.

Precharge Command

It has been discussed that each bank within a DRAM device has only one Row-Buffer and it has been explained that the first step to access a row is the activation process that moves data from DRAM array to the row-buffer. Thus, the located data in the row-buffer is ready to perform a read/write operation. In this situation, accessing a different row within the same bank is not possible since the row-buffer is already occupied. The *Precharge Command* solves this problem by resetting the row-buffer and corresponding bit-lines. This prepares them for another row-access command within the same bank. The associated timing parameter for this command is called *Row Precharge* (T_{RP}). T_{RP} is the time it takes to precharge the row-buffer and corresponding bit-lines properly after the assertion of the precharge command. Considering the T_{RP} described here and the *Row Access Strobe latency* (T_{RAS}) described before, a new timing parameter can be defined as *Row Cycle* ($T_{RC} = T_{RAS} + T_{RP}$) which is the timing constraint

of accessing two different rows within the same bank. Figure 2.10 presents the timing parameters associated to the precharged command.

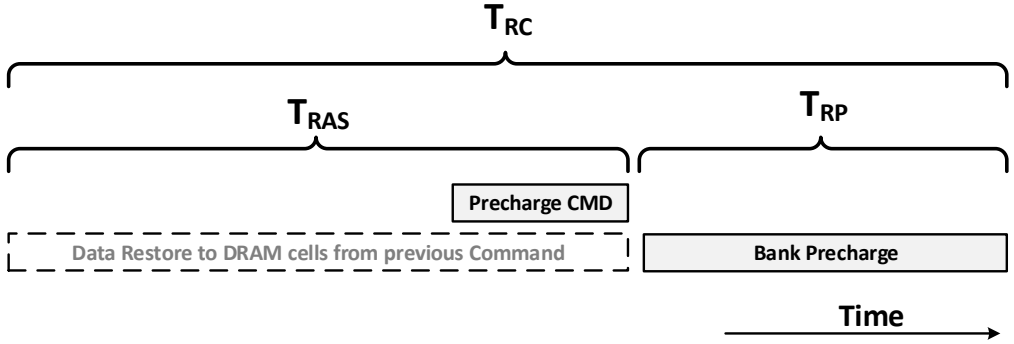


Figure 2.10: Precharge command timing.

Refresh Command

Due to the dynamic nature of DRAMs and considering the basic structure of a DRAM cell presented in figure 2.3, the electrical storage in DRAM cells is not persistent. It means that a DRAM cell will lose its charge, and as a result its stored value, gradually over time. Therefore, to maintain data integrity, the data values must be read and restored to their original value periodically. The purpose of the *Refresh Command* in DRAMs is to perform this periodic read out and restoration of data. In theory, the Refresh process can guarantee data integrity in DRAMs if the time interval of refresh command issuance is shorter than the minimum retention time of the DRAM cells (worst case scenario). For modern DDR3 memory systems the DRAM cell's retention time is around *64 ms*. Thus, the memory controller must issue a refresh command to every row at least every *64 ms*. The associated timing parameter to the refresh command is *Refresh Cycle Time* (T_{RFC}) which is the timing constraint between a refresh command and the next activation to the same row. An overview of the timing parameters associated with the refresh command is presented in figure 2.11.

2.3.2 Row-Buffer Access Classification

Row-Buffer access classifications affect many aspects of a memory system including Performance, Power and Reliability. Considering the basic operation of DRAMs and the associated access protocol discussed in the previous section, a memory access to a DRAM device can be classified in three different categories; *Page-Hit*, *Page-Miss* and

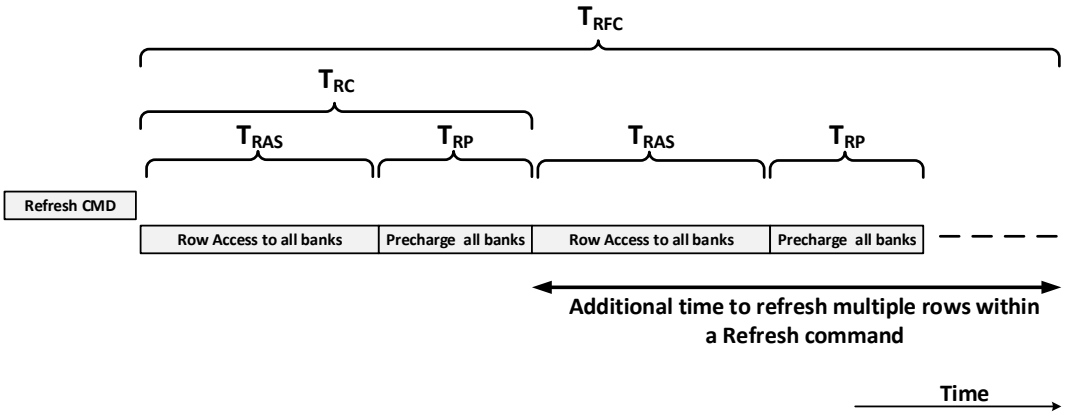


Figure 2.11: Refresh command timing.

Page-Empty. The memory latency of a specific memory access can be significantly different depending on the category that the memory request falls into. In the following, each category and its associated timing parameters will be discussed.

Page-Empty

To perform a read or write operation in a DRAM device the target row must be activated (moved to the Row-Buffer) first. In this scenario if the Row-Buffer is empty the accessed row will be activated and ready for a read/write operation after T_{RCD}. This is called "Page-Empty".

Page-Hit

Considering two memory requests to the same row within the same bank, after the first memory request the target row is open in the Row-Buffer. Thus, the second memory request does not need to wait for T_{RCD} and the read/write operation can be done immediately. This is called a "Page-Hit" which delivers the lowest access latency across the Row-Buffer access classifications.

Page-Miss

Considering two consecutive memory requests to different rows within the same bank, since after the first access the Row-Buffer holds the previously target row then to access a new row the row buffer must first be precharged before the second row can be activated. This imposes an extra T_{RP} to the overall memory access latency. Moreover, the Row-Buffer cannot be precharged until the data is fully restored to the DRAM cells

(T_{RAS}). Therefore, $T_{RC} = T_{RAS} + T_{RP}$ is the memory latency of accessing different rows within the same bank. This is called a "Page-Miss" which imposes the highest access latency in a DRAM device.

2.4 DRAM vs SDRAM

Synchronous Dynamic Random Access Memory (SDRAM) is a version of DRAM. Traditional DRAM has an asynchronous interface which means that it responds to its control inputs as soon as they arrive. On the other hand, SDRAM has a synchronous interface with the processor clock cycle. SDRAM uses this clock to pipeline the incoming commands from processor. This means that the chip can accept new commands while processing the previous command. Therefore, this increases the throughput in comparison with an asynchronous DRAM.

2.5 DRAMs: Memory System Organisation

A basic overview of a DRAM device's internal structure was presented in the previous section. In this section, different organisations of a DRAM device that are currently used in memory systems will be discussed .

2.5.1 DRAM Rank

It was mentioned that each DRAM device has a narrow data bus. Thus, to improve the performance of DRAM modules multiple DRAM devices work in parallel within a *Rank* to provide the bandwidth required by modern processors. Typically, a DRAM rank is designed to support 64-bit data for each read or write request. Furthermore, modern DRAM DDR3 memory systems are restricted to perform read or write operations in bursts (generally limited to a fixed burst size of 4 or 8 items). In this way for each read/write operation 64 bytes of data can be delivered by the DRAM module which is enough to fill a cache-line. Figure 2.12 presents an overview of a DRAM rank organisation with a 64-bit wide data bus.

2.5.2 DRAM Channel

A DRAM channel is the physical connection between a processor and the DRAM modules. Multiple DRAM modules, each of which can consist of 1 to 4 ranks, can

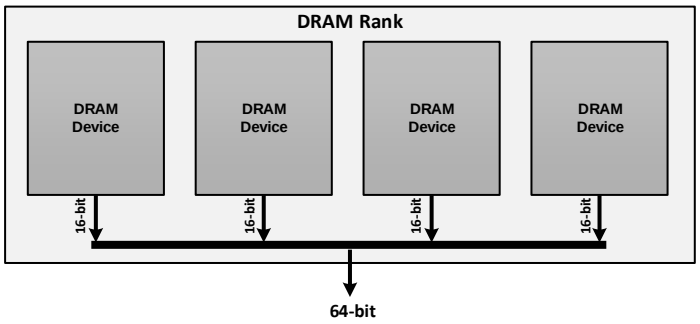
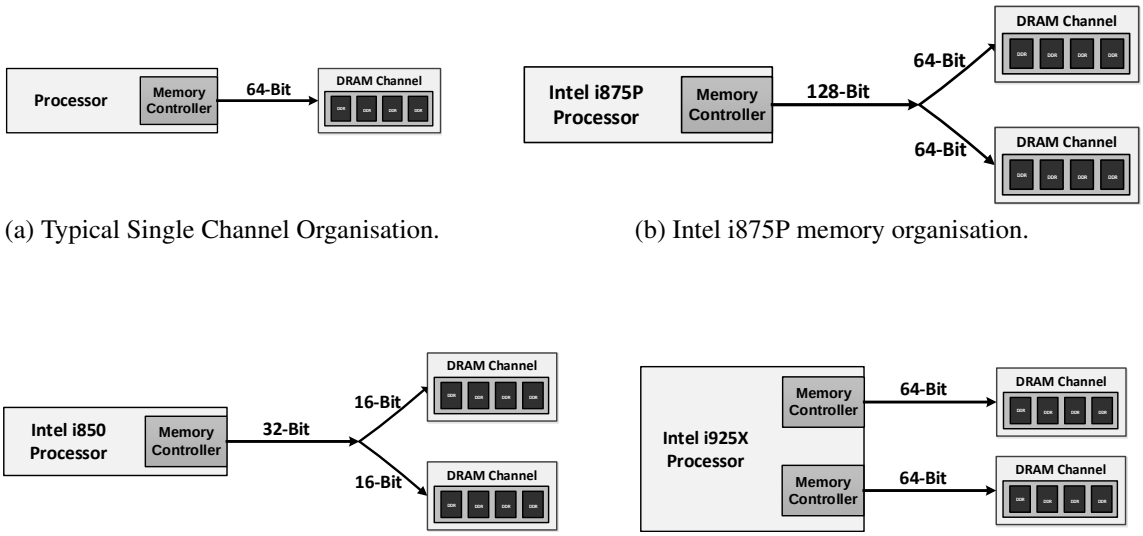


Figure 2.12: A DRAM rank structure.

be placed on a DRAM channel. A processor communicates with DRAM modules through a Memory Controller which will be discussed in detail in the next section. In general, each processor can support multiple memory controllers each of which can manage single or multiple memory channels. Figure 2.13 presents a few examples of existing memory organisations [JNW10]. Figure 2.13a shows a simple processor with one memory controller that manages only one channel. Figure 2.13b depicts the Intel i875P system controller that has one memory controller that manages two channels at the same time, each of which is 64-bit wide. Similarly, figure 2.13c presents the Intel i850 memory system where one memory controller handles two physical channels with narrower data width (i.e. 16-bit) than the Intel i875P. Figure 2.13d, shows the Intel i925X memory system that has two independent memory controllers each of which can manage different 64-bit wide channels. Having independent memory controllers increases the efficiency of the memory system since the memory pressure will be distributed.

2.6 DRAM Memory Controller

The memory controller is the key component of a computer architecture that is in charge of handling communication between the processor and the main memory. In modern computer systems, the memory controller is implemented on the same silicon die as the processor to reduce the communication latency as much as possible. Since, in the multicore era, the main performance bottleneck has moved toward the memory system (rather than processors) the performance of DRAM controllers has become critical. In this section, a high level overview of a modern DRAM memory controller will be investigated and its key components will be analysed in detail. Figure 2.14 presents a abstract view of a DRAM memory controller. This figure is simplified to



(a) Typical Single Channel Organisation.

(b) Intel i875P memory organisation.

(c) Intel i850 memory organisation.

(d) Intel i925X memory organisation.

Figure 2.13: Different industrial memory organisations [JNW10].

show only the main components of such a system.

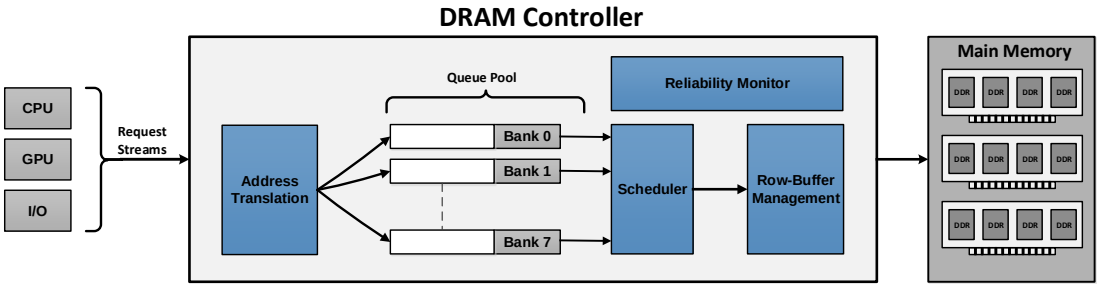


Figure 2.14: An abstract view of a DRAM controller.

The main four components highlighted in figure 2.14 are described as follows.

2.6.1 Address Mapping

The address mapping unit is one of the most important components of a DRAM controller and has a significant effect on the overall performance of the memory system. The first step to service a memory request is to decompose the physical address to the internal structure of the memory system. This process maps given physical address bits (e.g. 32 address bits) to their corresponding Channel, Rank, Bank, Row and Column indices of a given memory organisation. Typically, DRAMs use a *fixed* address-mapping scheme. Figure 2.15 presents an example of how a physical address

is divided to the corresponding DRAM internal structure using two different address mapping schemes.

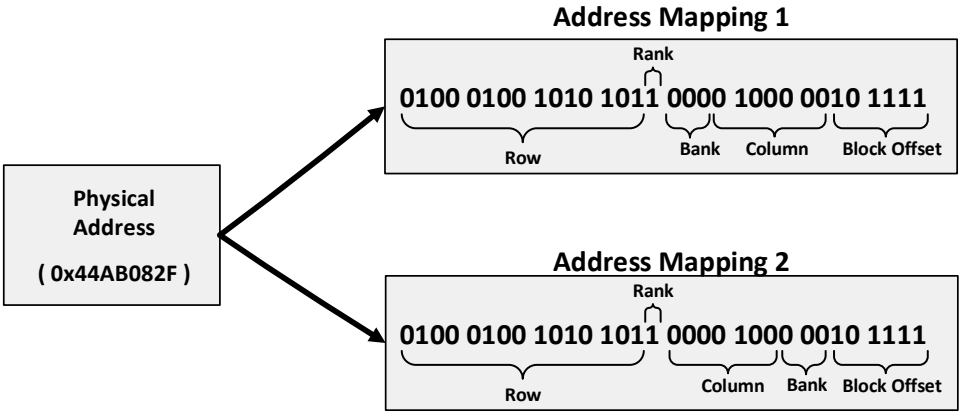


Figure 2.15: Examples of address mapping using different schemes.

As presented in figure 2.15, an address-mapping scheme simply dictates which part of the physical address is used to address specific parts of DRAMs. Although, this process looks very simple, choosing different address mapping schemes can have a significant effect on the overall performance of a memory system. The rationale behind this is that the address translation process has a strong correlation with the data placement in the memory. Therefore, choosing one address-mapping scheme might distribute consecutive memory requests to different banks and choosing another address-mapping scheme might place consecutive memory requests to different rows within the same banks. In the first case, since the multiple banks can be accessed in parallel, the consecutive memory requests can be serviced in parallel but in the second scenario consecutive memory accesses to different rows within the same bank cause a large number of page-misses and, as a result, degrade the memory latency significantly. There are a few novel address-mapping schemes that have been proposed by other researchers in this area to mitigate page-misses [KSJ11, ZZZ00].

2.6.2 Command Scheduling

In the second phase, after the translation process, the translated memory requests will be placed in different request queues depending on their target bank. It has been discussed that to service a memory request multiple commands must be issued (e.g. *ACTIVATION*, *READ/WRITE*, *PRECHARGE*, etc.). To take advantage of the maximum available parallelism in a memory system DRAM controllers employ a *Scheduler* to

re-order the issuable memory request from the request queues. The main aim of the scheduling process is to reduce the page-misses and to increase the page-hits in the system. Therefore, a scheduler can prioritise memory requests targetting the open row within a bank over those that target other rows within the same bank.

In general, there are two traditional scheduling algorithms called First Come First Served (FCFS) and First Ready First Come First Served (FR-FCFS). The FCFS algorithm schedules the memory requests according to their order of arrival. This delivers a poor level of parallelism. On the other hand, FR-FCFS scheduler scans the request queue and prioritises the memory request whose target destination is ready (i.e. memory request targetting an open row within a bank). In addition to the traditional scheduling algorithms, there has been a significant number of other novel algorithms developed to improve the performance of memory systems [EMF+11, KPMHB11, MM08, IFSK12, MKK+12, LWWX12].

2.6.3 Row-Buffer Management

Considering the basic operation of DRAMs explained so far, accessing a row implies transferring the entire row to the row-buffer (sense amplifier). Thus, after performing the read or write operation a DRAM device can either keep data in the row-buffer or return the data to the original row. DRAM controllers employ techniques, collectively called the Row-Buffer Management Policy, to decide either to leave a row-buffer open or close it immediately after it has been accessed. This decision can have a significant effect on the access latency for the next memory request to the same bank. Typically, there are two traditional (static) Row-Buffer management policies that are used by DRAM controllers: Open-Page and Close-Page. A memory controller that employs an Open-Page policy leaves a page open after it has been accessed. This reduces the access latency of the next memory request to the same bank and the same row since the target row is open and there is no extra cost to opening it again (Page-Hit). This policy is desired for workloads with high-locality behaviour. On the other hand, a memory controller that employs the Close-Page policy closes a row immediately after it has been accessed. This reduces the access latency for the next memory request to a different row within the same bank since the row-buffer has been prepared (closed) in advanced for the new row to be activated. This page policy is desired for workloads with random memory access behaviour.

In addition to the static page closure policies, modern memory controllers also take advantage of Hybrid (Dynamic) page policies which are a combination of Open-Page

and Close-Page policies. This type of policy, instead of using a fixed page closure policy, monitors the memory access pattern at run time and switches between static policies considering the locality and randomness of the memory accesses. Although, the idea of Hybrid page closure polices looks promising the main challenge is the compromising of the prediction accuracy and cost of implementation. There are a number of previously published works in this area [KSJ11, ANBD11, XAD09, SM05a, MC07].

2.6.4 Reliability Monitor

The performance of all the components described so far (i.e. Address-Mapping schemes, Schedulers and Row-Buffer management policies) can all affect both the *Performance* and *Power* of the memory system. *Reliability* of memory systems is another crucial factor that must be taken into consideration when designing a computer architecture. Typically, DRAM controllers do not employ a standard standalone Reliability Monitor unit as is highlighted in figure 2.14. The highlighted block in this figure represents a combination of different fault tolerant techniques, such as Error Correcting Codes (ECC), that a DRAM controller implements to take care of reliability issues. Employing smaller memory technology makes DRAM cells more vulnerable to errors. Therefore, modern DRAM controllers employ more resources to implement different fault detection and correction techniques to overcome this issue.

2.7 Performance, Power and Reliability

In this section, considering the basic background on DRAM systems discussed so far, the potential performance and reliability improvement of DRAM-based memory systems will be investigated. Referring to Figure 2.14, it has been highlighted that there are a few critical component in a DRAM controller whose performance has a significant effect on the overall memory systems. In this section, the effect of these key component on the performance and reliability of the overall memory system will be discussed.

2.7.1 Memory Access Pattern

DRAMs' performance, power and reliability are susceptible to the memory access pattern. Due to the heterogeneous nature of the internal structure of these memory

systems, accessing different locations of a DRAM device imposes different latencies and energy consumptions on the overall memory system. In addition, following a specific access pattern while performing read/write operations to the DRAM arrays can affect the reliability of the memory system.

Data placement is one of the factors that affects the memory access pattern and, as a result, the overall performance of the memory system. The key components of a DRAM controller, highlighted in Figure 2.14, will affect the data placement and the memory access pattern in one way or another. In the following sections, the effect of each component on the memory access pattern will be investigated more closely to see how these components can affect the performance, power and reliability of a DRAM device.

2.7.2 Susceptibility to Address Translation

The address translation process is one of the most significant factors in determining the final latency and power for each specific memory request. Considering a sequence of memory requests, each requested address will be mapped to its destination (i.e. channel, rank, bank, row and column) based on a pre-defined address-mapping scheme employed by the DRAM controller. In other words, the address translation process controls the data placement.

Data placement in DRAMs has a strong correlation with the memory access pattern. For instance, placing consecutive cache-lines in a DRAM's row can improve the locality of memory access pattern and placing consecutive cache-lines in different banks can increase the level of parallelism of memory access pattern (depending on the application access pattern). Therefore, if the application behaviour is known in advance, choosing an optimum address mapping scheme can bias the memory access pattern towards minimum page-misses and maximum page-hits. This will improve the memory access latency as well as the power of the memory system. However, memory controllers employ a predefined address-mapping scheme without any knowledge about the application access behaviour. Hence, if the application access behaviour complies with the predefined address-mapping scheme it can improve the memory performance but if it does not, then the performance of memory system can be degraded significantly.

2.7.3 Susceptibility to Page Closure Policy Prediction

The row-buffer management policy (also known as the 'Page Closure Policy') is another critical component that has a strong interaction with the memory access pattern. Considering the traditional page closure policies, leaving a row open (i.e. Open-Page) or closing it immediately after it has been accessed (i.e. Close-Page) can deliver different memory access latency depending on the memory access pattern. For instance, if consecutive memory accesses go to the same row and the memory controller employs an open-page policy the memory access latency will be improved. Using a close-page policy in this case would degrade the performance of the memory system. The hybrid page closure policies try to predict the memory access pattern and switch between open-page and close-page policy at run time. Similarly to the address translation process, page policy prediction outcome can significantly affect the performance and power of the memory system.

2.7.4 Susceptibility to Error Detection

Maintaining data integrity in DRAMs is one the most crucial factors in the area of memory system design. Modern memory systems often employ ECC modules to improve the reliability of DRAMs. To get a better insight into the importance of reliability, some server-grade memory systems accept a 12.5% area overhead just to implement these ECC modules [KDK+14]. One of the harmful kinds of error in DRAMs are those where their possibility of occurrence depends on the application behaviour. Since the application behaviour changes dynamically, the possibility of occurrence of these types of error also changes at run time, which makes it very difficult to capture them. One specific type of error with such a dynamic nature of occurrence is called the *Row Hammer* phenomenon [Mik, Mica, KDK+14] which can be categorised as Disturbance Errors. A Row Hammer phenomenon will happen if the number of activations of a specific row (which is known as Row-Aggressor) exceeds a specific threshold within the refresh interval. In this situation the neighbouring rows (known as Victim-Row) to the row-aggressor may start losing their data. Since an activation stream to a DRAM module depends on the application access pattern, Row Hammer occurrence also depends on the memory access pattern.

2.8 Future Memory Technology and Scalability Challenges

The traditional approach to scaling down memory technology and the advanced development of designing novel memory systems such as HMC, both facilitate the employment of high capacity memory systems in modern computer architectures and data centres. This reveals a new challenge for memory controller designers; *Scalability*. Moreover, some of the recent approaches for data centres, such as keeping the entire database in DRAM, RAMCloud (e.g. 64 *TB* of DRAMs) [Joh], turn the scalability issue into a serious problem for future DRAM-based architectures.

In the previous section, the susceptibility of the key components of a modern DRAM controller to memory access patterns has been discussed. One solution to mitigate this susceptibility is to develop an access pattern aware memory controller that can adapt itself to the application behaviour. Developing a workload adaptive memory controller requires careful monitoring and analysis of the workload behaviour at run time using prediction algorithms and pattern detection techniques. Typically, this imposes extra hardware overhead on the memory controller design. In this situation, the scalability of modern memory systems demands scalable prediction and monitoring algorithms to be implemented in the memory controller.

In the following, some of the state-of-the-art memory technologies available in the market are briefly discussed.

2.8.1 DDR4 DRAM

Double Data Rate Fourth generation Dynamic Random-Access Memory (DDR4 DRAM) is the next generation of DRAM memory that delivers a better performance and power than the previous generation (DDR3). In summary, it has a lower operating voltage (1.2 V), higher clock frequency (up to 1.6 GHz), higher device density (2 Gb-16 Gb), higher number of banks per device (16) and faster burst access by employing up to 4 *Bank Groups (BG)* than DDR3 memory systems [Mic14b, Mic14a, SNS+13, LCC+07].

One of the important features of DDR4 as opposed to its predecessor, DDR3, is its ability to mitigate row hammer error as discussed in Chapter 6. By definition, aggressive activation of a specific row (above a certain threshold) in a DRAM within a refresh interval can cause data corruption in the neighbouring rows. The latest DDR4

modules uses Target Row Refresh (TRR) mode to mitigate row hammering. TRR allows DRAM controllers to request the refreshing of the neighbouring rows of a target address (row aggressor). For instance, TRR mode has been included in the latest 4 Gb Micron devices (MT40A1G4HX-083E) [Mic14c]. Note that there is still a demand for detecting row aggressors before a TRR command to be issued.

Moreover, the DDR3 DRAMs that have been shipped to the market so far do not have inbuilt support to mitigate row hammer errors. To address this issue, Chapter 6 of this PhD thesis presents a technique to prevent row hammer error.

2.8.2 Hybrid Memory Cube (HMC)

HMC is a 3D stacked multibank DRAM that provides 15× more bandwidth than conventional DDR3 modules and requires 70% less energy and 90% less space than existing modern memory technologies. HMCs use Through-Silicon Via (TSV) technology to create a 3D stack of standard DRAM memory dies (typically 4 or 8). It has more data banks than traditional DRAMs and its unique 3D structure makes it scalable in comparison with normal DRAM systems. The HMC memory controller is integrated into the memory package as a separate logic die [Paw11, Mic14d, Mic14e].

2.8.3 Non-Volatile Memory

Non-Volatile memories are another category of promising future memory technology. For instance, Phase-Changed Memory (PCM) is a type of non-volatile random-access memory that, due to its physical nature, can switch between crystalline and amorphous states in response to heating. PCMs take advantage of this alterable physical and electrical properties to store information in memory systems and since there is no need for electrical power to hold a PCM's physical state it is a non-volatile memory. Moreover, this type of non-volatile memory is able to achieve some intermediary physical state, between crystalline and amorphous state, that allows them to store multiple bits in a single memory cell [WRK+10, JD04, Mic14f, IBM14].

2.9 Workload Adaptation

According to the discussion in this chapter so far, it can be concluded that all the performance aspect of a DRAM controller can be affected by the application behaviour and the memory access pattern. Therefore, this PhD thesis describes an investigation

into the possibility of developing a workload adaptive DRAM controller, improving the flexibility and adaptivity of individual key components in such a system.

In a high level overview, Figure 1.3 in Section 1 presented how a traditional memory controller (figure 2.14 in section 2.6) will be affected by the outcome of the research that has been carried out while producing this thesis. To sum up, this PhD thesis proposes three different techniques (i.e. HAPPY, DReAM and ARMOR) to improve performance and scalability of three key components of a DRAM controller by increasing the flexibility and adaptivity of each component to the application memory access behaviour at run time. Thereby, by integrating these new adaptive components, this thesis presents a proposal for a novel and scalable workload adaptive memory controller to improve overall performance of the memory system.

Chapter 3

Experimental Methodology

This chapter presents the experimental methodology used in the research presented in this PhD thesis as well as including a brief description of the tools that have been developed to speed up this research process. The methodology presented here is a general overview and highlights the common strategy that has been employed in the next three chapters. However, each of the following chapters might use slightly different memory parameters to investigate the corner cases, depending on the nature of the targeted problem.

3.1 Tools Development

During this research several tools were developed to understand the basic infrastructure of DRAMs, speed up the research process and improve the evaluation phase of the proposed techniques. A brief summary of the two main platforms that have been employed are described below.

3.1.1 A Simple DRAM Simulator

Figure 3.1 presents a screenshot of the simple DRAM simulator developed in the initial stage of this PhD work. The eight lower right 3D graphs depicted in this figure monitor the memory access pattern of eight internal banks of a DRAM. Each memory access to the DRAM is presented as a coloured point in these graphs depending on the access latency of the memory request. The main purpose of developing this platform was to study memory access patterns, in a graphical format, when using different memory configurations. For instance, the effect of different address-mapping schemes

on the memory access pattern can be investigated for different workloads using this tool. Moreover, this tool will accept some of the main DRAM timing constraints as inputs and generate a rough estimate of the execution time for different workloads based on their memory access traces. The experimental results produced by this tool were the main motivation to develop DReAM (Chapter 5), one of the core contributions to this PhD research which is a technique to change the address-mapping schemes at run time.

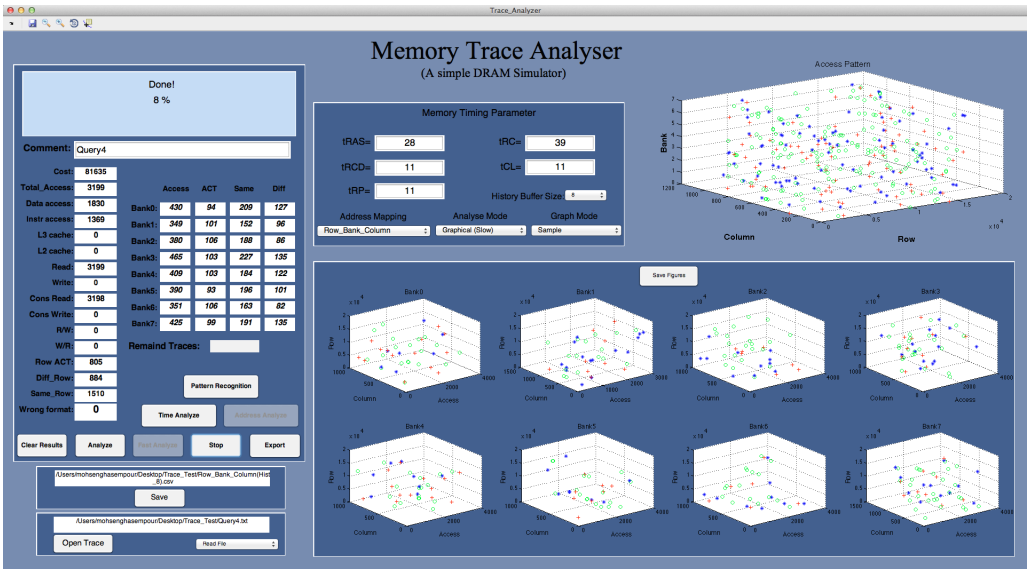


Figure 3.1: A simple DRAM simulator.

3.1.2 A Synthetic Kernel Generator (SKG)

The evaluation methodology is one of the most important parts in the research and can have a significant effect on the final results and conclusions. To debug the simulation infrastructure and improve the evaluation methodology a Synthetic Kernel Generator (SKG) is developed. In general, and more specifically in the computer architecture related research area, there is no single benchmark that can be used to evaluate all the aspects of a computer architecture. Typically, each benchmark has been designed to simulate specific behaviour. For instance, five different benchmark suites were used to evaluate all the techniques proposed in this PhD thesis. However, there are special behaviour and corner cases that are not covered by these standard benchmarks. To investigate these special and extreme cases some synthetic kernels are generated using SKG. This tool provide a flexible platform to generated a wide range of synthetic

kernels with full control on the memory access pattern, memory intensity, access distribution etc. Figure 3.2 presents a high level overview of the internal structure of this tool. This tool was mainly used to generate the synthetic kernels required to evaluate the technique proposed in Chapter 6 (ARMOR).

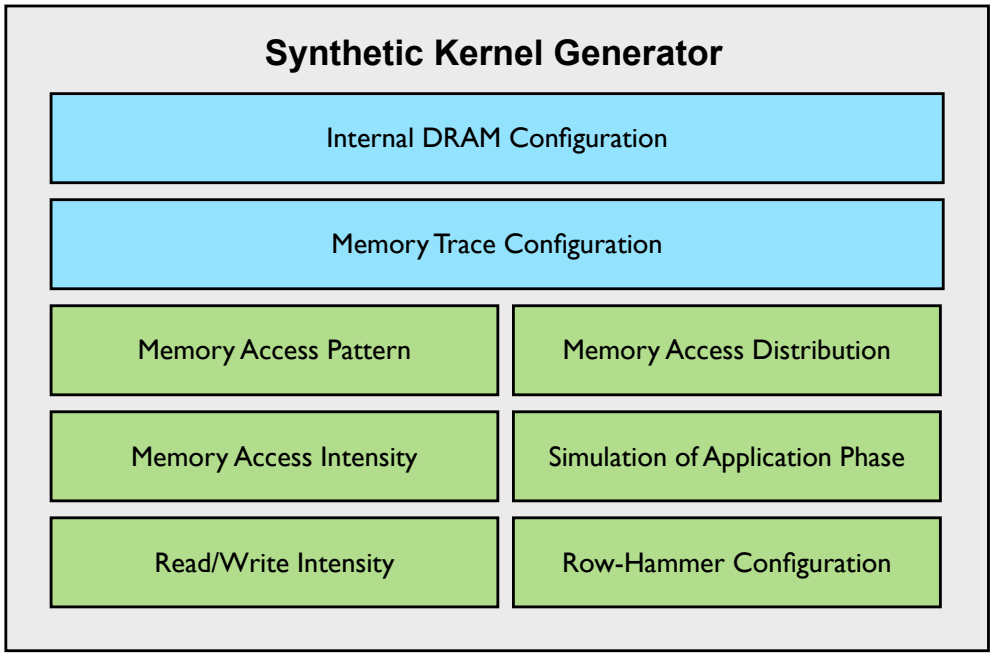


Figure 3.2: Block diagram of the synthetic kernel generator.

A brief summary of the functionality of each block in Figure 3.2 follows:

- **Internal DRAM Configuration**: this module accepts different parameters to configure the internal structure of the DRAMs, such as the number of channels, ranks, banks, rows, columns, cache-line size etc.
- **Memory Trace Configuration**: this module accepts some parameters, such as LLC Misses Per Kilo Instructions (MPKI), Simulation Length or Trace Size, to produce a memory trace with the desired specification.
- **Memory Access Pattern**: this module uses different parameters to produce memory traces with various access patterns (e.g. sequential or random).
- **Memory Access Distribution**: this module manages the memory access distribution across the memory space. For instance, using this module it is possible to generate a memory trace that only uses two specific banks of one specific channel in the system etc.

- **Memory Access Intensity**: this module controls the time interval between memory accesses. Therefore it is possible to tune this time interval to have high or low intensive memory traces.
- **Simulation of Application Phase**: this module makes it possible to change the application access pattern between sequential or random with a fixed or random time interval. Therefore, it is possible to simulate different application phases with different behaviour.
- **Read/Write Intensity**: this module manages the distribution of read and write in the memory traces with a programable ratio. Thus it is possible to generate memory traces that are either write or read intensive.
- **Row-Hammer Configuration**: this module is designed to produce the row hammer effect in the generated memory traces. For instance, using this module it is possible to specify the desired number of row-aggressors with different distribution of occurrence such as, Uniform, Gaussian and Poisson distribution. Thus, this tool can produce different row-aggressors by accessing random rows more frequently than others using different access pattern.

3.2 Evaluation Platform

The Utah Simulated Memory Module (USIMM) [CBS+12] was used as the main evaluation platform for the experiments described in this PhD thesis. The initial version of this simulator was significantly improved and extended to support various address-mapping schemes, page-management policies, row-hammer monitoring etc. USIMM is introduced briefly below.

3.2.1 USIMM: The Utah Simulated Memory Module

USIMM is a trace-based cycle accurate DRAM simulator developed by Utah university [CBS+12]. Initially, it was introduced in a memory scheduling competition in conjunction with one of the top international conferences in the area (ISCA39). The initial version of USIMM supports a FR-FCFS scheduling algorithm. USIMM supported only two static page management policies (i.e. Open-Page and Close-Page) which was extended by this work to support five different dynamic page management

policies described in Chapter 4. Initially, this simulator supported two different address mapping schemes to maximise page locality and bank-level parallelism, this was extended by this work to support two state-of-the-art address mapping schemes (minimalist open-page [KSJ11] and permutation-based page interleaving [ZZZ00]). In addition to the mentioned improvements, USIMM was extended to support ARMOR, the row hammer solution explained in Chapter 6. Finally, USIMM was extended to support specific performance counters to monitor and extract the patterns required by different techniques proposed in this PhD thesis.

3.3 Evaluated Benchmark Suites

The workloads include a wide range of memory intensive applications (48 workloads) from different benchmark suites (PARSEC [BKSL08], SPEC [Dix91], BIOBENCH [AJW+05], HPC and COMMERCIAL) and representative regions of interest for each application. Table 3.1 lists the workloads and their corresponding benchmark suites.

Benchmark Suites			
SPEC		**PARSEC**	**COMMERCIAL**
GemsFDTD_r	astar_B	canneal	comm1
bzip2_1	bzip2_t	streamcluster	comm2
cactusADM_b	gcc_1	blackschols	comm3
gcc_2	gcc_c	facesim	comm4
gcc_cp	gcc_g	ferret	comm5
gcc_sc	mcf_r	fluidanimate	**BIOBENCH**
milc_s	omnetpp_o	freqmine	mummer
soplex_r	sphinx3_a	swaption	tigr
xalancbmk_r	zeusmp_z	**HPC**	
libquantum	leslie	hpc1 - hpc13	

Table 3.1: Standard workloads and benchmark suites.

In addition to the single-thread workloads presented here, each proposal (i.e. HAPPY, DReAM and ARMOR) will be evaluated against multithread workload mixes built by combining the single thread applications. Each chapter considers a different set of multithread workload mixes to evaluate each proposal on desired application access behaviour. For, instance in Chapter 6, the multithread workload mixes are produced to increase the possibility of row hammer error occurrence. Therefore, each chapter includes a table of multithread workload mixes describing which combination of single thread application used in each mix.

3.3.1 Workload Characterisation

As discussed, DRAM performance is susceptible to application access patterns and workload behaviour. Therefore, it is crucial to evaluate the techniques proposed in this PhD thesis over a wide range of workloads with different characterisations. In this section some of these characterisations such as Instruction Per Clock cycle (IPC), LLC Misses (defined as Misses Per Kilo Instructions (MPKI)) and Read Latency of each application are profiled. Figure 3.3 to Figure 3.5 presents the MPKI for all the workloads. These results shows that the evaluated workloads cover a wide range of MPKI (between 2 to 70 LLC misses per 1000 instructions).

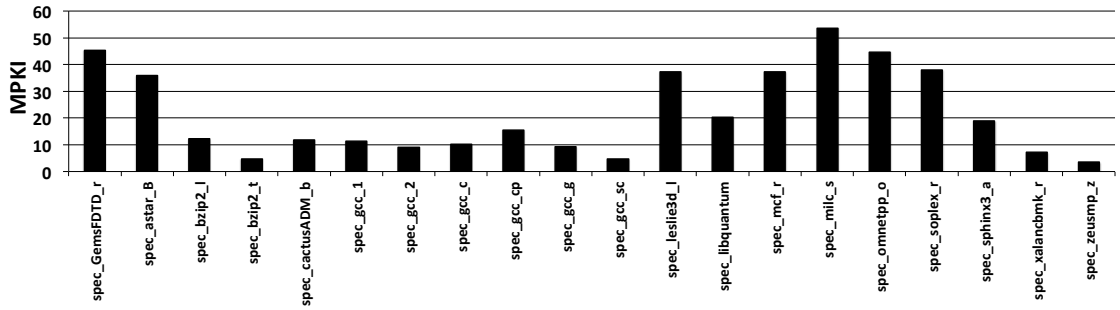


Figure 3.3: MPKI for the SPEC benchmark suite.

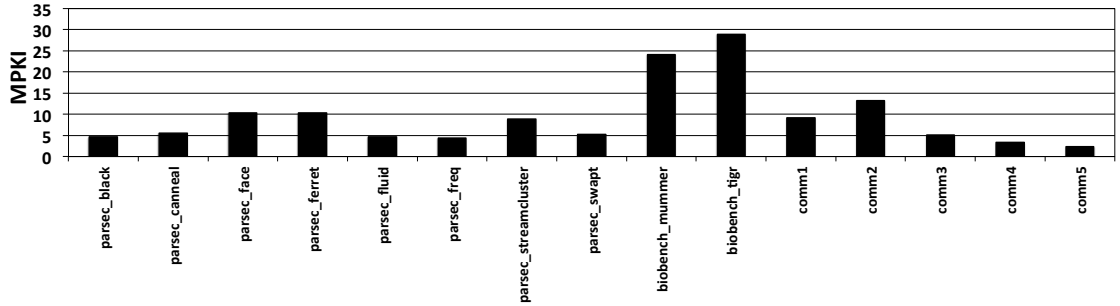


Figure 3.4: MPKI for the PARSEC, BIOBENCH and Commercial benchmark suites.

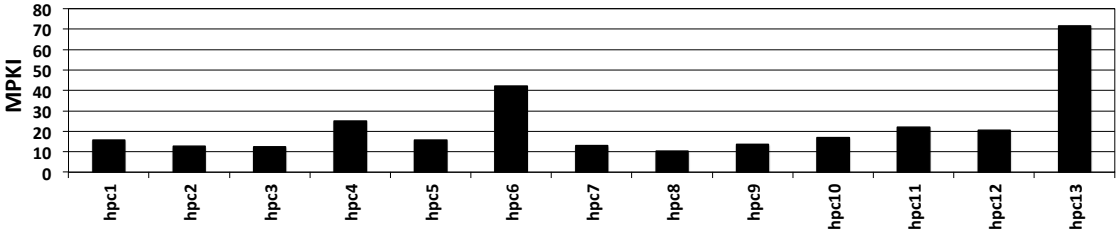


Figure 3.5: MPKI for the HPC benchmarks.

Figure 3.6 to Figure 3.8 depict the IPC for all the workloads running on the x86 architecture. These results show the IPC changed between 0.2 to 1.6 across different

workloads.

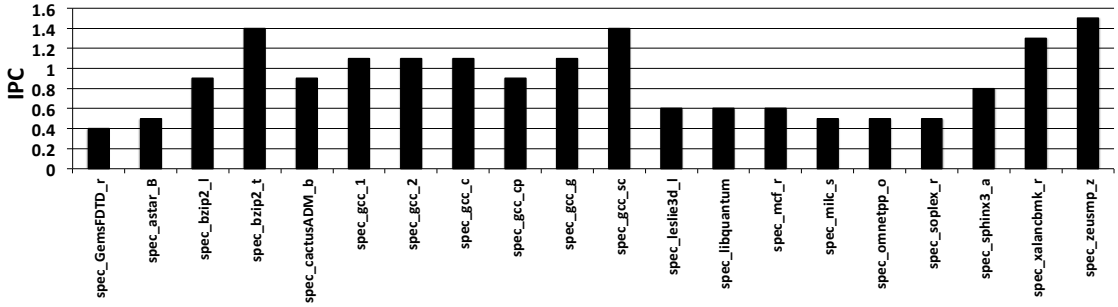


Figure 3.6: IPC for the SPEC benchmark suite.

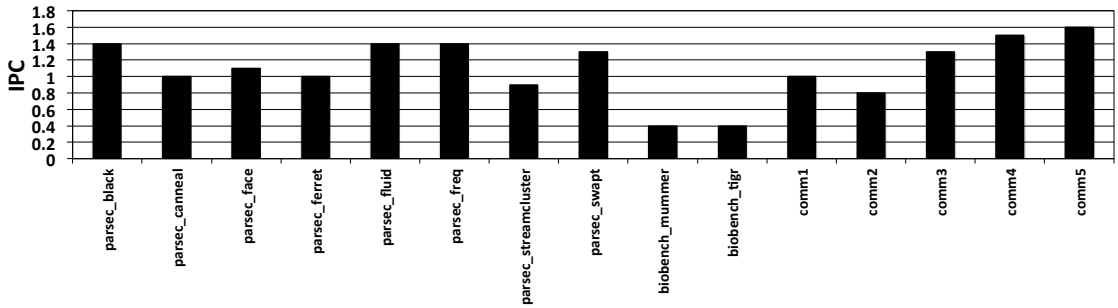


Figure 3.7: IPC for the PARSEC, BIOBENCH and Commercial benchmark suites.

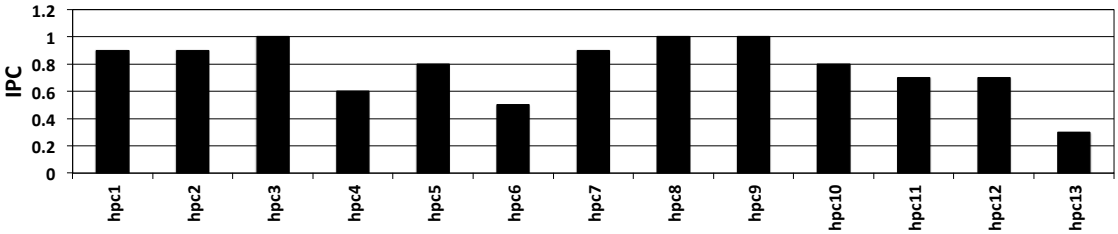


Figure 3.8: IPC for the HPC benchmarks.

Figure 3.9 to Figure 3.11 show the read latency for all the workloads. The experimental results show the read latency of 100 to 200 clock cycles across all workloads.

The general characterisation of different workloads presented in this section shows that a wide range of applications with different characteristics is used to evaluate the techniques proposed in this PhD thesis. However, there are more interesting characteristics that can be extracted from each application that will be investigated in the next three chapters depending on the targeted problem in each chapter.

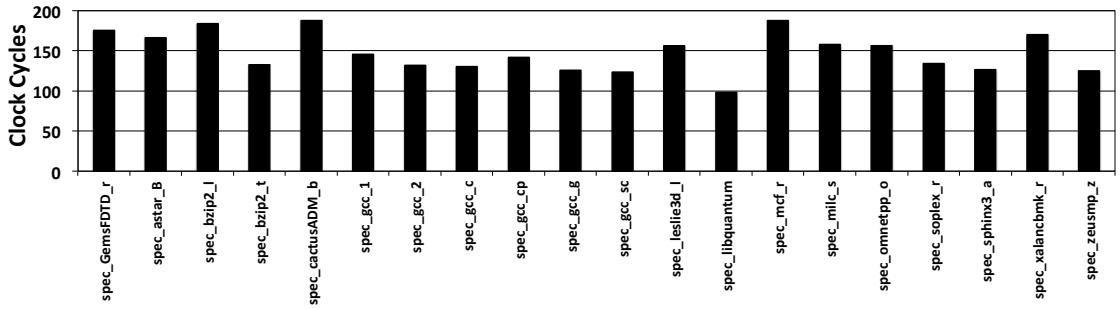


Figure 3.9: Read latency for the SPEC benchmark suite.

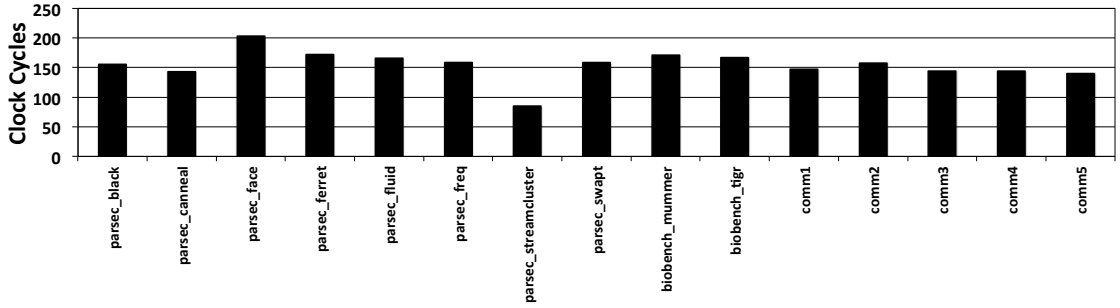


Figure 3.10: Read latency for the PARSEC, BIOBENCH and Commercial benchmark suites.

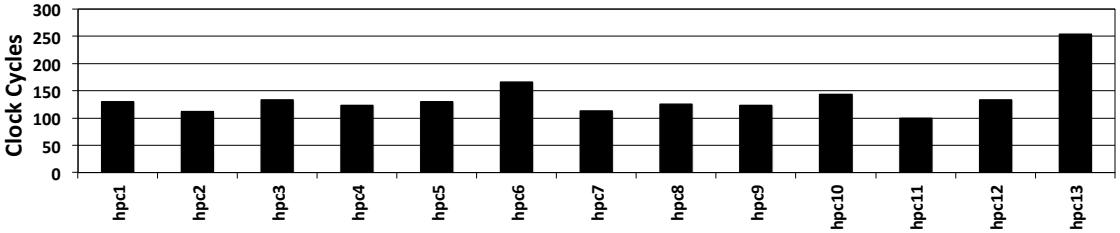


Figure 3.11: Read latency for the HPC benchmarks.

Chapter 4

HAPPY: Hybrid Address-based Page PolicY

4.1 Introduction

The performance of DRAM is sensitive to the memory access pattern of the running applications. Accesses to a DRAM device can be categorised in three ways: page-empty, page-hit and page-miss. If a memory request goes to a bank which has no open row the access called 'page-empty'. In this situation an activation command is required to open the desired row. If a memory request goes to a row which is already open then it is a 'page-hit' and the request can be serviced fairly cheaply. Finally, if a memory request goes to a different row from the open one in a bank it is a 'page-miss' which imposes an extra latency on the memory system; the open row must be closed before the desired row can be opened.

Traditionally DRAM controllers have used a static row-buffer access policy, either *open-page* or *close-page*, to decide whether a row should be left open or closed immediately after their access [JNW10]. For workloads with high locality of accesses open-page works best since the target row is already open and multiple accesses to that row can be serviced with one activation. However, for workloads with more random memory accesses, close-page is a better option. In this case a row will be closed immediately after a memory access so the next memory request within the same bank does not need to wait for the precharge process of the open row. Moreover, neither the open-page nor close-page policy can deliver the 'best' execution time for all workloads due to the dynamic nature of memory accesses. In this situation a *hybrid-page* policy, which is a mixture of open-page and close-page, is more desirable [KDD10].

Different techniques have been proposed to select between open-page and close-page in DRAM memory controllers [KSJ11, ANBD11, XAD09, SM05a]. *Access-based techniques* are those that monitor and keep a history of the row hits and row misses at different granularities in DRAMs to make a prediction of the future page closure policy. On the other hand, *time-based techniques* focus on predicting the optimum time that a row should be left open. In general, these techniques rely on predictors that monitor the number of accesses per row, the number of row hits or row misses, the time between hits or misses etc. to predict the page closure poilicy for each row in DRAM. Intel included two time-based techniques in the Xeon X5650 [Int]. As the size of DRAM increases with data analytic applications, having a fine-grain prediction and monitoring scheme becomes inefficient and not scalable. On the other hand going toward coarse-grain schemes reduces the accuracy of the prediction. A key challenge for page-closure techniques is to balance the hardware overhead and the prediction accuracy. The trend towards keeping entire databases in DRAM, such as RAMCloud (64 TB of DRAM) [Joh] or Facebook using 150 TB of DRAM with memcache [ORS+11], turns the scalability issue into a critical problem for future DRAM systems. This chapter presents a scalable and compact memory address-based encoding technique, called *HAPPY*, that can be employed in DRAM memory controllers. HAPPY is an efficient encoding that reduces the cost of implementation of existing page closure techniques while maintaining the prediction accuracy of the original implementation. As case studies, this chapter shows how to integrate HAPPY with a state-of-the-art implementation – the Intel-adaptive open-page policy employed by Intel [Dod06, Int] – and with a traditional hybrid-page policy. HAPPY and existing page closure techniques are evaluated across a wide range of workload mixes consisting of single-thread and multi-thread applications. The experimental results show that using the HAPPY memory address-based encoding applied to the Intel-adaptive page policy can reduce the hardware cost of implementation by 5$\times$ for the evaluated 64 GB memories (up to 40$\times$ for a 512 GB memory) while maintaining the prediction accuracy. In other words, HAPPY achieves similar, or better, performance to existing high performance industry and academic techniques while requiring less hardware overhead.

4.2 Background on DRAM Page Closure Policy

DRAM controllers employ a page closure policy to alleviate the effect of page-misses on the memory system's performance. The most common schemes are the open-page and the close-page policy. A DRAM that uses the open-page policy will leave the last accessed row open in the row buffer to eliminate the activation cost of the next memory request to the same row. A DRAM that uses the close-page policy will close a row immediately after it has been accessed to eliminate the possibility of getting page-miss for the next memory request [Kee08]. In general the open-page policy is more desirable for systems with high access locality and the close-page policy is desired for systems with highly random access behaviour. Table 4.1 presents the timing cost of page-hits and page-misses when using the static page closure policies.

Page Policy	Page Hit	Page Miss
Open-Page	t_{CL}	$t_{RCD} + t_{CL} + t_{RP}$
Close-Page	$t_{RCD} + t_{CL}$	$t_{RCD} + t_{CL}$
Static Profiling	t_{CL}	$t_{RCD} + t_{CL}$

Table 4.1: Cost of page-hits and page-misses when using different page closure policies.

Motivation: Figure 4.1 depicts the execution time (normalised to open-page policy) of all the workloads that are used in this chapter using open-page and close-page policy. The results show that around 68% of workloads 'prefer' the open-page policy while 32% of workloads deliver a better performance using the close-page policy. According to this figure, a memory system that employs the open-page policy can save up to 18%, in comparison with the close-page policy, when running 'libquantum' from the SPEC benchmark and, at the same time, might lose up to 18% when running 'tigr' from the BIOBENCH benchmark. Therefore, there is almost 40% performance variation in the system depending on the static page policy that a memory controller employs. This is motivation enough for researchers to start thinking about developing dynamic page policies that switch between open- and close-page policy at run time based on the application access behaviour.

The *Static Profiling* presented in Table 4.1 shows the cost of page-hits and page-misses when the memory controller selects *the best* static page closure policy scheme for each workload by static profiling of memory accesses. The static profiling provides a baseline to evaluate the performance of dynamic page closure policies discussed in this chapter.

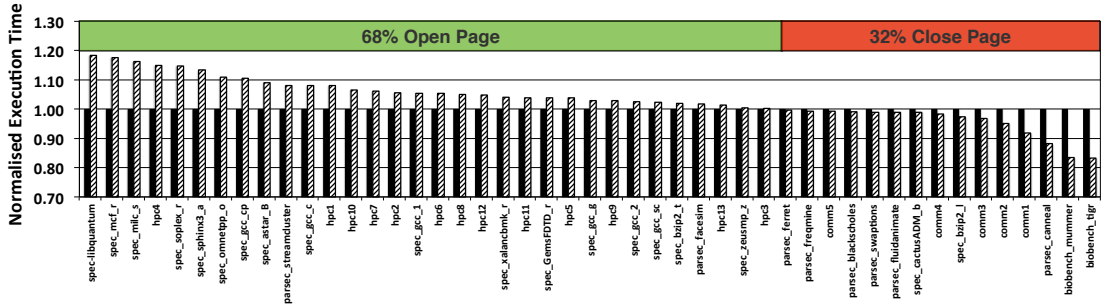


Figure 4.1: Performance of static page policies for all workloads.

Motivated by this observation, Hybrid page closure policies have emerged. This type of page policy uses various prediction algorithms to switch dynamically between the open-page and close-page policies according to the application access behaviour. Prediction accuracy and scalability are the two main challenges of designing such page policy predictors. In general, there is a linear relationship between the memory size and the resources required to monitor the memory access pattern of that memory space. Thus, as the memory size grows, the cost of the page closure policy predictor grows and, as a result, the memory controller complexity and area of the implementation increases. In modern computer architectures, the memory controller is integrated on the die as a part of the processor. Thus, increasing the size of the memory controller increases the die area of the processor which may not be economical.

Overall, considering the scalability of emerging memory systems such as HMCs, increasing interest in using large amounts of DRAM instead of disk storage in servers and database applications for instance using 64 TB of DRAMs in RAMCloud [Joh, OAE+10, ORS+11] and using 150 TB of DRAMs by Facebook in *memcached* [ORS+11] these trends demand a scalable approach for efficient design.

In the next section, HAPPY is introduced as an efficient technique to address the scalability problem of the page closure policy prediction technique for DRAM memory systems.

4.3 HAPPY:Hybrid Address-based Page PolicY

4.3.1 HAPPY: Basic Principles

HAPPY is a compact memory address-based encoding built on the observation that there is a strong correlation between physical address bit values and the internal structure of DRAMs. Understanding the basic operation of DRAMs shows that one of the first steps to access the DRAM structure is the address mapping process. During this process, the physical address bits provided by a core are translated to the corresponding channel, rank, bank, row and column of a DRAM device using a fixed and pre-defined address-mapping algorithm. Having a fixed translation mapping creates a strong correlation between physical address bits and the DRAM's structure. It means that, if some useful information can be extracted from the physical address bits after translation, it is possible to extract the same information before this stage.

All the page closure algorithms proposed so far focus on monitoring the memory access behaviour after the translation phase. In general, these techniques use different performance counters on a channel, bank or row basis to monitor page hits/conflicts, the time that a row could be kept open etc. HAPPY proposes a novel binary-encoding scheme with performance counters storing the page closure history directly from the physical address bits. In other words, HAPPY introduces one predictor per physical address bit to forecast the page closure policy of each row in the memory system according to the run-time memory accesses.

Sections 4.3.2 and 4.3.3 show how HAPPY can be applied to the two main page closure categories: access-based and time-based techniques. One traditional and one state-of-the-art technique are illustrated to demonstrate how the HAPPY encoding can be applied to different systems with different implementation characteristics.

4.3.2 HAPPY: Access-based Prediction

To demonstrate how HAPPY can be applied to access-based algorithms the traditional hybrid-page policy algorithm is selected. This employs simple, saturating counters to monitor the memory access pattern and dynamically switch between open- and close-page policy at run time. Figure 4.2 depicts the basic structure of such page closure policy predictors.

In this technique, one saturating counter (e.g. a 2-bit counter) is assigned to each row of a DRAM bank. Every time that a row-miss occurs the corresponding counter is

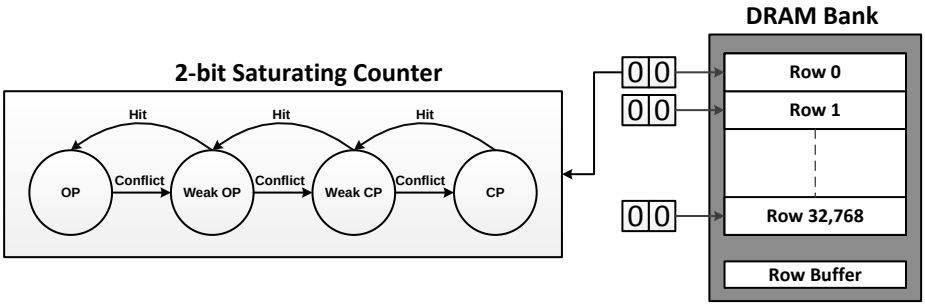


Figure 4.2: Basic structure of hybrid page policy.

incremented; Whenever a row-hit occurs the counter is decremented. For each memory request, the accessed row's counter value determines the page closure policy; if the value is 0 or 1 the open-page policy is predicted and if it is 2 or 3 the close-page policy is predicted. However, having a counter for each row in a DRAM device imposes a high area and power overhead to the memory system. For example, a 4 GB DRAM memory system with 1 channel, 2 ranks, 8 banks and 32,768 rows per bank, require 524,288 counters, which is not scalable (analysis presented in Section 4.5).

Figure 4.3 depicts the HAPPY implementation of a Hybrid page policy. The binary representation of the requested physical address is a pattern of zeros and ones. HAPPY dedicates two encoding counters per physical *address bit* location: one counter to monitor the position when its value is '1' and one to monitor it when it is '0'. Training of these counters is similar to the original implementation of Hybrid; that means for every page conflict the counter corresponding to each physical address bit is incremented and for every page hit the same counter will be decremented. Considering the page-hits and page-misses happen on a row basis then there is no need to monitor all the available physical address bits. Thus, the physical address bits corresponding to columns and cache-lines offset are not used. This reduces even further the implementation costs of HAPPY.

Having done this, each physical address bit correlates with the possibility of getting page hit/conflict depending on the value of that bit. Therefore, for a given physical address the likelihood of getting page hit/conflict can be calculated simply by considering all the participant bit's counter values in the requested address and using one of the following techniques: Majority vote or Aggregation.

Majority vote: Figure 4.4 explains this scheme using a simple example. Each physical address bit has a counter which has its own standalone vote to choose an open- or close-page policy for the requested physical address. The page closure policy

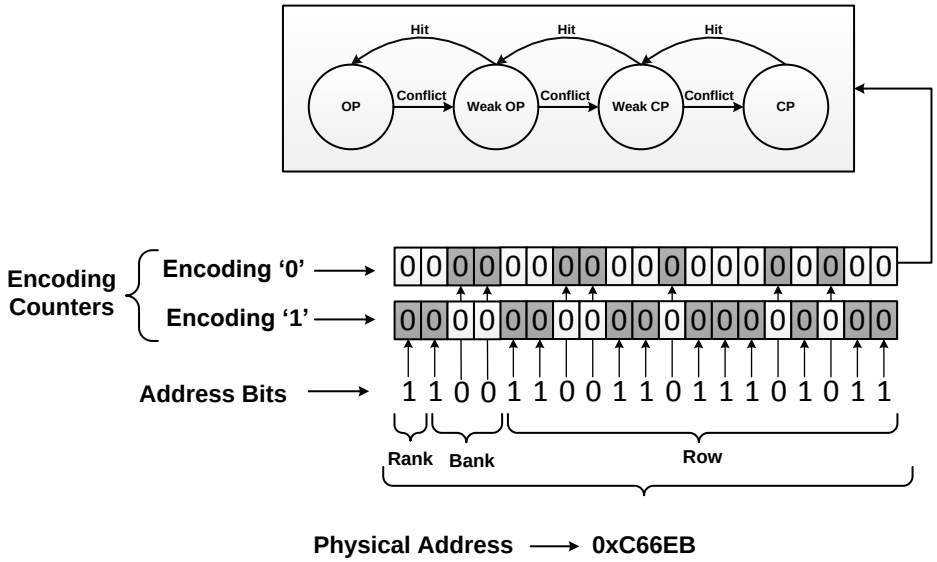


Figure 4.3: HAPPY implementation of hybrid page policy.

vote of each bit can be extracted by looking at the more significant bit of the saturating counters. If this bit is '0' there is an open-page policy vote and if the value is '1' there is a close-page policy vote. As the name of decision implies, the final vote is determined by the majority vote across all the counters.

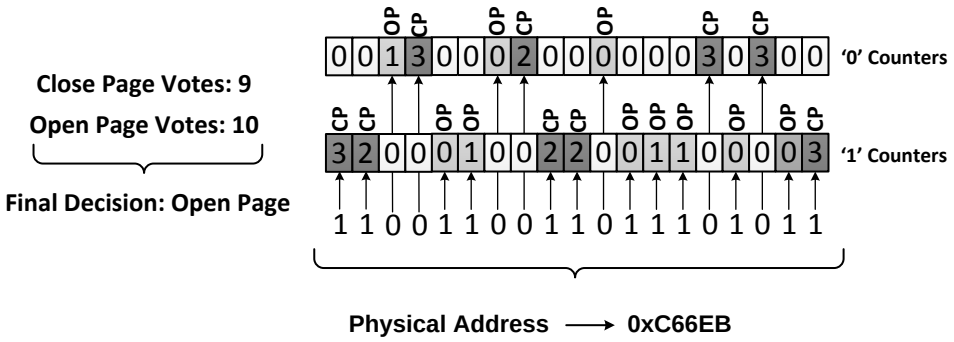


Figure 4.4: Example of HAPPY majority vote decision.

Aggregation: the final page closure policy decision can also be determined by comparing the aggregation of the all the counters' values, Equation (4.1), against a certain threshold, Equation (4.2).

$$\begin{aligned} &if \sum AddressBitCounters < Threshold \rightarrow Open\ Page \\ &else \qquad\qquad\qquad\qquad\qquad\quad \rightarrow Close\ Page \end{aligned} \tag{4.1}$$

$$Threshold = \frac{AddressBitsWidth \times CounterSaturatingValue}{2} \tag{4.2}$$

The experimental results show similar results using either majority or aggregation, thus only the majority vote decision scheme is used in the final experimental results.

4.3.3 HAPPY: Time-based Prediction

As a case study to show how HAPPY can be applied to a time-based prediction algorithm, the Intel-adaptive open-page policy [Raj] employed by the Intel Xeon X5650 [Dod06, Int] is chosen. The basic structure of such a page closure policy is presented in Figure 4.5.

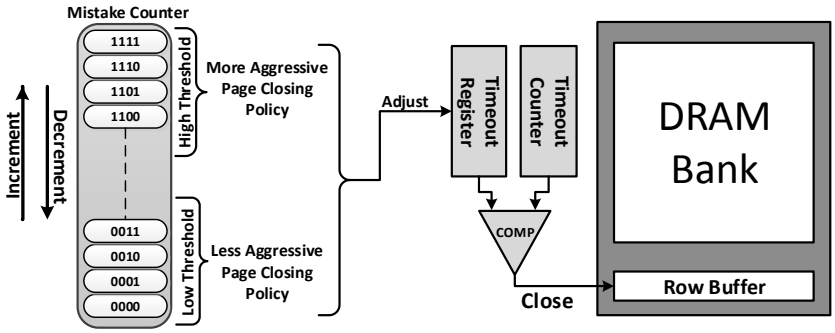


Figure 4.5: Basic structure of the Intel-adaptive page policy predictor.

The integrated memory controller used in this Intel processor can be configured at boot time to employ one of the following three different page closure policy schemes: close-page, fixed-open and Intel-adaptive open page. The *fixed-open page policy* keeps a row open for a fixed period of time and closes it after that. The Intel-adaptive scheme is an advanced version of the fixed-open schemes. Similar to the fixed-open policy, in this structure, each row buffer within a bank has a Timeout Counter (TC) and a Timeout Register (TR). A row will be kept open until TC reaches the TR and then closed. However, the initial TR might not be a suitable value for all the benchmarks. Thus, the Intel-adaptive scheme provides a technique to update the TR at run time using a 4-bit Mistake Counter (MC). Whenever a page conflict happens that could have been a page-empty, since there was enough time to precharge the last accessed row, then the MC is decremented. Whenever a page empty could have been a page-hit, since the row being accessed is the same as the last accessed row in that bank, then the MC is incremented. After a specific time interval the MC will be checked against a predefined low and high threshold to see if either a less or more aggressive page

closure policy is required. If the MC is higher than the high-threshold then the TR will be incremented to keep the accessed row open for a longer period and if the MC is lower than the low-threshold the TR will be decremented to close the accessed row sooner.

Figure 4.6 depicts the HAPPY implementation of the Intel-adaptive open page policy. This time the aim is to extract the timeout value for each row to be kept open from the physical address bits. The same methodology explained in the previous section is used to address the issue; the only difference is that instead of using simple saturating counters a monitor unit is dedicated to each physical address bit location. Each monitoring unit includes a MC and a TR with the same function as the original implementation of Intel-adaptive page policy. A global timeout counter is still required to keep track of row closing and opening times on a per bank basis.

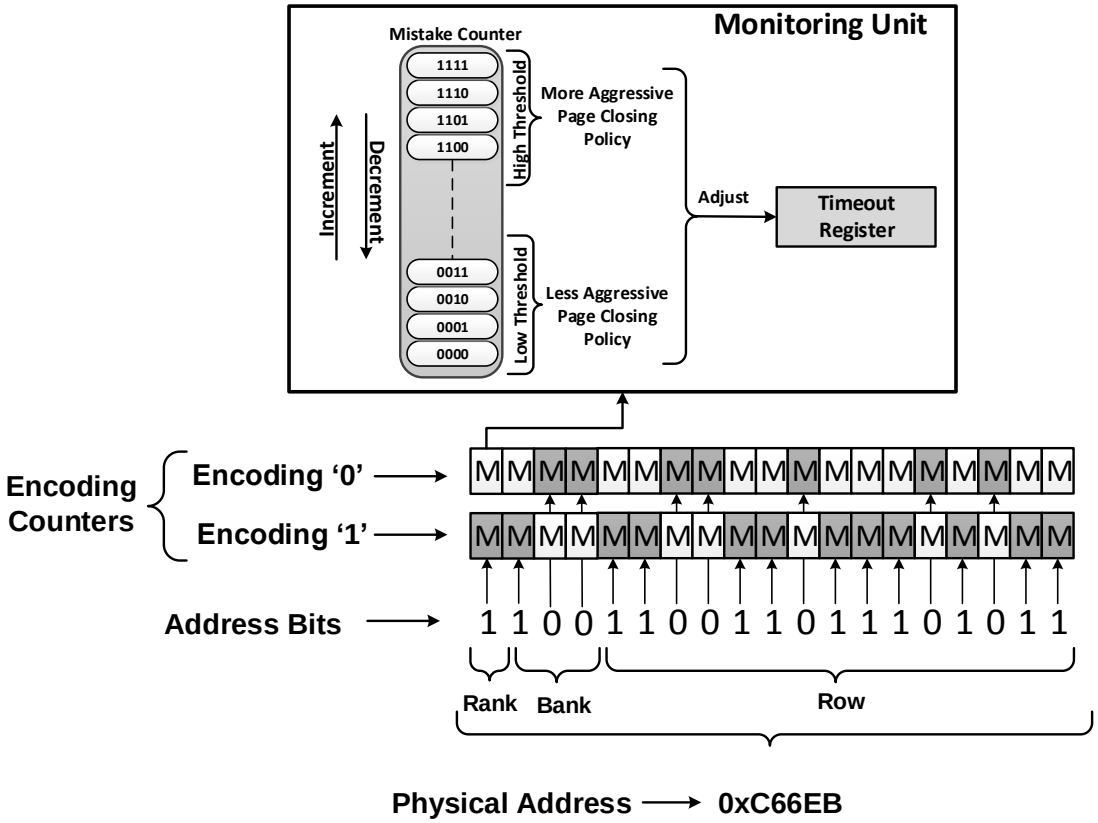


Figure 4.6: HAPPY implementation of Intel-adaptive page policy predictor.

Updating the MCs is as before but this time it is applied per physical address bit basis rather than per bank. Moreover, instead of having a global timeout register per bank, the time period that a row can be kept open will be calculated from the aggregation of all the participant bits for an accessed physical address.

To make a fair comparison between HAPPY and original implementation of this page policy, the size of the MC is chosen as before (e.g. 4-bit) and the size of TR at each physical bit location is chosen to be small enough that the maximum time that a row can be kept open is equal to having one global TR per bank.

4.3.4 HAPPY: Further Possible improvements

This chapter is mainly intended to explain the basic principles behind HAPPY and how it can significantly improve the scalability of the prediction algorithms. However, a significant reduction in the cost of implementation provides further opportunity to design a more efficient and accurate predictor.

For instance, the experimental results show that increasing the size of saturating counters from 2-bit to 4-bit for the first case study can improve the prediction accuracy and, as a result, the performance of memory system. However, for a 4 GB DRAM that requires around 500K saturating counters; two extra bits per counter impose a significant area and power overhead to the system which is impractical. Considering that HAPPY can reduce the cost of implementation by 13,000$\times$ (for a 4 GB DRAM) without harming the performance there is enough space to increase the counter size with negligible overhead.

4.3.5 HAPPY: Intuition

- **Observation:** The main intuition behind HAPPY is based on the observation that addresses that are spatially close together tend to have a similar page-closure policy preference. HAPPY is devised to exploit such behaviour by fine grain monitoring of physical address bits' access pattern.

- **Ensemble Methods:** Although HAPPY is devised based on a careful observation of the basic concepts and operations of DRAMs, there are Machine Learning principles that can justify the intuition behind HAPPY. A mathematical/theoretical framework that can explain HAPPY is that of Ensemble Methods [SW11]. The family of algorithms categorised as Ensemble methods combines multiple (normally simple) predictors. The theory explains how combining such predictors can obtain a much improved predictor, provided certain diversity properties among the predictors are met. Random forest, and neural networks are examples of very successful prediction algorithms part of the ensemble family. This

chapter addresses an online learning scenario and uses a fixed number of predictors with a non-linear combination function. When applying HAPPY to Intel-adaptive, a pair of simple predictors will be generated per physical address bit and a regression problem will be solved. Each pair of predictors is trained using a different single physical bit (different features) and each member of the pair is trained using only Zero or One occurrences (different dataset).

4.4 Evaluation Methodology

Investigating the page closure prediction algorithms for DRAMs requires a careful evaluation. These types of algorithm are highly sensitive to the application access pattern, therefore their evaluation using only a small set of workloads with the specific access pattern and for specific memory configuration cannot explore the real capabilities of each scheme. To address this, an intensive evaluation was carried out, described as follows:

Simulator: as described in Chapter 3, USIMM [CBS+12] was used as the main simulation platform for these experiments. USIMM was modified to support five different, existing page closure policies (i.e. Open-Page, Close-Page, Hybrid, Fixed-Open and Intel-adaptive open page) plus the two implementations of HAPPY which were discussed in section 4.3 (i.e. Hybrid-HAPPY and Intel-adaptive-HAPPY). A FR-FCFS scheduling algorithm was used in the experiments. HAPPY was evaluated based on different memory configurations, 2 GB for single-thread and 4 GB for multithread workloads. To increase the memory congestion USIMM is configured with 1 channel and 1 rank. The USIMM system configuration parameters that were used in the experiments in this chapter are captured in Table 4.2.

Address Mapping Schemes: The number of page conflicts in DRAMs, and as a result the memory performance, is susceptible to the memory address mapping scheme. The experiments consider three different address mappings presented in Figure 4.7. The first mapping (1) is a standard mapping to maximize row buffer locality. The next two address interleaving policies are schemes proposed by Kaseridis *et al*. [KSJ11] and Zhang *et al*. [ZZZ00]. The proposed mapping by Zhang *et al*. XORs part of the address bits with the bank address bits to produce a new bank index (see Figure 4.7b). Kaseridis *et al*. [KSJ11] extend this technique by producing the column index using a different section of physical address bits (Figure 4.7c). Both techniques aim to reduce page conflicts in DRAMs. The experiments show that the permutation-based page

Model	Description	Value
Processor	Clock Speed	3.2 GHz
	Pipeline depth	10
	ROB size	128
Memory System	Bus Speed	800 MHz
	Number of Channels	1-4
	Ranks per channel	1
	Bank per rank	8
	Row per bank	65,536
	Cache line per row	128
	Cache line size	64 Byte

Table 4.2: USIMM configuration parameters.

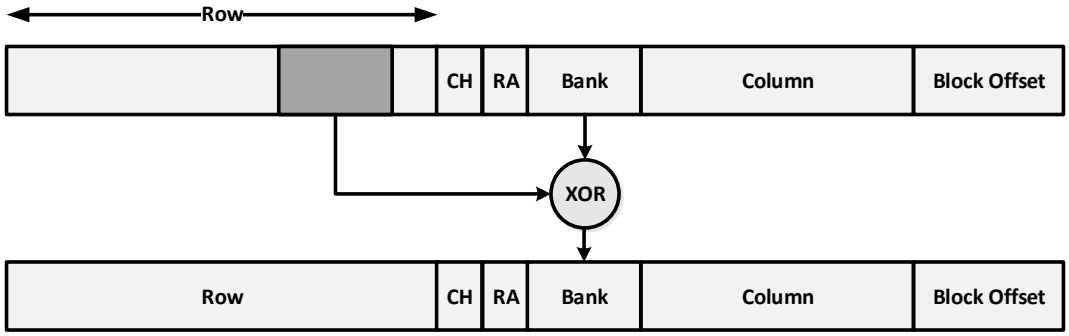
interleaving (address mapping 2) performs better for most of the workloads. Therefore, this address mapping scheme is employed in all the experiments. Focusing on the best page closure policy (i.e. Intel-Adaptive-HAPPY), the sensitivity of HAPPY's prediction accuracy to all the three address mappings schemes is investigated.

Workloads: As described in Chapter 3, the workloads include a wide range of memory intensive applications (i.e. 48 workloads) from different benchmark suites (PARSEC [BKSL08], SPEC [Dix91], BIOBENCH [AJW+05], HPC and COMMERCIAL) and representative regions of interest for each application. For a recap, Table 4.3 lists the workloads and their corresponding benchmark suites. A prefix is added to each application that will facilitate the naming of multi-thread workloads constructed from these applications later on.

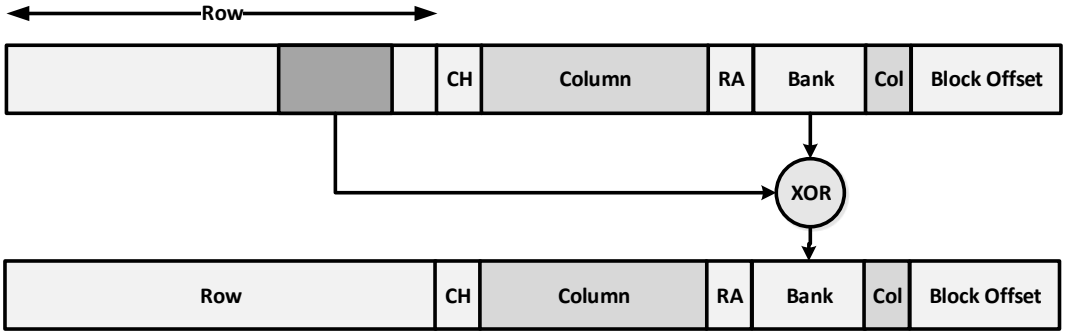
Benchmark Suites			
SPEC		**PARSEC**	**COMMERCIAL**
(a) GemsFDTD_r	(k) astar_B	(u) canneal	(D1) comm1
(b) bzip2_1	(l) bzip2_t	(v) streamcluster	(D2) comm2
(c) cactusADM_b	(m) gcc_1	(w) blackschols	(D3) comm3
(d) gcc_2	(n) gcc_c	(x) facesim	(D4) comm4
(e) gcc_cp	(o) gcc_g	(y) ferret	(D5) comm5
(f) gcc_sc	(p) mcf_r	(z) fluidanimate	**BIOBENCH**
(g) milc_s	(q) omnetpp_o	(A) freqmine	(E) mummer
(h) soplex_r	(r) sphinx3_a	(B) swaption	(F) tigr
(i) xalancbmk_r	(s) zeusmp_z	**HPC**	
(j) libquantum	(t) leslie	(C) hpc1 - hpc13	

Table 4.3: Evaluated workloads and benchmark suites.

(a) Mapping 1: Maximise row-buffer locality



(b) Mapping 2: Permutation-based Page Interleaving [ZZZ00]



(c) Mapping 3:Minimalist Open-Page Scheme [KSJ11]

Figure 4.7: Different address mappingg schemes.

The USIMM simulator can run arbitrary multi-application workloads using multiple traces. To increase the variety of memory access patterns, USIMM was set up for multi-applications to produce 22 random workload mixes; a combination of 4-thread, 8-thread and 16-thread applications. Table 4.4 listed these multithread workloads considering the prefix of single thread workloads presented in Table 4.4. Overall the experiments consider 70 workload mixes.

Memory Footprint (MF): To evaluate the performance of page closure predictors a careful study has to be carried out otherwise the performance and accuracy numbers might be misleading. For instance, if an application targets a very small portion of memory then it might be possible to predict its behaviour using a very small number of performance counter whereas if the application accesses all over of the memory space then it might be more difficult to keep track of the application access pattern with only a few counters (e.g. HAPPY). To have a fair evaluation methodology the memory

Multithread Workloads	
MIX1: (w-D3-D3-F)	MIX12: (b-n-s-w)
MIX2: (D1-D5-E-F)	MIX13: (w-D1-D5-x-y-z-t-F)
MIX3: (D1-x-y-F)	MIX14: (D1-D4-D5-x-z-x-B-F)
MIX4: (D2-D4-A-F)	MIX15: (D1-D4-D5-j-E-D5-v-F)
MIX5: (D2-D4-j-E)	MIX16: (D2-D4-D5-z-A-A-D4-F)
MIX6: (D2-y-t-F)	MIX17: (C5-C6-u-l-e-o-p-h)
MIX7: (D4-D5-g-F)	MIX18: (C13-C14C17-C18-C21-C2-C4-v-k-l-c-m-e-n-h-s)
MIX8: (D4-x-x-F)	MIX19: (C13-C18-C21-C2-C6-u-v-C21-u-l-l-o-t-p-h-s)
MIX9: (x-y-z-F)	MIX20: (C14-C17-C21-C22-C2-C4-C5-C8-C14-C21-C4-k-e-o-a-p)
MIX10: (C21-C22-C4-b)	MIX21: (C17-C21-u-C17-q-q-i-t-o-b-o-a-t-p-q-i)
MIX11: (C5-C6-u-e)	MIX22: (C18-C22-C5-C6-C8-u-k-l-d-e-n–o-p-q-h-r)

Table 4.4: Randomly generated multithread workloads.

traces should cover a wide range of access pattern. To this aim, the total number of physical pages accessed (Memory Footprint) was monitored at run time and the results show that the single thread applications have the average MF of 30% (up to 97%), the 4-thread workloads have the average MF of 50% (up to 75%), the 8-thread workloads have the average MF of 70% (up to 85%) and the 16-thread workloads have the average MF of 95% (up to 99.8%).

4.5 Results and Discussions

This section analyzes the different page closure policy prediction schemes compared with using HAPPY by looking at execution time, accuracy and scalability. Before jumping to the result graphs the following summary might be helpful:

- The HAPPY implementation of Hybrid page policy is called Hybrid-HAPPY for brevity.
- The HAPPY implementation of Intel-adaptive open-page policy is called Intel-adaptive-HAPPY for brevity.
- Figure 4.8 presents the prediction accuracy in terms of page hit and misses.

- The results in Figure 4.9 and Figure 4.10-4.13 are normalized to the 'Static Profiling'; the lower the bar the better performance.
- Overall, the HAPPY implementations perform similarly to (or better than) state-of-the-art policies while reducing significantly the hardware overheads for dynamic page closure policies. Note that while average geometric mean performance improvements are single digits (5%-8%), HAPPY requires a minimum 5× less storage overhead than earlier techniques. Unlike prior proposals, the hardware overhead of HAPPY is scalable with the DRAM memory size.
- Figures 4.14 and 4.15 present the prediction accuracy of the best predictors (Intel-adaptive and Intel-Adaptive-HAPPY) analysing the effect of different address mapping schemes. This can be observed that Intel-Adaptive-HAPPY using fewer hardware resources delivers a slightly better accuracy (i.e. 2%) than Intel-Adaptive for all three address mapping schemes.

4.5.1 Prediction Accuracy

Understanding the prediction accuracy for the different types of page closure predictors has its pitfalls. For instance, prediction accuracy in the case of Hybrid predictors is straightforward as the prediction outcome is either opening or closing a page (binary classification). However, in the case of the Intel-adaptive technique the accuracy needs to be described based on the timeout value (regression). Consider a scenario where a page has to be open for 40 clock cycles to get a page hit and Intel-adaptive predicts 39 clock cycles. In this case the prediction accuracy should be calculated differently. To have a fair evaluation across all the predictors with a different nature of prediction, we calculate the prediction accuracy based on the Page-Hit and the Page-Miss prediction outcome. The main purpose of using page policy predictors is to increase the page hits and reduce the page misses in the DRAM. Thus, the *Oracle page-hits* (maximum possible page-hits when having a perfect predictor) and *Oracle page-misses* (minimum possible page-misses when having a perfect predictor) were calculated. Then the actual page-hits and page-misses occurring during execution time of each workload were evaluated against the oracle numbers. Figure 4.8 presents the prediction accuracy (Geometric Mean (GMEAN)) of different predictors across all the workloads evaluated. The open-page and close-page policies deliver the maximum prediction accuracy for page-hits and page-misses, respectively. This happens because an open-page policy leaves all the pages open and then it can cover all the possible page-hits in the system

and none of the page misses. A close-page policy behaves similarly but in the opposite scenario. The hybrid-page policy delivers a moderate page-miss and page-hit prediction accuracy (around 60%). The Intel-adaptive and fixed-open both deliver a good prediction accuracy for both page-hits (80% and 75.8%) and page-misses (83.5% and 90.4%) respectively. Overall, the HAPPY implementation of both Intel-adaptive and hybrid are slightly more accurate than the original implementation. These prediction accuracy numbers justify the execution time presented in Figure 4.9. Also, from the accuracy results it can be concluded that the page-hit prediction accuracy has a higher impact on the overall execution time than the page-miss prediction accuracy.

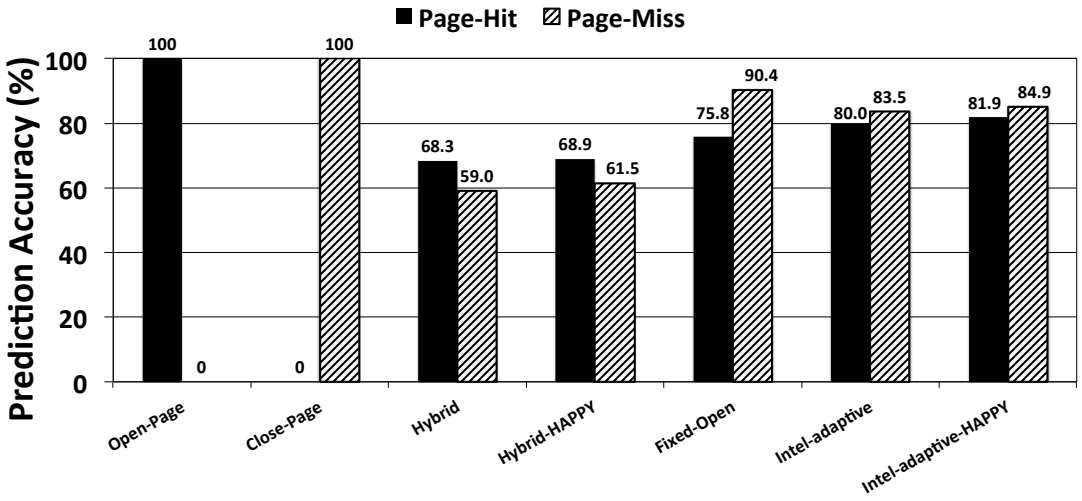


Figure 4.8: Prediction accuracy for different predictors.

4.5.2 Performance Analysis

Figure 4.9 summarises the performance of different prediction algorithms, normalised to static profiling, for all the benchmarks. Each bar graph in this figure represents the GMEAN of the execution time for the number of running workloads for each category. The detailed performance of the prediction algorithms for individual workloads is presented in Figures 4.10–4.12. These figures again confirm that a static page closure policy cannot deliver the optimum execution time for all the workloads. The corresponding workloads for HPC and SPEC benchmarks mostly prefer open-page policy. On the other hand, the corresponding workloads for PARSEC, BIOBENCH and COMMERCIAL workloads mostly prefer close-page policy.

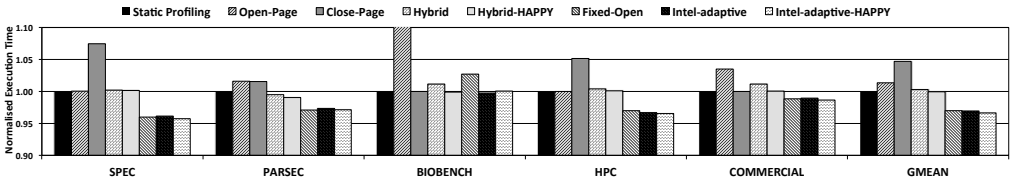


Figure 4.9: Average execution time normalised to static profiling for the single-thread workloads.

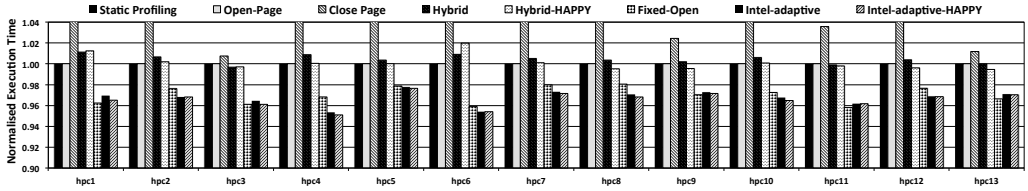


Figure 4.10: Execution time normalised to static profiling for HPC workloads.

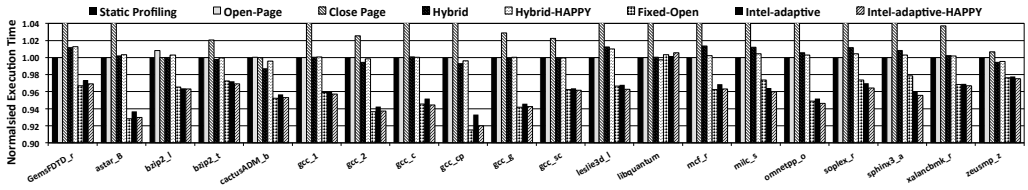


Figure 4.11: Execution time normalised to static profiling for SPEC workloads.

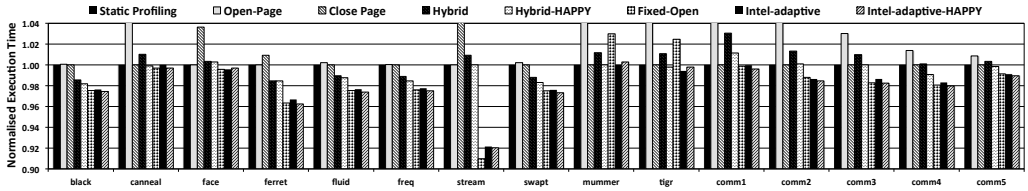


Figure 4.12: Execution time normalised to static profiling for PARSEC, BIOBENCH and Commercial workloads.

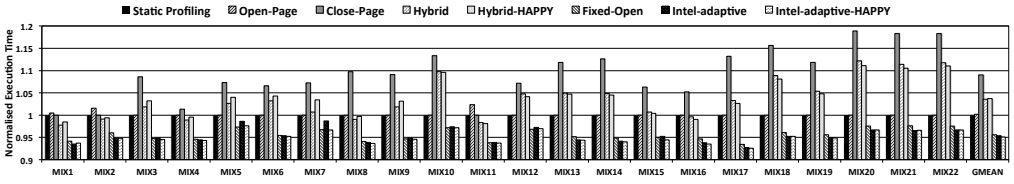


Figure 4.13: Execution time normalised to static profiling for multithread workloads.

Overview: The experimental results show that the best page closure prediction scheme (i.e. Intel-adaptive-HAPPY) delivers 5% and 8% better average performance across all the workloads (up to 12% and 20%) in comparison with open-page and close-page policy respectively. Overall, the HAPPY implementation of both Hybrid and Intel-adaptive achieved similar performance when compared with the original implementation of these page closure policies albeit with a much lower hardware overhead. Comparing the Intel-adaptive with the Intel-adaptive-HAPPY page policy shows that the HAPPY implementation can reduce the cost of implementation by 5× for the evaluated 64 GB memory size (up to 40× for a memory size of 512 GB) while maintaining the prediction accuracy. Similar behaviour can be observed for Hybrid and Hybrid-HAPPY. Hybrid-HAPPY shows 182,000× reduction in cost of implementation for the evaluated 64 GB memory size (up to 1.2M× for memory size of 512 GB) while maintaining the prediction accuracy.

Similarly, as Figure 4.13 presents, for multi-thread applications, even with a very high MF, HAPPY performance is consistent and delivers similar or slightly better performance than the original implementation. The experimental results show that Intel-adaptive-HAPPY delivers 5% and 14% better average performance across all the workloads (up to 9% and 22%) in comparison with open-page and close-page policy respectively.

Sensitivity to Address Mapping Schemes: To investigate the sensitivity of HAPPY to different address mappings the best page closure policy (i.e. Intel-adaptive-HAPPY) across all the predictors presented in this paper is selected and evaluated against the three address mappings presented in Figure 4.7. Figures 4.14 and 4.15 illustrate the prediction accuracy of Intel-adaptive (original and HAPPY implementation) using the different mapping schemes. These results show that the HAPPY implementation of Intel-adaptive always delivers identical or slightly better results than the original implementation no matter which address mapping is used.

4.5.3 Sensitivity to Memory Size

HAPPY was evaluated for up to 64 GB DRAM and the results shows that it has a consistent behaviour as the memory size increases. The experimental results suggest that the effective factor in HAPPY performance is the utilisation of memory address space rather than the size of memory. For this reason, a 4 GB memory organisation with up to 99.8% memory space utilisation is considered for the multithread experiments (results are presented in Figure 4.13). Even in this situation, the results show that HAPPY

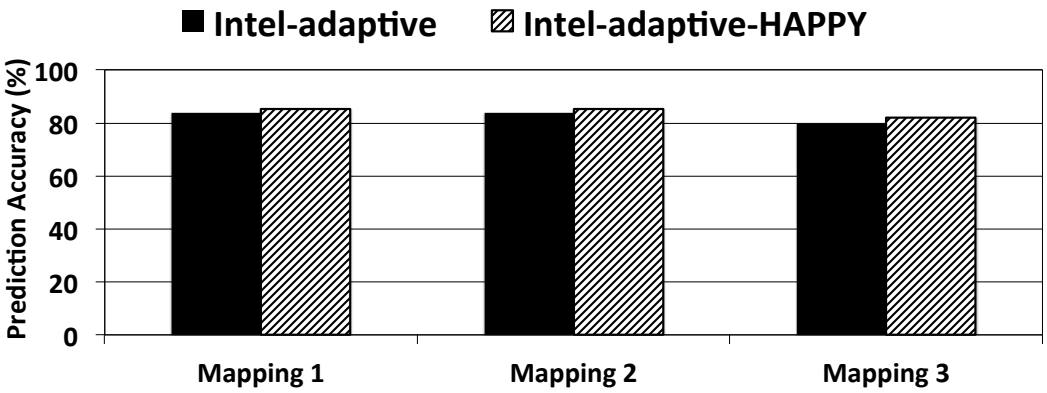


Figure 4.14: Page-hit prediction accuracy with different address mappings.

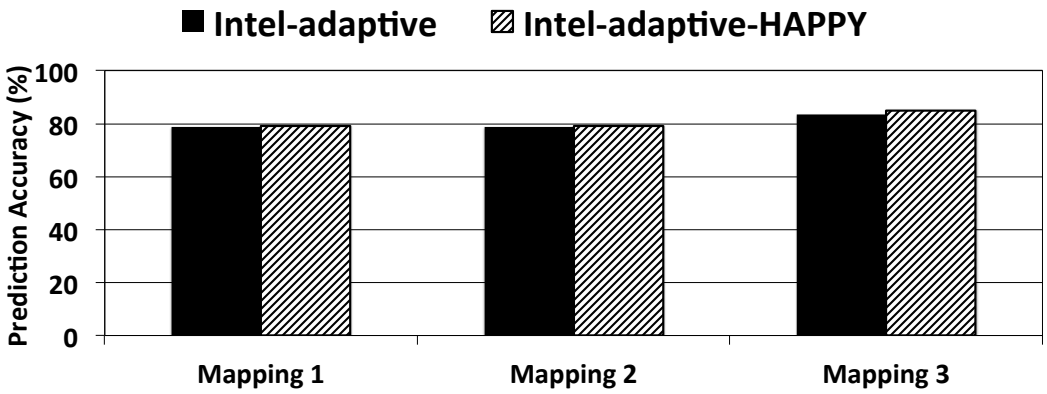


Figure 4.15: Page-miss prediction accuracy with different address mappings.

delivers a competitive performance against the original implementation of both Hybrid and Intel-adaptive page policies while reducing the hardware overhead significantly.

4.5.4 Scalability with Memory Size

Figure 4.16 depicts the required storage (bytes) for each prediction algorithm for different sizes of memory. The HAPPY implementation of the hybrid prediction technique is orders of magnitude (e.g. up to 1.2M×) cheaper than the original implementation while it delivers similar performance to original implementation. In the case of Intel-adaptive page closure policy, the HAPPY implementation requires slightly more resources than the original implementation for memory sizes of less than 8 GB. However, as the memory size grows, the Intel-adaptive-HAPPY uses less area than the original implementation up to 40× for a memory size of 512 GB. Table 4.5 depicts the required performance counters for different page closure policies with and without

HAPPY considering a memory system with X channels, Y ranks, Z banks and W rows.

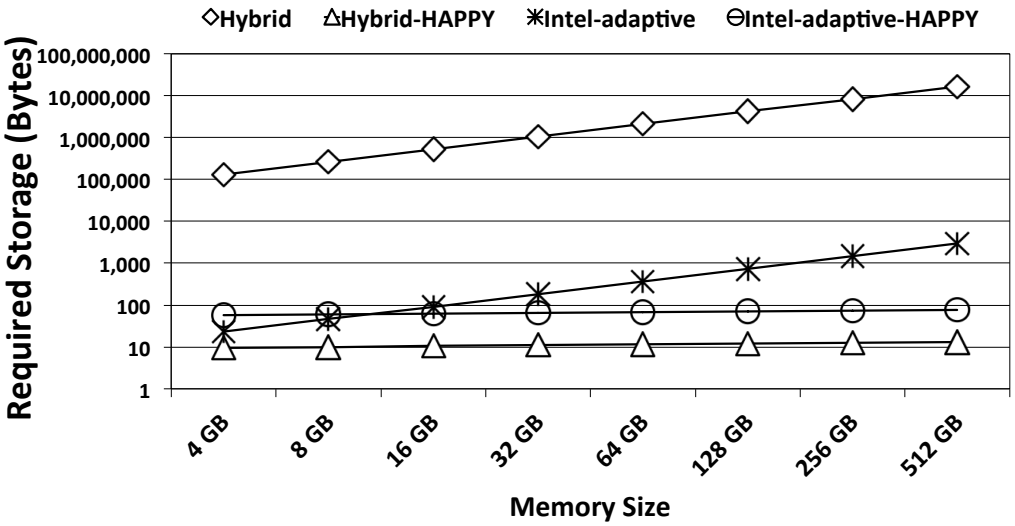


Figure 4.16: Scalability of different page closure prediction algorithms.

Implementation	Required Counters
Hybrid	$X \times Y \times Z \times W$
Hybrid-HAPPY	$(\log_2 X + \log_2 Y + \log_2 Z + \log_2 W) \times 2$
Intel-adaptive	$(X \times Y \times Z) \times 2$
Intel-HAPPY	$(\log_2 X + \log_2 Y + \log_2 Z + \log_2 W) \times 4$

Table 4.5: Required performance counters for different page closure policies.

4.5.5 Prediction Algorithms - Weakness and Strength

Due to the nature of implementation and structure of each predictor, each might or might not work in a specific situation. Here, such situations are discussed.

Static Policies: the open-page policy works best for high locality workloads but degrades the performance of DRAMs significantly for workloads with highly random or dynamic memory accesses. The close-page policy has the completely opposite behaviour. PARSEC and SPEC workloads are good examples which show the different behaviour of open-page and close-page policies (see Figure 4.11 and Figure 4.12).

Fixed-Open: The performance of this type of algorithm is fairly susceptible to its predefined timeout value. Similarly to the methodology presented in [KSJ11], this value is selected to be equal to t_{RC}, that is the minimum time limitation between consecutive accesses to different rows within a bank, in the experiments. This time delay

provides enough opportunity to capture a possible page hit; it does, according to the results presented in Figure 4.9. However, for non-memory intensive threads with high locality or memory intensive with low locality (e.g. 'mummer' and 'tigr') this technique might not work well. The reason is that, for the first category, the time interval between memory requests might be higher than the fixed timeout value which means this technique will close the row before a page hit happens. Similarly, for the second category the time interval between memory requests might be lower than the fixed timeout value, which means that a row would be kept open for an unnecessary time which, most likely, would lead to other page conflicts.

Hybrid: the integrated saturating counters employed in this category (either the original or HAPPY implementation) are trained by the number of page-hits and page-conflicts that they face. Therefore, the prediction accuracy of these types of techniques is fairly sensitive to the distribution of page hits/conflicts within DRAMs. For instance, 'streamcluster' presents such a behaviour.

Intel-adaptive: our experiments show that this prediction algorithm is the best across all the presented schemes in this paper. However, one weakness of this technique is the updating granularity of TR. In our experiments, every time that checking of the MC suggests a more or less aggressive page closure policy, TR is incremented or decremented by one respectively. Updating granularity by one step (increment or decrement) delivers a fine tuning of the TR but reduces the training rate of the overall prediction technique. This means that, for workloads where the application access pattern behaviour changes frequently (e.g. between high and low locality accesses pattern) within different time phases, the Intel-adaptive scheme might not be able deliver its best performance. Similar behaviour can be observed in 'canneal' or 'comm1'.

HAPPY: so far only the advantages of HAPPY are explained. However, like all the other proposed techniques, HAPPY also has weaknesses. Considering the global nature of a HAPPY implementation it is expected that HAPPY cannot perform as efficiently as fine grain schemes for workloads with fairly dynamic behaviour targeting a small part of DRAMs locally. This can be seen in workloads like 'tigr'.

4.5.6 Flexibility

HAPPY is the proof of the concept that the physical address bits can be the source of useful information that can be extracted using the right encoding and decoding techniques. This makes HAPPY a fairly flexible tool that can be applied to different prediction algorithms that have not been practical due to the cost of implementation, making

them feasible. In this thesis HAPPY was applied to two completely different prediction schemes and showed how the performance and scalability of these schemes improved.

4.6 Related Work

Succinctly, prior research in this area can be categorized in two main groups: access-based and time-based techniques.

Access-based techniques are those that monitor and keep a history of the row hits and row misses at different granularities in DRAMs enabling them to make a prediction of the future page closure policy for each row or bank within a DRAM memory system. Xu *et al*. [XAD09] proposed a two-level dynamic SDRAM policy predictor which collects the row hit/miss behaviour for the last *n* accesses in a history register. For each entry in the history register, there is a 2-bit saturating counter that keeps track of the page closure policy for each access. Huan *et al*. [HLHL06] proposed the Processor-Directed dynamic page policy where the processor keeps track of the last row access to each bank to predict page hits or misses for future memory requests. The processor sends this information to the memory controller to specify the page closure for the next memory access. Awashti *et al*. [ANBD11] keep track in a history table of the number of accesses each row has before closing it. When a row is open the number of expected accesses to that row is looked up, if there is no recorded entry for the accessed row in the history table that row is kept open. However if there is an entry for the accessed row it will be closed after the expected number of accesses suggested by the history table. More techniques using access-based page closure prediction can be found in [PP03, MC07, MAW01, SM04, SM05a, SM05b, Sch97].

Time-based techniques mainly focus on predicting the optimum time that a row can be left open. Blackmore [Bla13] presented a quantitative analysis of page closure predictors. This work specifically focused on the Intel-adaptive page policy structure and tried to improve it by introducing the inter-arrival distribution concept. Stonkovic *et al*. [SM05a] used the concept of *live-time* and *dead-time* to predict the page closure. The live-time is the time interval between opening a row until the last access to that row while dead-time is the interval from the last access to an open row until its closing. If the predictor predicts a zero live-time or if it predicts that the row has entered its dead-time period, then the row will be closed immediately after the DRAM access otherwise it will be kept open for future accesses. In another work, Kaseridis *et al*. [KSJ11] used the concept that in DRAMs there is a minimum time limitation of t_{RC} between two

activations within a bank and speculatively leave the pages open for the t_{RC} period.

To sum up, HAPPY is the only technique which considers an encoding based on the memory address bits offering a compact means of storing history to inform the predictions. In addition, it has been shown how to apply HAPPY to time-based (i.e. Intel-adaptive HAPPY) and access-based techniques (i.e. Hybrid HAPPY).

4.7 Summary

DRAM performance depends on the memory access pattern and, more specifically, the number of page-hit and page-conflicts that occur at run time. Modern DRAM controllers employ advanced page closure policy predictors to improve performance by trying to transform page-conflicts into page-empty (e.g. by closing the last accessed row at the "right time"), and page-empty cases into page-hits (e.g. by keeping open the last accessed row for longer time). However the main challenge is to balance the prediction accuracy of these predictors with manageable hardware overheads (scalability) as we increase the size of DRAM.

In this chapter, HAPPY – a compact and efficient binary-encoding technique – was described to alleviate the scalability problem of DRAM page closure predictors. HAPPY relies on the simple observation that there is a strong correlation between the physical address bits of memory addresses requested by processors and the internal structure of the DRAM as there is a fixed-address mapping scheme. Thus, the physical address bits carry the information that a memory controller needs to predict the page-hit or page-conflict for a particular access. Considering this, the required performance counters and monitoring units needed by the page closure prediction algorithms can be encoded from the physical address bits. Doubling the size of DRAM only implies one extra physical address bit. This means that with HAPPY only two extra monitoring units are required to predict the DRAM page closure policy when the size of memory is doubled. In other words, HAPPY offers the lowest hardware overhead to implement dynamic DRAM page closure predictor algorithms.

HAPPY was evaluated by applying it to a traditional Hybrid page closure policy, as well as the state-of-the-art Intel-adaptive open page policy included in Intel Xeon X5650. The experimental results show that the HAPPY implementation of Intel-adaptive page policy can reduce the cost of implementation by 5$\times$ for the evaluated 64 GB memory size (up to 40$\times$ for a memory size of 512 GB) while maintaining the

prediction accuracy. The other case study shows 182,000× reduction in cost of implementation for the evaluated 64 GB memory size (up to 1.2M× for memory size of 512 GB) while maintaining the prediction accuracy. The experiments have also reported the accuracy of the predictors and have studied the sensitivity towards the memory address-mapping. In both scenarios, HAPPY maintains its key advantage of offering no degradation of prediction accuracy while reducing significantly the hardware overhead.

Chapter 5

DReAM: Dynamic Re-arrangement of Address Mapping

5.1 Introduction

Increasing the number of general purpose cores and accelerator cores (e.g. GPU cores) integrated into a single chip and competing for access to DRAM, demands better performance from the main memory. In this situation, exploiting the maximum performance obtainable from the memory system is crucial. However, due to the internal structure and organisation of DRAMs, described in Chapter 2, there is always some memory bandwidth (Performance) wasted due to internal conflicts. One of the most serious conflicts in a DRAM memory system is referred to as a page-conflict (or row-conflict). This happens when two consecutive memory requests go to different rows within the same bank. In this situation, these memory requests must be serviced one after another which causes a high access latency for the second request. Dealing with page conflicts becomes even more challenging considering the fact that they are completely dependent on the memory access pattern. This means that the rate of page conflicts and the time of their occurrence change dynamically according to the application behaviour.

To mitigate the vulnerability of DRAM performance to page conflicts, state-of-the-art memory controllers have evolved into complex hardware components employing subsystems such as schedulers. These schedulers take advantage of workload runtime information (the sequence of memory requests) to reduce page conflicts. An important role of the scheduler is to minimise DRAM page conflicts (or row conflicts) by reordering the memory commands that are available to issue to the DRAM. However, the main

limitation for schedulers is the number of options (memory requests) that they have to choose from at the time of scheduling. In general, the number of available memory requests at the time of scheduling is limited by data dependencies between memory requests, the number of running threads, the number of cores etc. The freedom available significantly affects the scheduler performance. Therefore, there are conflicts that schedulers cannot eliminate. These page conflicts result from the address-mapping and data placement in DRAMs. As discussed in the next section, the address mapping is a process that maps the physical address bits provided by processors to the internal structure of DRAMs. This process controls the initial data placement in DRAMs. Thus, it is important to understand how to select a good address-mapping scheme to place and distribute data in DRAM devices to mitigate page conflicts. This is possible using a software-only approach; e.g. with OS support and intelligent memory allocators. However, this option faces complex problems when considering multiple independent applications executing concurrently, or with virtualised scenarios (both hypervisors and containers) and relies on software being compiled for specific memory hardware.

This chapter presents DReAM (Dynamic Re-arrangement of Address Mapping), a novel hardware technique based on approximating the entropy of each memory address bit for a set of memory requests, to generated workload specific address-mappings at run-time. To re-arrange the address mapping at run-time DReAM needs to support the online-data migration imposed by changing the address-mapping scheme. DReAM proposes four hardware-based solutions for data-migration inside DRAM with different levels of complication. The proposed solutions were evaluated over a wide range of mapping-sensitive and mapping-insensitive workload mixes. Three different address mapping schemes were evaluated over all the workloads and the best one was chosen to compare against DReAM. Overall, DReAM is complementary with existing schedulers in memory controllers and is the first on-the-fly mechanism capable of generating workload specific address-mappings without requiring to stop the running applications.

5.2 Background on DRAM Address Mapping

To recap what was discussed in Chapter 2, a brief background of DRAMs will be reviewed here with the focus on the address mapping process. Figure 5.1 presents the

basic organisation of a DRAM device. Each DRAM device consists of multiple banks each of which has a data array and one row buffer. In practice, the data array within a bank consists of multiple subarrays, each of which has its own local row buffer. The local row buffers within a bank are connected to other local row buffers as well as the global row buffer. There are some interesting works by Chang *et al.* [CLC+14], Kim *et al.* [KSL+12] and Seshadri *et al.* [SKF+13] to exploit these subarrays to improve the DRAM performance and bulk data copy in DRAMs.

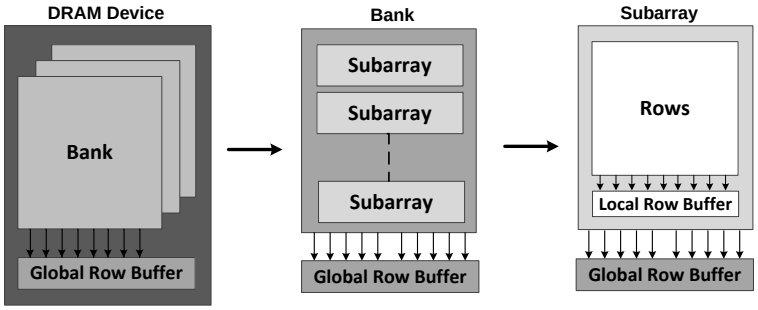


Figure 5.1: DRAM device organisation.

The address mapping mechanism for DRAMs transforms the flat 1D of physical addresses into the internal 2D structure of DRAMs devices (row & column). Figure 5.2 illustrates how one physical address can be interpreted with two different mapping schemes. Most memory systems contain DIMMs and a DIMM can have multiple *ranks* of DRAMs. Multiple DIMMs can be placed on a *channel*; i.e. the physical connection between a memory controller and DRAMs [JNW10]. The reason for these many hierarchical levels is to maximise the parallelism that can be exploited when servicing multiple memory requests.

In general an address-mapping scheme extracts the corresponding address for Channel, Rank, Bank, Row and Column from the physical address. Due to the internal structure and electronic circuit characterisation of the DRAMs, consecutive access to different memory locations can have a different memory cost depending on the previous state of the memory. For instance, if there are two consecutive accesses to the same row in the same bank of a DRAM, the second access can have significantly smaller latency than the first access since the target row has been 'opened' by the first memory request. On the other hand, if there are two consecutive accesses to different rows within the same bank, the second access has significantly higher latency in comparison with the first access. The reason is that, in this case, the previous row must be 'closed' before the new row is 'activated'. These scenarios describe a *page-conflict*

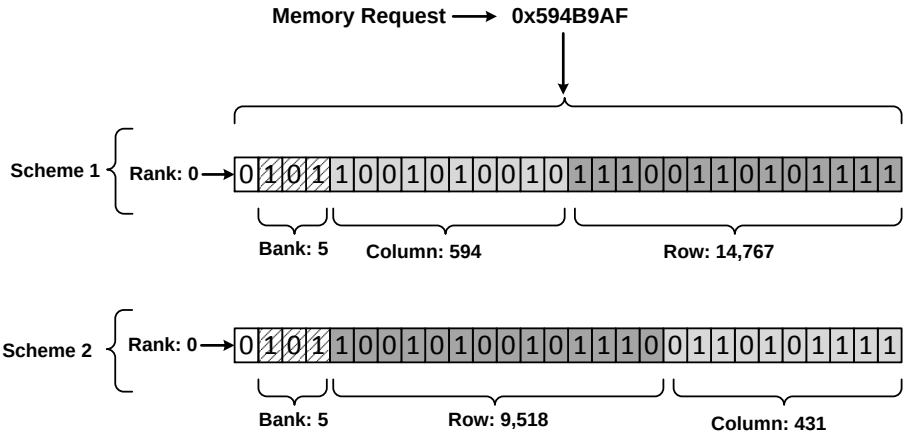


Figure 5.2: Two different address mapping schemes.

and degrades the overall performance of DRAMs.

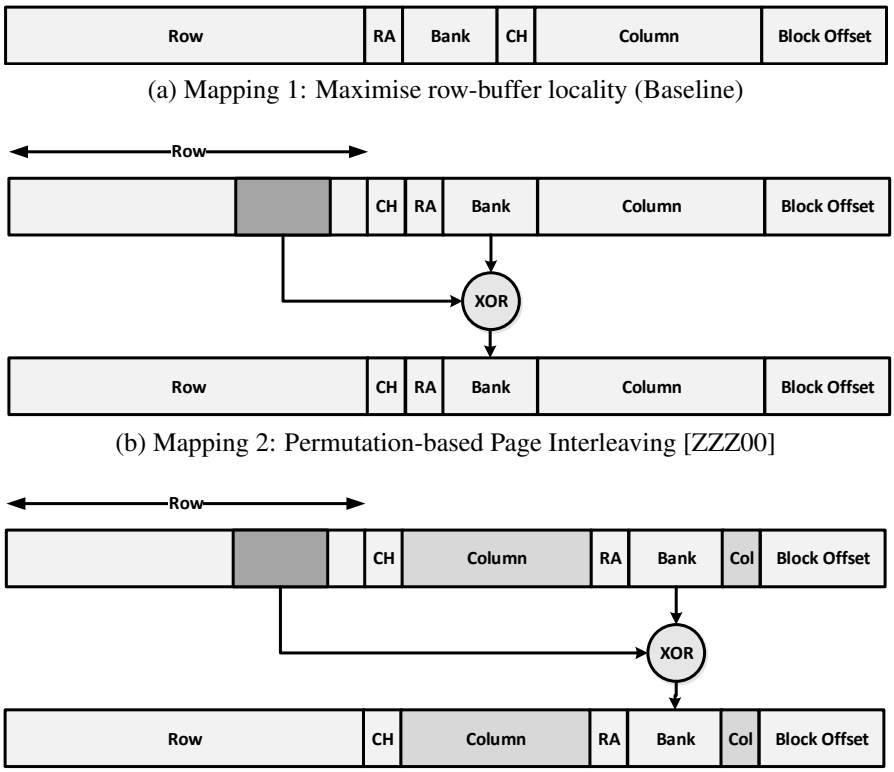
Page-conflicts are sensitive to the data placement in DRAMs and data placement is determined by the address-mapping schemes in the first place. Therefore, choosing an address mapping scheme carefully can reduce the page-conflicts and improve the performance of DRAMs.

5.2.1 Motivation - Address Mapping Analysis

Figure 5.3 presents three different well-known address-mapping schemes currently employed by modern DRAM controllers. The first mapping (Figure 5.3a) is a standard mapping intended to exploit the spacial locality by placing the column address at the bottom. The next two address interleaving policies are schemes proposed by Kaseridis *et al*. [KSJ11] and Zhang *et al*. [ZZZ00]. The proposed mapping by Zhang *et al*. XORs some of the row address bits with the bank's address bits to produce a new bank index (Figure 5.3b). This tries to change the bank ID whenever the Row ID is changed to reduce the page-conflict. Kaseridis *et al*. [KSJ11] extend this technique by producing the column index using a different section of the physical address (Figure 5.3c). Both techniques aim to reduce page conflicts in DRAMs.

There might be other variations of address-mapping schemes, in addition to those presented in this figure, that can be used to perform the required translation phase to service a memory request. However, the important point to consider is that current memory controllers can only use one such address-mapping scheme to translate the physical address to the internal structure of DRAMs. Moreover, modern DRAM controllers are limited to perform read/write operations in bursts (typically bursts of 4 or 8

items). This implies that some bits are used as a block offset, as shown in Figure 5.3.



(a) Mapping 1: Maximise row-buffer locality (Baseline)

(b) Mapping 2: Permutation-based Page Interleaving [ZZZ00]

(c) Mapping 3:Minimalist Open-Page Scheme [KSJ11]

Figure 5.3: Different address mapping schemes.

To motivate the technique presented in this chapter, Figures 5.4 to 5.8 present a performance comparison of different address-mapping schemes for all the benchmarks evaluated in this chapter. Each bar in these graphs represents the execution time normalised to the baseline address-mapping scheme (address mapping 1 in Figure 5.3). These experimental results suggest that a pre-defined address mapping scheme is not efficient in all situations and thus employing a fixed address mapping scheme cannot deliver the best execution time across all workloads.

According to Figure 5.4, the Permutation address mapping scheme almost always (except for the BIOBENCH benchmark) delivers a better average (GMEAN) execution time compared with the two other address mapping schemes. This address mapping is chosen as the best baseline of those presented in this chapter to be compared against DReAM address mapping.

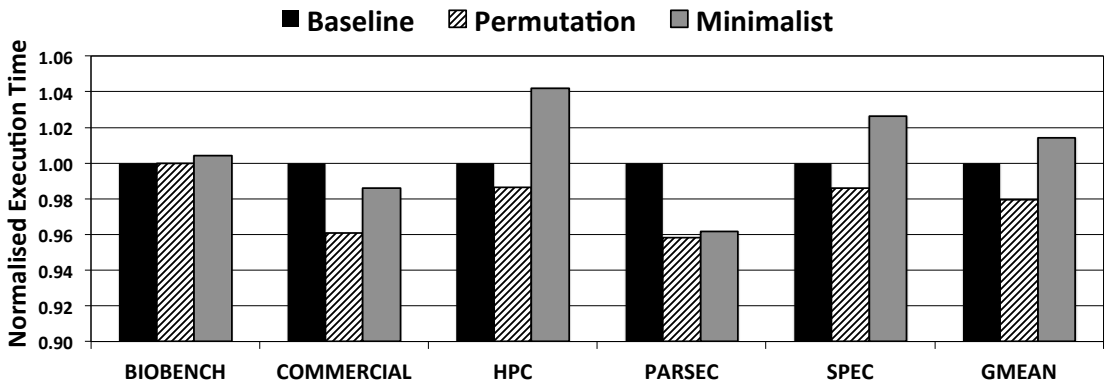


Figure 5.4: Performance comparison of different address-mapping schemes.

5.3 DReAM: Dynamic Re-arrangement of Address Mapping

DReAM is a novel technique to analyse the memory access pattern (produced either by single or multi-thread applications) at run-time and estimate an efficient address-mapping scheme, that reduces page-conflicts and improves page-hits. DReAM consists of two main phases: 'online prediction of address mapping' and 'on-the-fly data migration' that will be discussed in the following sections.

5.3.1 DReAM - Online Prediction of Address Mapping

The first step is to discover whether the current workload, a set of executing applications, is a good match with the baseline address mapping scheme. To investigate the efficiency of a baseline mapping it is useful to recap the address mapping process from a slightly different angle. As described in the previous section, a baseline address-mapping scheme decides which physical address bits should be used to address which specific part of a DRAM device (e.g. rank, bank, row etc.). Therefore, a physical address is divided into different sets of bits each set pointing to a specific part of the internal hierarchy of the DRAM system.

Considering consecutive requests to a DRAM module, the changing rate of each physical address bit (as a result of the changing rate of each bit within different sets) in comparison with the previous access has a strong correlation with the changing rate of a specific DRAM location that has been accessed. On the other hand, accessing different rows within the same bank causes page-conflicts and imposes a power and

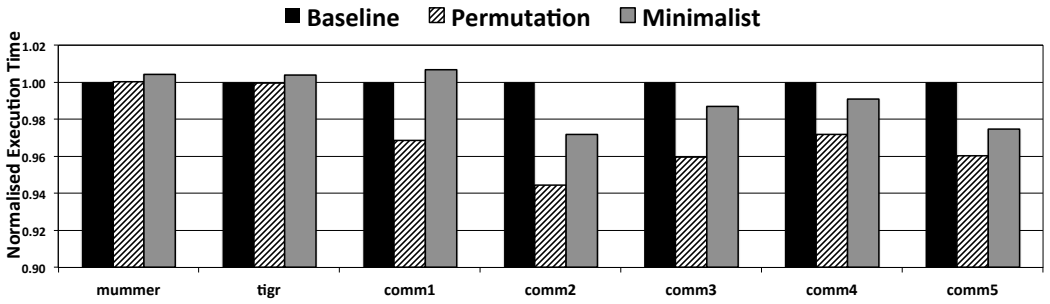


Figure 5.5: Address mapping profiling for BIOBENCH and COMMERCIAL benchmark suites.

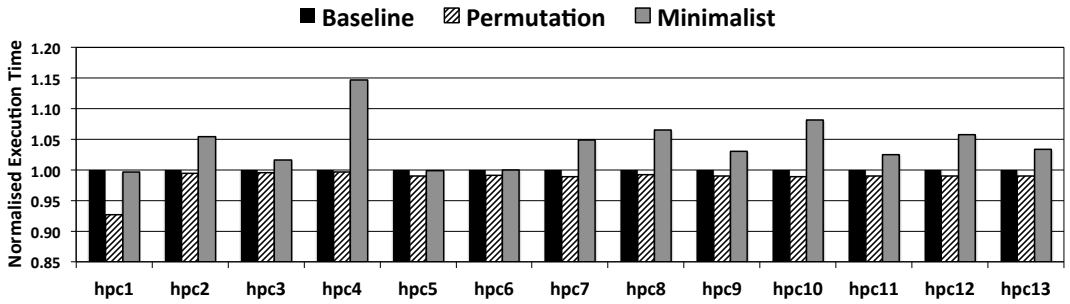


Figure 5.6: Address mapping profiling for HPC benchmarks.

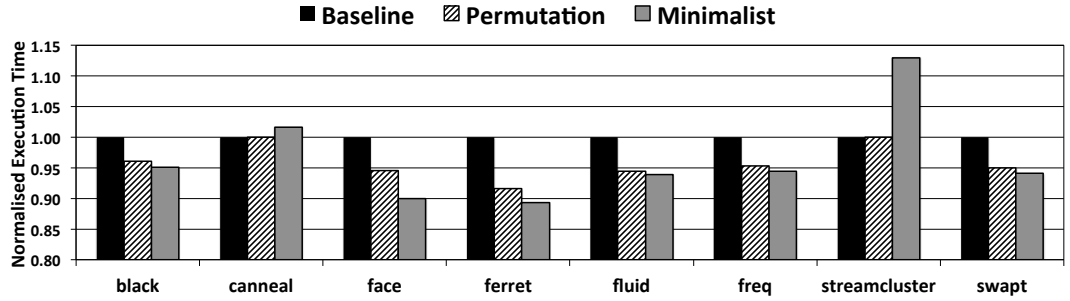


Figure 5.7: Address mapping profiling for PARSEC benchmark suite.

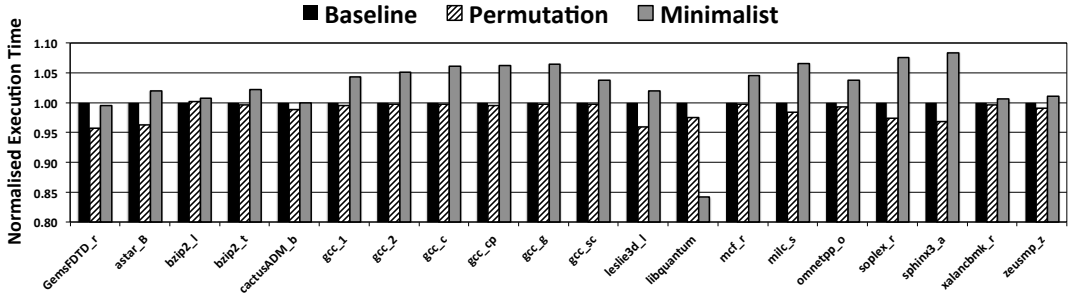


Figure 5.8: Address mapping profiling for SPEC benchmark suite.

performance overhead. Therefore, ideally, it is desired to keep the change rate of the physical address bits that are used to address the row, as low as possible to reduce the

row switches within a bank,

DReAM estimates how much each physical address bit changes by observing memory requests over a period of time as a means of generating improved memory mappings. The estimations of change per bit require minimum extra hardware; one counter per physical address bit per memory controller. Those bits changing the most have higher entropy and those bits changing the least have smaller entropy. Understanding the approximated entropy is enough to generate a new improved workload-specific address mappings.

For a given period, these counters (or frequency change estimators) keep track of the number of changes of each bit of the physical address in comparison with the previous memory address request. The given period creates time windows and can be based on number of clock cycles or number of memory requests. Figure 5.9 shows an example of five consecutive accesses (physical address) to demonstrate the function of these counters. The counter value of the two highlighted bits shows that bit 16 and bit 27 have been changed once and 4 times, respectively, in the last five memory requests.

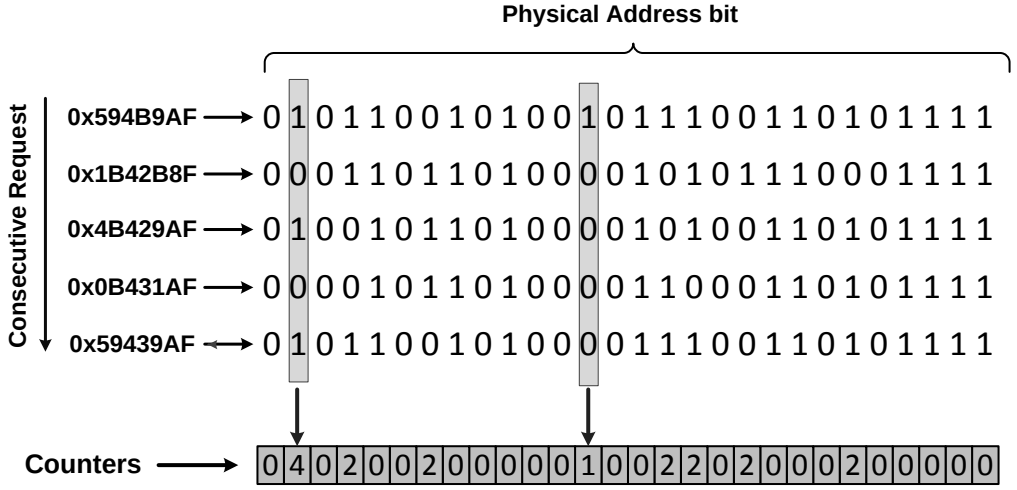
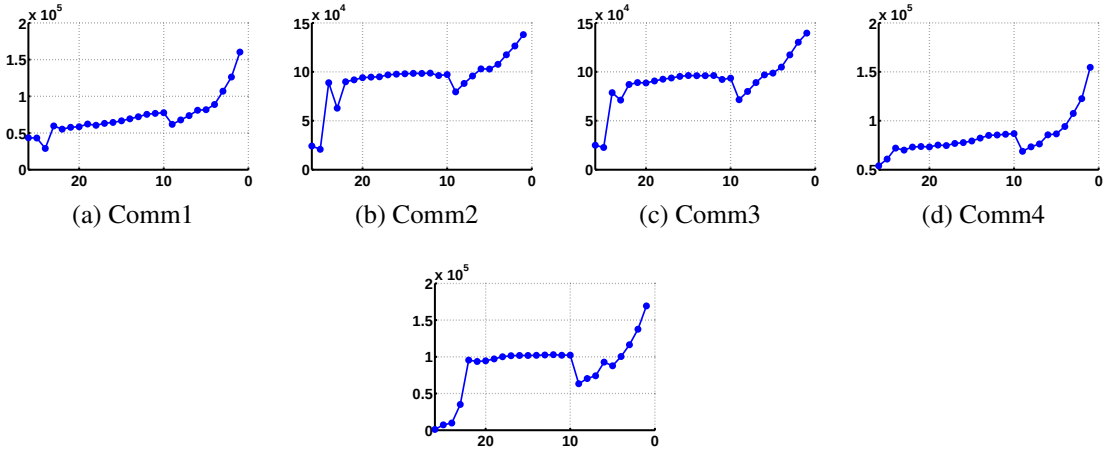


Figure 5.9: Bit-counters mechanism.

These counters generate a pattern (or signature) that is representative of the current memory access behaviour as perceived by the memory controller. Figures 5.10 to 5.14 show such a signature extracted from these counters for all the benchmarks evaluated in this chapter. The X-axis in each plot represents the corresponding counter ID per physical address bits and Y-axis shows the overall bit change rate over the application execution time.

There is an exponential growth in the rightmost 5 bits of almost all the patterns.

This is due to spacial locality that implies accessing the sequential physical addresses. Looking at these pattern and the address-mapping schemes presented in Figure 5.3 justifies why the column address bits are typically placed in the bottom of the physical address space. In this way, the memory requests accessing consecutive cache lines will be mapped to the consecutive columns within the same row (i.e. Page-Hit). There is another interesting behaviour that can be observed from these patterns. For instance, Figure 5.12c and Figure 5.13m show an overshoot (i.e. high change rate) for bit 14. Considering the address-mapping scheme presented in Figure 5.3 bit 14 will be mapped to the row address space. Thus, the high change rate of this bit increases the possibility of changing row within the same bank (i.e. Page-Miss). This suggests that most likely these two workloads (*face* and *libquantum*) suffer from a high rate of page-conflict.



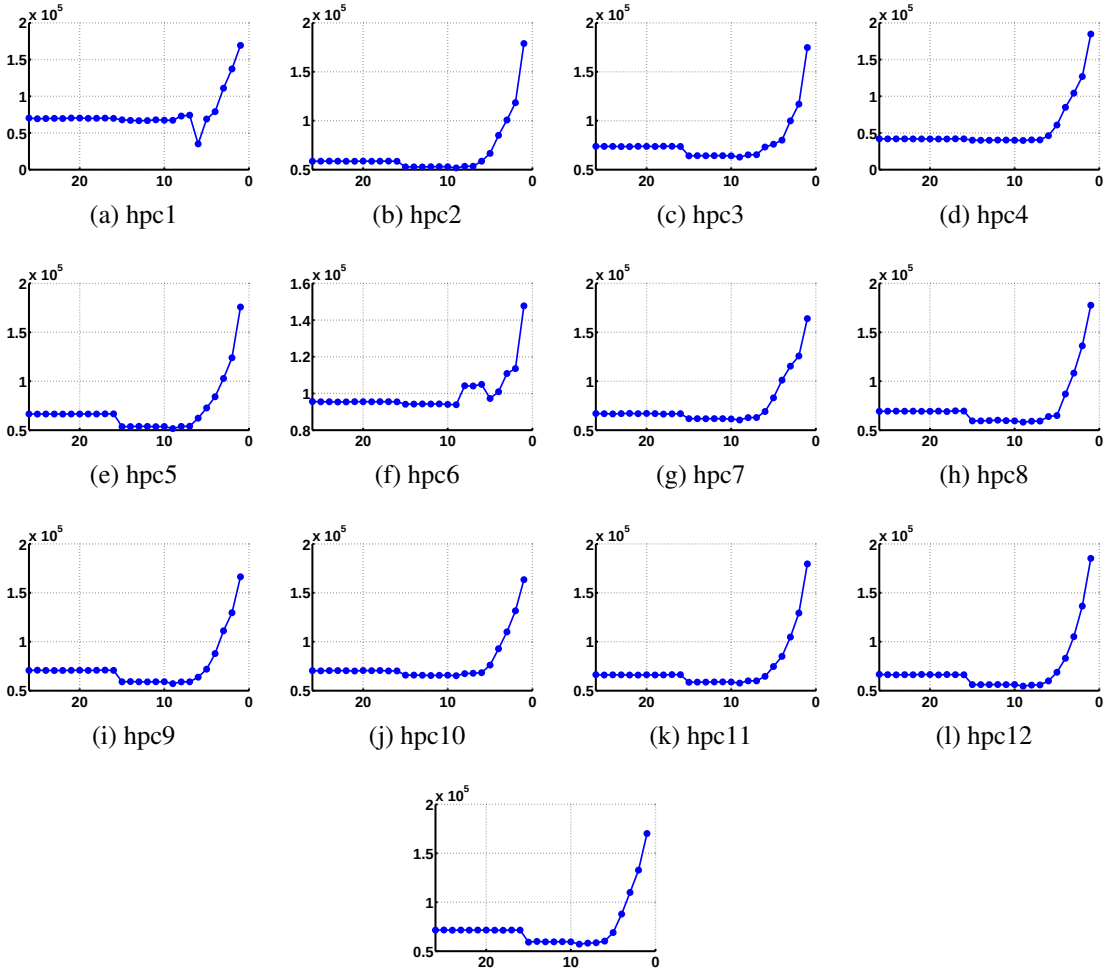
(a) Comm1 (b) Comm2 (c) Comm3 (d) Comm4

(e) Comm5

Figure 5.10: Extracted bit-change pattern for the COMMERCIAL benchmark suite.

Address Mapping Prediction

Given the signature for a set of running applications, the next issue is how to generate an optimised address-mapping scheme. The idea is to map the physical address bits with low variation to rows (to reduce the row switching or page-conflicts), the physical address bits with medium variation to banks and the physical address bits with highest rate of change to columns to increase the locality and decrease the page conflicts. Moreover, it is possible to limit DReAM to re-arrange only some of the physical address bits to mitigate the associated cost of the address mapping change in DRAMs that will be discussed later.



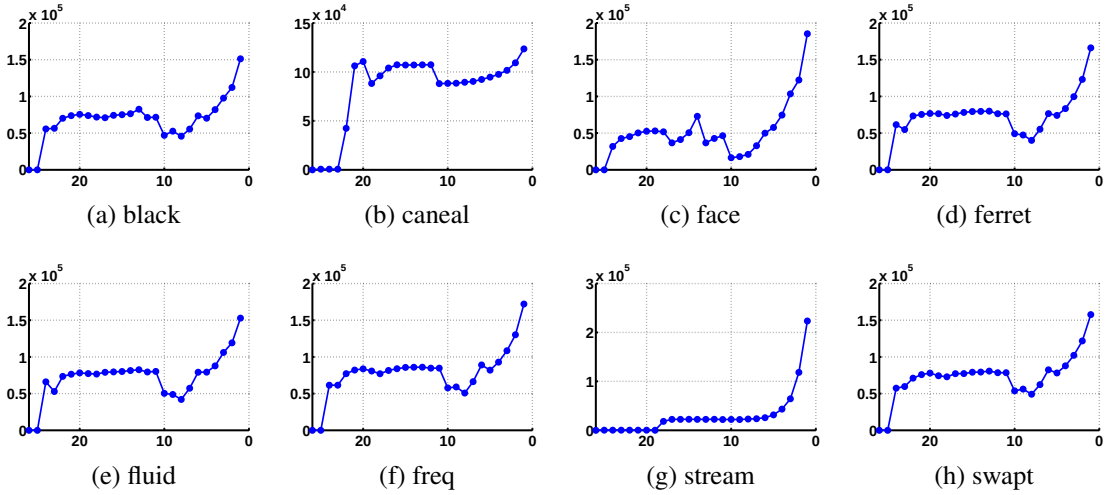
(a) hpc1 (b) hpc2 (c) hpc3 (d) hpc4

(e) hpc5 (f) hpc6 (g) hpc7 (h) hpc8

(i) hpc9 (j) hpc10 (k) hpc11 (l) hpc12

(m) hpc13

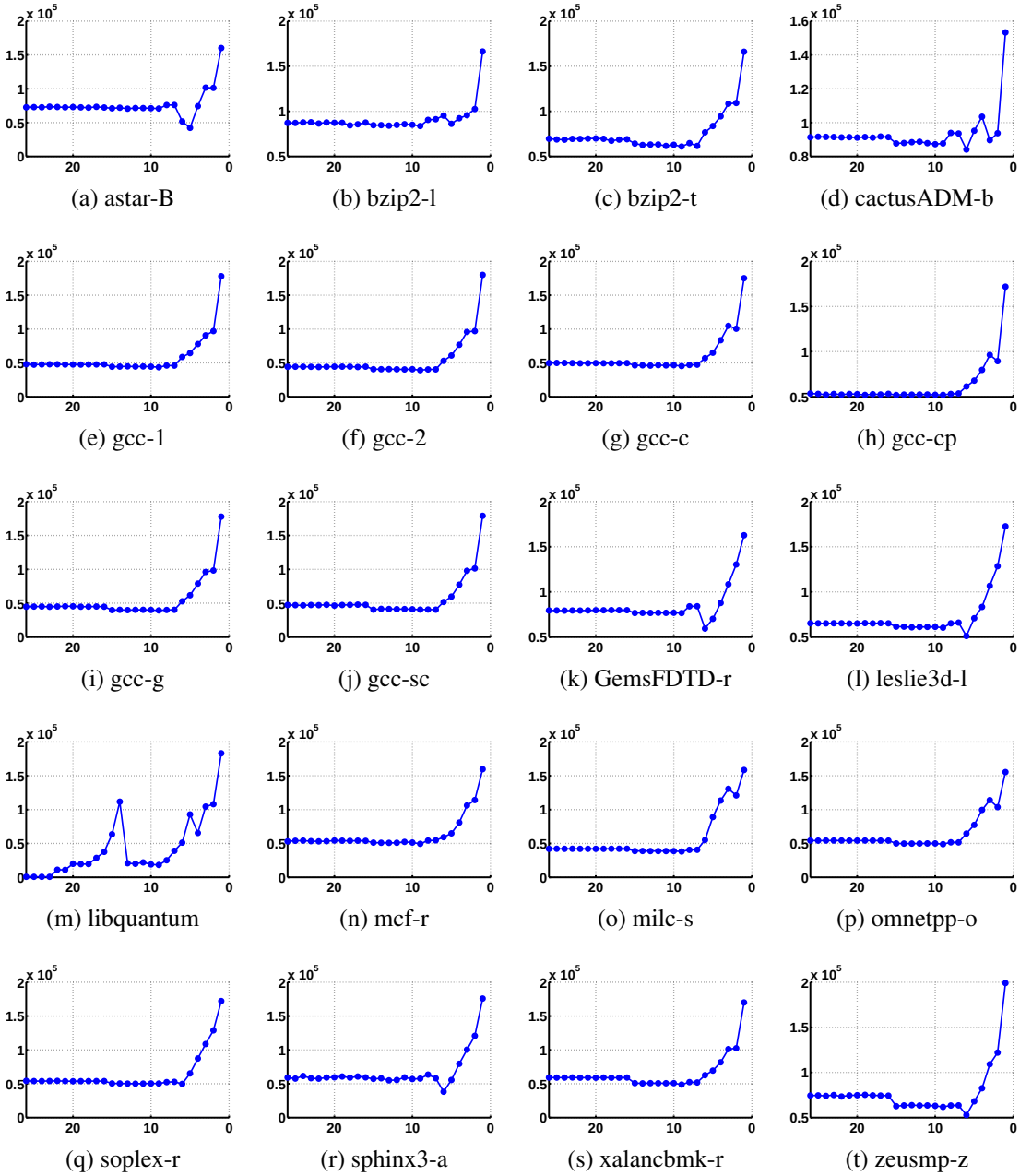
Figure 5.11: Extracted bit-change pattern for HPC benchmarks.



(a) black (b) caneal (c) face (d) ferret

(e) fluid (f) freq (g) stream (h) swapt

Figure 5.12: Extracted bit-change pattern for PARSEC benchmark suite.



(a) astar-B (b) bzip2-l (c) bzip2-t (d) cactusADM-b

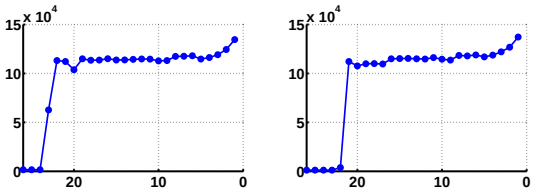
(e) gcc-1 (f) gcc-2 (g) gcc-c (h) gcc-cp

(i) gcc-g (j) gcc-sc (k) GemsFDTD-r (l) leslie3d-l

(m) libquantum (n) mcf-r (o) milc-s (p) omnetpp-o

(q) soplex-r (r) sphinx3-a (s) xalancbmk-r (t) zeusmp-z

Figure 5.13: Extracted bit-change pattern for SPEC benchmark suite.



(a) mummer (b) tigr

Figure 5.14: Extracted bit-change pattern for BIOBENCH benchmark suite.

To produce a new address mapping scheme at run-time the following procedure will be carried out:

- The bit-change rate of physical address bits will be monitored for each time-window.
- A new address mapping scheme will be estimated based on each time-window, monitoring information.
- The bit-change rate monitored, based on the pre-defined and new address mapping schemes for each time-window will be compared.
- If the new address-mapping scheme can improve the bit-change rate in comparison with the baseline address-mapping above a desired (and programmable) threshold (for consecutive time-windows defined by 'Consistency Threshold') then the new address mapping will be used as the primary address mapping scheme in the system.

Mathematical Insight

Intuitively, DReAM proposes a simple technique to detect an application-specific address mapping scheme based on the physical address bit-change monitoring process. However, the question is to find an analytical proof to show that the application-specific address-mapping scheme predicted using this method can actually improve the performance of the memory system.

As discussed, the predicted address-mapping scheme will be exploited only if it can reduce the bit-change rate, in comparison with the baseline address mapping, beyond a certain threshold. This means that DReAM assumes that there is a correlation between the bit-change rate of physical address bits and the performance of DRAMs. To investigate this, the correlation coefficient between the average bit-changed improvement reported by DReAM and the performance improvement of memory system, while using the DReAM address mapping, was investigated. The experimental results show that there is a strong correlation, 0.89 with a very small P-value (i.e. 1.97×10^{-15}), between the bit-change rate and the final performance improvement. This justifies why the predicted address mapping scheme proposed by DReAM can improve the performance of DRAMs.

This section presented how DReAM can detect an efficient address mapping scheme at run-time. Employing such a workload-specific address-mapping scheme aims to distribute the frequently accessed rows across the available banks. However, changing the address-mapping scheme of a DRAM, on-the-fly, has a very important obstacle which is the requirement for Data Migration. Initially, a DRAM places data into memory based on a pre-defined address mapping scheme. Therefore, changing the address-mapping scheme implies that the data previously loaded into the DRAM cannot be accessed using the new address mapping scheme. Thus, before employing the new address mapping, the existing data in DRAMs *must* be migrated to a new location based on the new address mapping scheme.

However, as discussed earlier, it is possible to limit DReAM to re-arrange only some of the physical address bits. For instance, if DReAM rearranges the physical address bits associated with columns then every column in the DRAM must be migrated to a new location. On the other hand, if DReAM only rearranges the physical address bits associated with rows without changing the column address bits then the location of columns inside rows will not be changed. Considering such limitations for DReAM implies different migration costs. This research investigates the data migration at page-level (migrating rows without changing the column location). The *Data Migration* challenge will be discussed in more detail in the next section.

5.3.2 DReAM - Data Migration Solutions

DReAM provides an opportunity for page-conflict reduction in DRAMs and, as a result, improves the performance of the overall memory systems. However, as explained, DReAM also imposes some overhead to the system due to the required data migration process. In this section, different scenarios will be investigated to exploit recent and emerging memory technologies to alleviate the cost of data migration.

Scenario 1 - Offline Data Migration

This scenario explains the simplest DReAM implementation that imposes a minimal hardware overhead on the overall memory system. In general, this scenario is well suited for application-specific computer architectures, such as datacenter and database systems, where a specific application is running on the system repeatedly. For instance, in a database system, depending on the type of database (e.g. financial, medical etc.),

usually only a few specific queries with minor variations are used to search for specific data. Moreover, in the big-data research area running a query over a database might take a few days or weeks. This produces a specific memory access pattern in the system that usually is consistent over a long period of time.

In this implementation, the memory access pattern of applications (single or multithread) will be monitored at run-time for a desired period (e.g. it can be a few hours, a few days etc). This period is called the Region Of Interest (ROI). Ideally, the ROI should be chosen to be long enough to represent the application access behaviour. For instance, if the ROI for a medical database is chosen to be one day, then the memory access pattern of almost all the possible queries that are usually run on the database during the day can be covered by the ROI.

In this situation, DReAM will estimate an optimised address-mapping scheme based on the average bit-change rate extracted from the dedicated counters per physical address bits for the entire ROI. This new mapping will be saved in the memory controller and upon rebooting the system user has an option to choose the DReAM address mapping scheme over the baseline from the system BIOS. Thus, whenever the user reboots the system the memory controller can employ a new address mapping that is estimated based on the DReAM calibration mode. A similar approach has been implemented for the Intel-adaptive page policy and a special beta BIOS provided by ASUS that allows the user to choose a desired page closure policy at system start up [Raj].

In this scenario, there is a penalty for the rebooting process but after that, as far the usual workloads running on the system are concerned, the overall performance of the memory system will be improved by taking advantage of the new address-mapping scheme. This is why this scenario is well suited for systems with consistent behaviour over time.

Scenario 2 - Online Data Migration using Non-Volatile Memory Technology

This scenario takes advantage of a novel non-volatile DIMM structure, that includes flash devices in the same DIMM as DRAM, to avoid rebooting the system as required by the first scenario. For instance, ArxCis-NV [Vika, Vikc, SK14] is a non-volatile DDR3 DIMM which is already available in the market, produced by Viking Technology [webb]. The preliminary aim of ArxCis-NV is to preserve critical data in the event of power or system failure. This memory structure continuously monitors voltage levels from the host system and, in the case of a voltage drop, triggers a SAVE

function which mirrors DRAM states to the flash memory. In this situation, ArxCis-NV uses backup power provided by embedded supercapacitors to transfer data into the non-volatile flash. At reboot time data will be restored from flash similarly.

DReAM will use the functionality provided by ArxCis-NV in a slightly different manner. The only difference is that the ArxCis-NV triggers the SAVE functions in the case of voltage drop but DReAM triggers the SAVE function whenever a new address-mapping scheme is detected at application run-time. Therefore, the existing data in the DRAM will read using the pre-defined address-mapping scheme and copy to the flash, and after that, data will be restored to the DRAM using the predicted address-mapping scheme. According to the datasheet of ArxCis-NV the read and write bandwidth of this module is 4 GB/s which means the content of 4 GB DRAM can be copied in 1 second and restored in 1 second [Vikb]. Therefore, there is a 2 second penalty each time that DReAM changes the address mapping scheme. This delay is much smaller than the delay imposed by rebooting the system.

Scenario 3 - Online Data Migration using NanoCommit

Although the second scenario imposes a lower overhead for the data migration process than the first scenario in reality there is no need to copy all the data back from flash when the address-mapping is changed. The reason is that some of the existing data in DRAM might not be needed anymore by the application. Thus, a more elegant approach would be to return data to the memory only when it has been requested again. To demonstrate this scenario, Memory Channel Storage (MSC) [Diaa] and NanoCommit technology [Diab] can be used which were invented by Diablo Technology [weba].

The main principles behind MSC is to bypass the traditional interface between non-volatile memory (e.g. flash) and memory subsystem to provide a shorter and faster path from a CPU to the massive data storage offered by non-volatile memories. Similar, to the second scenario, MSC also employs flash memory on the same DIMM that houses DRAMs. Diablo Technology has invented another technology on top of MSC which is called NanoCommit [Diab]. Using NanoCommit, all the write operation to the DRAM devices will be also written to the flash memory with a latency of 48 ns. This allows DRAM modifications to be rapidly made persistent in flash.

DReAM can take advantage of MSC and NanoCommit to mitigate the cost of online data migration. Assuming that MSC and NanoCommit are in place, in the case of any changes in the address-mapping scheme using DReAM, there is no need to copy

DRAM contents to the flash since data is already in flash. On the other hand, as discussed, there is no need to copy all the data back to DRAMs from flash before any access to data. A more efficient solution is to move data from flash to the DRAMs based on the memory requests. Therefore, there is no unnecessary cost for migrating data which are not needed anymore. This reduces the cost of data migration significantly in comparison with scenario 2.

Scenario 4 - Online Data Migration using a novel DRAM Structure

The last three scenarios take advantage of existing memory architectures to perform the data migration required by DReAM. However, in all of the aforementioned scenarios, data will be migrated using off-chip communication links which imposes huge overhead to the overall data migration cost. Probably, a more efficient solution would be migrating data inside the DRAM device which requires some modification to the internal structure of this memory system. The fourth scenario investigates the possibility of performing on-the-fly data migration inside a DRAM device by proposing some modification to the internal structure of this memory system. In the following a high-level overview of the necessary steps to perform the online data migration inside DRAM will be discussed.

Basic Procedure

Figure 5.15 presents the basic flowchart of servicing a memory request while using DReAM considering the fourth data migration scenario. As discussed, to minimise the overhead of migration, a row is migrated only when it has been accessed. In practice, this means that the migration occurs gradually as described in the following.

On the first access to a row, the requested physical address is translated to the internal structure of the DRAM using both the Pre-defined Address-Mapping Scheme (PAMS) and the Estimated Address-Mapping Scheme (EAMS). The translated address by PAMS (called address 'A') is the source row address and the translated address by EAMS (called address 'B') is the destination row address. There are two main functions that might be applied on the requested address in different situations which are Migration and Swap. The requirement for these two function and what they are will be discussed later on in this section and they are declared here just for initial familiarity to explain the flowchart.

The first step is to determine if the accessed row is in its original location, pointed to by PAMS, or not. Two bits are dedicated to each row in a DRAM bank to keep

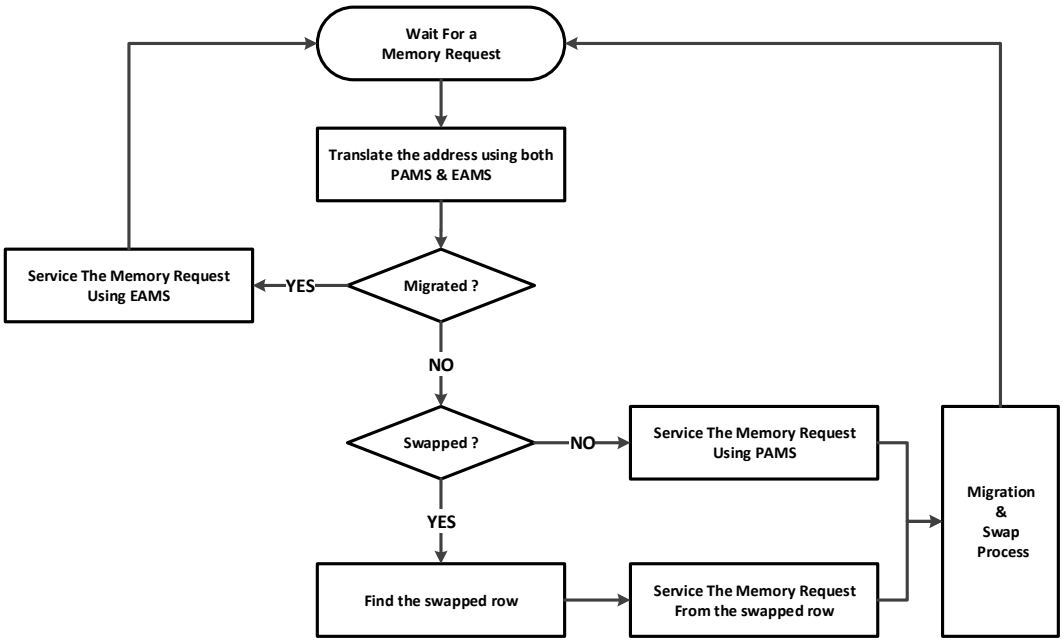


Figure 5.15: DReAM flowchart.

track of the current status of that row: one bit (Migration-Bit) to determine if the row has been moved to its new location (migrated) and one bit (Swap-Bit) to determine if the row has been swapped (this process will be discussed later). Two tables can be dedicated to accommodate these bits for the entire DRAM module: the Migration Table (MT) and the Swap Table (ST). At this point several situations might happen:

- If the requested row is in its original location (the migration-bit and swap-bit are 0) then:
  1. The PAMS will be used to access and service the requested row.
  2. The requested row will be migrated to the destination location pointed by EAMS.
  3. If the destination location is occupied by a different row then intuitively the content of destination row also needs to be migrated to a third place. This can produce a chain of unnecessary data migration which is costly. To avoid this, a simple row-swap algorithm is employed which means that in such situations the content of the destination row will be swapped with the content of the source row (and the corresponding swap-bit will change to 1).
- If the requested row has been migrated:

1. The EAMS will be used to access and service the requested row.

- If the requested row has been swapped:

1. The swapped location will be calculated by applying the reverse address-mapping mechanism to the source location.
2. Step 1 will be repeated until the swap-bit of the location pointed by the reverse address mapping scheme is 0.
3. The request will be serviced.
4. The requested row will be migrated to the destination location pointed by EAMS.
5. A swap will be performed if it is necessary.

To make all this happens inside DRAM some modification needs to be done to the traditional structure of DRAMs which is explained below.

Required DRAM Modification

Modifying the internal structure of DRAMs is a complex and expensive process which needs an intensive evaluation. To have a fair and realistic demonstration of the fourth scenario, two techniques from previously published works are considered: RowClone [SKF+13] and SALP [KSL+12]. RowClone proposed some modification to DRAMs to perform internal bulk data copy and similarly SALP suggested some modifications to the structure of DRAM to exploit subarray-level parallelism. In the following it will be explained how these two techniques can be used to perform the online data migration inside DRAMs.

There are two main requirements for DReAM to perform data migration in a DRAM device: the capability of bulk data copy inside DRAM and the capability of on-the-fly buffering of the entire row to perform the swap operation. Both of these requirements have been studied individually by previous work to address different issues, using existing subarray level parallelism in DRAMs,[SKF+13, KSL+12] which are described briefly in the following.

Bulk Data Copy in DRAM: Seshadri *et al.* [SKF+13] exploits the existing subarrays per bank in DRAMs to copy the entire row from one location to another inside DRAMs. Depending on the location of the source and destination rows, there are three different scenarios that should be considered: (i) copying between two rows within the

same subarray (intra-subarray), (ii) copying between two rows in different subarrays in the same bank (inter-subarray), (iii) copying between two rows in different banks (inter-bank).

- **Intra-subarray:** In this scenario, the source and destination rows share the same row-buffer. Therefore, the copying process involves two main steps: (1) loading the source row into the row-buffer and (2) loading the row-buffer into the destination row. The first step can easily be done by activating the source row which connects the bitlines of the source row to the row-buffer. Therefore, the source row will be loaded into the row-buffer. The next step is simply copy the row-buffer contents to the destination row by connecting the bitlines of the destination row to the row-buffer. However, it is not possible since the source row is still connected to the row-buffer (its wordline is raised) and the original implementation of DRAMs does not allow the raising of two wordlines at the same time. This is because all the rows within the same bank share one row-decoder. On the other hand, although precharging the source row lowers the wordline of the source row it also clears the row-buffer contents. To solve this issue, Seshadri *et al*. [SKF+13] proposed a new DRAM command called DEACTIVATION that only lowers the wordline of the source row without clearing the row-buffer. Therefore, after issuing the DEACTIVATION command to the source row, the destination row can be activated using an Activation command. Hence, the content of row-buffer will be loaded to the destination row. As they evaluated, this scenario requires the minimal modification to DRAM device, 0.0016% die-size overhead.

- **Inter-subarray:** In this scenario, the source and destination rows are located in different subarrays. Therefore, it is not possible to use the wide bitline/row-buffer communication bus to transfer data. Instead, data must be transferred using the 64-bit I/O bus that connects to all the row-buffers inside the same bank as well as to all the row-buffers in different banks. Intuitively, since both subarrays are in the same bank, it is possible to read from source row-buffer from one subarray and write to the destination row-buffer in the other subarray. However, the first problem is that the original read or write commands transfer data on the I/O bus to/from the DRAM device's data-pins which is unnecessary for the purpose of copying data inside DRAM. Therefore, Seshadri *et al*. [SKF+13] proposed a new DRAM command called TRANSFER that reads data from the

source row-buffer and writes it to the destination row-buffer using the I/O bus without transferring data to the chip's data pins. To copy the entire row multiple TRANSFER commands must be issued by the memory controller. Seshadri *et al.* shared that the additional control logic to implement the TRANSFER command (disconnect the I/O bus from data pins) incurs a negligible 0.01% die size increase [SKF+13].

On the other hand, since both subarrays use the same bank's I/O buffer to perform the read and write operation then the bank's I/O buffer has to switch between reading data from the source row-buffer (64-bit) and writing data to the destination row-buffer (64-bit). As discussed in Chapter 2, there is a cost associated with I/O switching between read and write operation. To work around this issue, Seshadri *et al.* [SKF+13] proposed saving one row in each bank as a temporary buffer to first copy data from the source row-buffer to a different bank and then to write it back to the destination row buffer. In this way the source bank's I/O buffer only performs the read and the destination bank's I/O only performs the write operation. They state that the capacity loss associated with one temporary buffer per bank is negligible, 0.0015%.

- **Intra-bank:** this scenario is similar to the previous scenario with one difference; since the source and destination rows are in different banks then they do not share a row-decoder. Therefore, both source and destination rows can be activated at the same time in different banks. Thus, the content of source row-buffer can be transferred using several TRANSFER commands to the destination row-buffer.

Subarray-Level Parallelism: As also discussed in Chapter 2, Kim *et al.* [KSL+12] observed that a DRAM bank is not implemented as a monolithic component with a single row buffer. Instead, each bank consists of multiple subarrays, each of which has its own local row-buffer. Based on this observation they proposed some small modification to DRAMs to be able to exploit subarray level parallelism. They discussed three different levels of modification to DRAM to improve the access latency by making subarrays work independently. Part of this work which, is more interesting from the point of view of this research, is that called MASA. The key idea of MASA is to allow multiple activated subarrays in the same bank. As discussed, MASA imposes (i) a designated-bit latch to each subarray, (ii) a new DRAM command, subarray-select (SA-SEL) and (iii) routing of a new global wire. Based on their experimental methodology, they showed that the required extra latches imposes 0.15% area overhead and

consumes 72.2 µW additional power for each ACTIVATE command. Moreover, they evaluated that there is an extra 0.56 mW of static power in the steady state imposed by multiple activation of subarrays (there is a baseline static power of 48 mW per DRAM chip so this value is negligible). Furthermore, they estimated that the power consumption of the new SA-SEL command is about 50% of the ACTIVATED command [KSL+12] .

Having explained the above techniques, an overview of the DReAM architecture will be explained in the following sections.

5.3.3 DReAM - Overview of Architecture

Figure 5.16 presents a high-level overview of the DReAM architecture. DReAM includes two main phases, Address-Mapping Estimation and Online Data Migration.

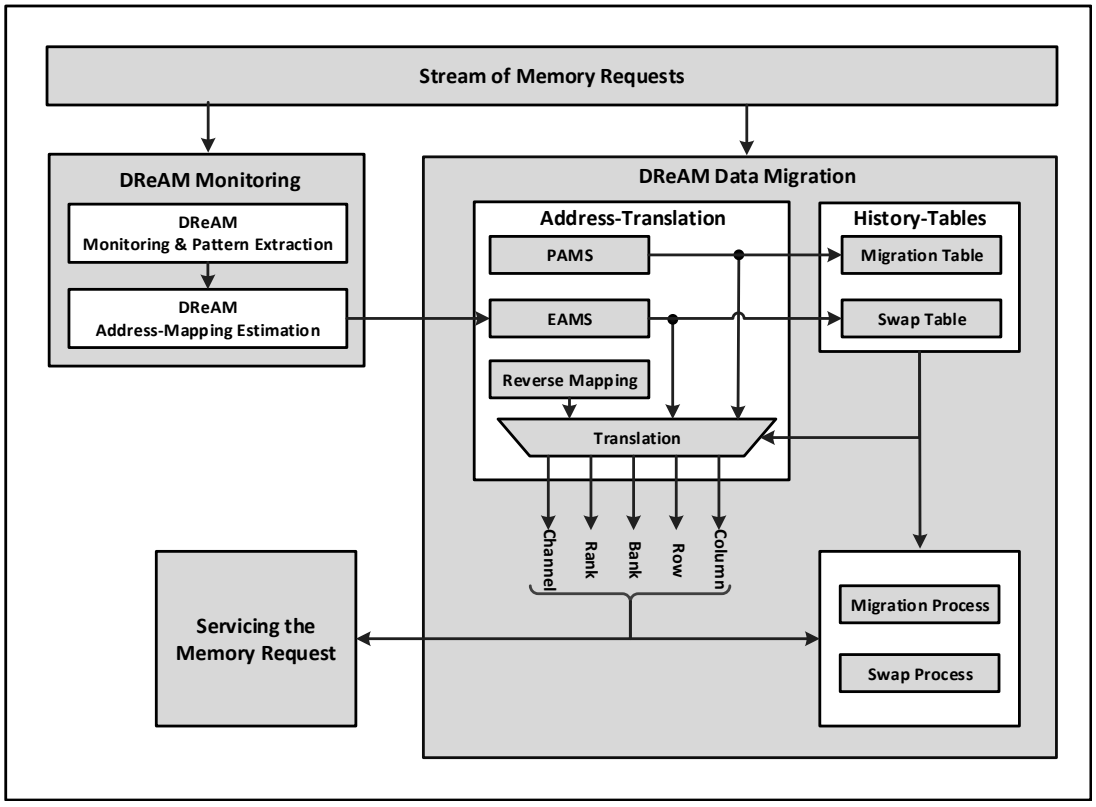


Figure 5.16: DReAM architecture.

Address-Mapping Estimation

Address-Mapping Estimation requires minimal architecture support. Only one counter per physical address bit, a history register to hold the last accessed address and an array of XORs to detect the bit-change between two consecutive memory requests are required to extract the access pattern at run-time. Figure 5.17 presents a simple overview of such a structure. In this structure each bit of the currently accessed address will be XORed with the corresponding bit of the last accessed address. Then, if there is a difference in the accessed bit the corresponding counter will be incremented. This will produce a pattern of physical address bit changes over a period that can be employed to estimate an application-specific address-mapping scheme. The key idea is to map the physical address bits with lower order of changes to rows, physical address bits with higher order of changes to banks and columns to reduce page-conflicts and increase page hits in the system. Moreover, having the information of physical address bit behaviour extracted from DReAM's monitoring unit, one can decide to reorganise the address-bits for a specific purpose rather than just a reduction in page-conflicts.

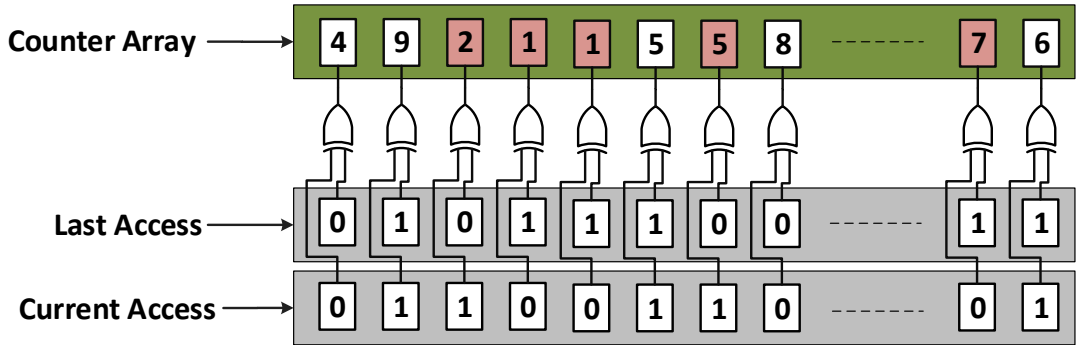


Figure 5.17: DReAM monitoring counter structure.

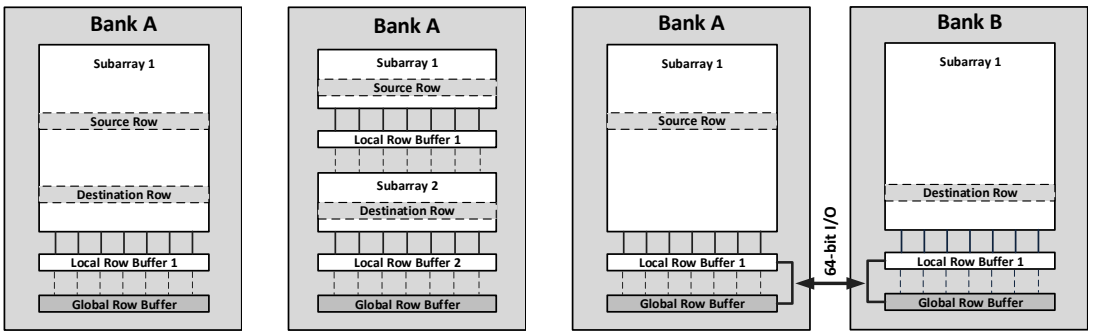
Data Migration - Operation

The Data Migration required by DReAM can be described considering the following observations:

First, all the local row buffers (one local row buffer in each subarray) within a bank are connected to the global row buffer using global bitlines and all the row-buffers (either local or global) within a DRAM device are connected together using a narrow I/O bus (64-bit wide) [Ito01, KSL+12]. Second, considering the modification proposed by Kim *et al.* [KSL+12] the DRAM module supports MASA. This supports multiple

activation of subarrays while only one of them can be connected to the global bitline at a time.

Figure 5.18 presents the possible scenarios in which data migration might happen. To describe the following scenarios it is assumed that the destination row always has been occupied by another row (the worst case scenario) and thus a swap process is necessary.



(a) Intra-Subarray (b) Inter-Subarray (c) Inter-Bank

Figure 5.18: Different data migration scenarios.

Intra-subarray Migration: In this scenario (5.18a), source and destination rows are both in the same subarray. Therefore to perform the migration and swap operation the following procedure will be followed.

1. Activate the source row and load its contents into the global row-buffer.
2. Activate the destination row locally and load its contents to the local row-buffer 1.
3. Connect the local bitline of source row to the local row-buffer. This will copy the destination row to the source row.
4. Connect the global row-buffer to the destination row. This will copy the source row to the destination row.

Inter-subarray Migration: In this scenario (5.18b), source and destination rows are in different subarrays within the same bank. One possible way to swap data in source and destination is to make a copy of the source row in the global row-buffer and then use the narrow I/O bus to transfer the destination row from the local row-buffer 2 to the local row-buffer one. Finally copy the source row from the global row buffer to the local row-buffer two and then to the destination row. However, there is only one I/O per bank which will be used for read and write operations. As discussed before,

reading from one row-buffer and writing back to the other row-buffer imposes extra penalty each time the I/O switches between read and write mode. To workaround this issue, the following procedure is suggested to perform the migration and swap process in this scenario.

1. Activate the source row to the global row-buffer and destination row to the local row-buffer 2.
2. Transfer the source row (located in the global row buffer of bank A) to the global row-buffer of bank B using the narrow I/O bus.
3. Connect the global bitlines of bank A to the local row-buffer 2 to load its content the global row buffer.
4. Connect the global row-buffer of bank A to the local row-buffer 1 and the source row. This will copy the destination row to the source row.
5. Transfer source row from the global row-buffer in bank B to the local row-buffer 2 in bank A.
6. Connect the local bitlines of the local row-buffer 2 to the destination row. This will copy the source row to the destination row.

Inter-bank Migration: in this scenario (5.18c), source and destination rows are in different banks. Therefore, both of source and destination rows can be activated in parallel. Thus, the following procedure is suggested to perform migration and swap process for this scenario:

1. Activate both source and destination row and load their contents into their local row-buffer.
2. Put bank A into the read mode and put bank B into the write mode.
3. Transfer the source row from local row-buffer 1 in bank A to the global row buffer of the bank B using the narrow I/O bus.
4. Put bank A into the write mode and put bank B into the read mode.
5. Transfer the destination row from local row-buffer 1 in bank B to the global row buffer of the bank A using the narrow I/O bus.
6. Connect the global bitlines of the global row-buffer in bank A to the source row and the global row-buffer of bank B to the destination row.

Data Migration - Timing Overhead

The main procedure to perform the online data migration has been discussed so far. In this section the timing overhead imposed by data migration will be discussed. In practice, considering the existing bank-level parallelism in DRAMs, it is possible to overlap some parts of the data migration process with other functions in the system. For, instance it is possible to perform data migration from bank 'A' to bank 'B' while other banks in the system are servicing memory requests. However, for the sake of simplicity, in this chapter it is assumed that all the DRAM functions will be stalled while data migration is in process.

As discussed, data migration can happen in three different scenarios: Intra-subarray, Inter-subarray and Inter-bank. The three scenarios have been explained but, considering the main aim of address re-mapping, minimising the page conflicts, it is possible to ignore the first two scenarios (i.e. Intra-subarray and Inter-subarray). The rationale behind this is that in these two scenarios a row will be swapped with another row within the same bank. Therefore, considering the page-conflicts definition, changing a row location within the same bank cannot help to avoid this phenomenon. Then, based on the available information in MT and ST tables DReAM will not perform the intra-bank data migration which will reduce the overall overhead.

Considering the above observations, the third data migration scenario is the only one that imposes extra latency to the overall execution time of a running application. Thus, the latency overhead imposed by the data migration for each workload is simply the number of inter-bank migration times cost of transferring a row using the internal narrow I/O (i.e. 64-bit) bus. Considering the transfer rate of 64 bits/clock and a row buffer size of 4 Kbit (per device) then 64 clock cycles are required to transfer a row from one bank to another. Another 64 clock cycles are required in the case that a swap is necessary. Therefore in the worst case scenario, the penalty for each data relocation between two banks is 128 memory clock cycles. Assuming that the CPU clock cycle is 4 times faster than the memory clock cycle then the data migration penalty is 512 CPU clock cycles. In a very pessimistic situation it is assumed that the processor will be stalled while the data migration is happening. Therefore the 512 clock cycles times the number of required inter-bank data migrations is an extra overhead imposed on the overall execution time.

Rollback Process to Avoid Degradation Loop

DReAM predicts an application-specific address mapping scheme based on the monitoring period of the past application access pattern. However, it is not guaranteed that the application access pattern will not change again in the future. Therefore, the predicted address-mapping scheme by DReAM might not be efficient anymore and, as a result, using such an address-mapping scheme might degrade the performance of the DRAMs (i.e. Degradation Loop). To work around this issue, DReAM supports a 'Rollback' procedure.

As discussed in Section 5.3.1, DReAM will switch to the predicted address-mapping scheme if the new mapping can improve the bit-change rate in comparison with the baseline, over a pre-defined threshold, for consecutive time windows (defined by the 'Consistency Threshold'). A similar approach will be used to evaluate the efficiency of the predicted address-mapping at run-time. DReAM keeps monitoring the bit-change pattern over the time-windows even after a new address-mapping scheme is predicted. If the bit-change improvement of the predicted address-mapping scheme no longer outperforms the baseline DReAM will switch back to the pre-defined address mapping scheme. This triggers the roll back function to return the migrated rows to their original location. In this situation the memory controller can switch between at least two address-mapping schemes based on the application access pattern. A third address mapping scheme can be employed if the rollback process completes which means that all the rows migrated by the previous address-mapping have moved back to their original locations.

5.4 Evaluation Methodology

This section describes the evaluation methodology used to investigate the performance of DReAM in different situations. Although Section 5.3.2 described four different data migration scenarios, only the first (Section 5.3.2) and the last (Section 5.3.2) scenarios are evaluated in this thesis. The reason is that the workloads available in this work are not representative of applications, with a very long execution time, desired to evaluate scenarios 2 and 3.

Simulator: as described in Chapter 3, USIMM [CBS+12] was used as the main simulation platform for these experiments. USIMM was modified to support Permeation-based Page Interleaving [ZZZ00] and Minimalist Open-Page scheme plus a full implementation of the DReAM architecture. DReAM was evaluated based on a 4 GB

DRAM organised in 1 Channel, running single core applications. To increase the randomness of memory access patterns the size of memory was fixed while running multithread applications. The USIMM system configuration parameters that were used in the experiments of this chapter are captured in Table 5.1.

Model	Description	Value
Processor	Clock Speed	3.2 GHz
	Pipeline depth	10
	ROB size	128
Memory System	Bus Speed	800 MHz
	Number of Channels	1
	Ranks per channel	1
	Bank per rank	8
	Row per bank	65,536
	Cache line per row	128
	Cache line size	64 Byte

Table 5.1: USIMM configuration parameters.

Scheduler: The basic structure of the baseline USIMM scheduler is presented in Figure 5.19. In this structure, every memory request will be placed in either the read or write queue. In general, write requests have less priority than read requests since they are mainly write backs from the last level cache. Therefore the scheduler gives the higher priority to service the read requests. The write queue has a low and high water mark. The scheduler keeps servicing read requests until the number of write requests passes the high water mark. Then the scheduler services the write queue until the number of write requests drops below the low water mark. A FR-FCFS scheduling algorithm is used in our experiments.

Address Mapping Schemes: The memory access pattern, and as a result the number of page conflicts in DRAMs, can be affected by the pre-defined memory address mapping scheme. The experiments consider three different address mappings presented in Figure 5.3. The experimental results presented in Section 5.2.1 (Figure 5.4) show that the Permutation-based Page interleaving policy (Mapping 2) performs best for most of the workloads. Therefore, this address mapping scheme is employed as a fair baseline to compare with the DReAM scheme.

Workloads: As described in Chapter 3, the workloads include a wide range of memory intensive applications (i.e. 48 workloads) from different benchmark suites (PARSEC [BKSL08], SPEC [Dix91], BIOBENCH [AJW+05], HPC and COMMERCIAL) and representative regions of interest for each application. To recap, Table 5.2

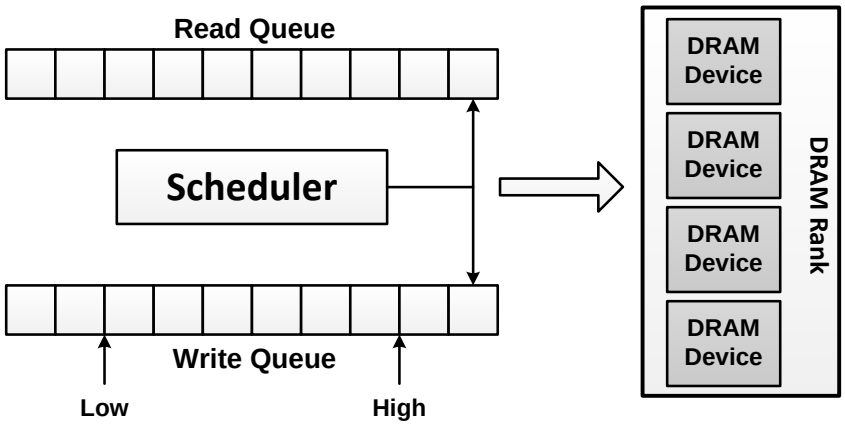


Figure 5.19: Basic structure of the USIMM scheduler.

lists the workloads and their corresponding benchmark suites. An identifier is assigned to each application to facilitate the naming of multithread workloads constructed from these applications.

Benchmark Suites			
SPEC		**PARSEC**	**COMMERCIAL**
(a) GemsFDTD_r	(k) astar_B	(u) canneal	(D1) comm1
(b) bzip2_l	(l) bzip2_t	(v) streamcluster	(D2) comm2
(c) cactusADM_b	(m) gcc_1	(w) blackschols	(D3) comm3
(d) gcc_2	(n) gcc_c	(x) facesim	(D4) comm4
(e) gcc_cp	(o) gcc_g	(y) ferret	(D5) comm5
(f) gcc_sc	(p) mcf_r	(z) fluidanimate	**BIOBENCH**
(g) milc_s	(q) omnetpp_o	(A) freqmine	(E) mummer
(h) soplex_r	(r) sphinx3_a	(B) swaption	(F) tigr
(i) xalancbmk_r	(s) zeusmp_z	**HPC**	
(j) libquantum	(t) leslie	(C) hpc1 - hpc13	

Table 5.2: Evaluated workloads and benchmark suites.

To increase the variety of memory access patterns, USIMM was set up for multi-applications to produce 20 random workload mixes; a combination of 4-thread and 8-thread applications. Table 5.3 lists these multi-core workloads employing the prefix of single thread workloads presented in Table 5.2.

Multithread Workloads	
MIX1: (C13-C5-x-t)	MIX11: (C9-C13-C5-w-x-t-j-q)
MIX2: (C9-w-j-q)	MIX12: (C8-C3-w-x-y-a-t-j)
MIX3: (w-x-y-t)	MIX13: (C8-C5-x-y-a-t-p-q)
MIX4: (C8-C5-t-p)	MIX14: (C9-C12-C13-C9-C12-C12-p-q)
MIX5: (t-t-p-g)	MIX15: (C13-x-t-g-p-t-p-g)
MIX6: (C8-w-p-q)	MIX16: (C8-C3-C5-w-C5-C5-p-q)
MIX7: (C3-C5-C5-C5)	MIX17: (C9-w-y-w-w-a-t-g)
MIX8: (C9-w-y-w)	MIX18: (C13-C3-x-C13-a-a-p-g)
MIX9: (C12-C13-a-a)	MIX19: (C12-C13-y-a-a-a-g-q)
MIX10: (x-t-j-q)	MIX20: (x-y-p-a-x-a-p-q)

Table 5.3: Randomly generated multithread workloads.

5.5 Results and Discussions

This section investigates the performance of DReAM for different scenarios over a wide range of workloads.

5.5.1 Bit-Change Rate vs Performance Improvement

As discussed in Section 5.3.1 there is a strong correlation between bit-change rate and performance improvement. Figure 5.20 shows the bit-change rate improvement reported by DReAM and the final performance improvement achieved using the predicted address-mapping scheme by DReAM. The correlation between these two metrics can be observed from this figure. The experimental results show that there is a correlation coefficient of 0.89 with a very small P-value (i.e. 1.97×10^{-15}), between the bit-change rate and the final performance improvement.

5.5.2 Performance Analysis

In this section the performance improvement of DReAM will be investigated. Before jumping to the result graphs the following summary might be helpful:

- The performance numbers presented in in this section are normalised to the baseline (Permutation Address-mapping) which delivers the best average execution time amongst three address-mapping schemes presented in Figure 5.3.
- Figures 5.21 to 5.24 show the normalised execution time for different benchmark suits.

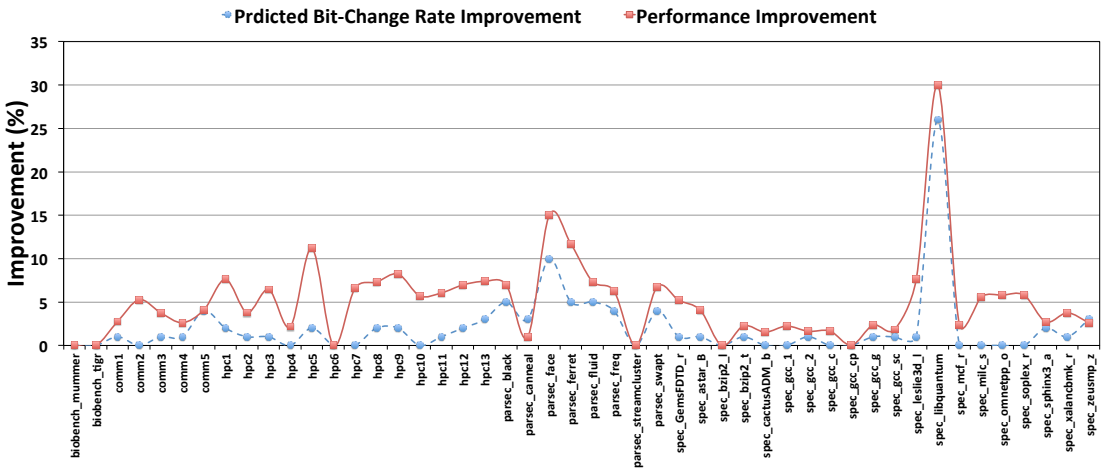


Figure 5.20: Comparison between the bit-change rate improvement predicted by DReAM and the overall performance improvement.

- The presented results for DReAM-Offline in Figures 5.21 to 5.24 correspond to the first data migration scenario described in Section 5.3.2. As discussed, the offline mapping is desired only in the case of applications with a consistent behaviour and will be achieved after a long calibration period (long ROI). Therefore, the rebooting cost will be negligible considering the long running period of the application. Thus, in the results presented in these figures the cost of rebooting is ignored and only the efficiency of the address mapping detected by DReAM in comparison with the baseline mapping is investigated.

Figure 5.21 presents the execution time, normalised to baseline, for BIOBENCH and COMMERCIAL benchmarks. This result suggests that the baseline address-mapping scheme is good enough for the workloads presented in these benchmarks and DReAM is not able to predict a better address mapping scheme. Therefore, there is no bit-change rate improvement when using DReAM in comparison with the baseline and the small degradation by DReAM-Offline (i.e. around 1%) manifested in Figure 5.21 is due to slightly different access patterns caused by re-ordering the baseline address bits. This can be counted as noise.

On the other hand, DReAM-Online mitigates this issue by on-the-fly checking the bit-change improvement, between two consecutive time windows, against a predefined threshold. For instance in these experiments DReAM-Online employs the new address mapping only if it can improve the bit change rate by more than 7%. Thus, although DReAM cannot predict a better address mapping scheme than the baseline it does not degrade the performance for most of the cases. A similar behaviour can be observed

for Figure 5.22 to Figure 5.24.

Overall, the DReAM-Offline outperforms the permutation-based address-mapping scheme (the best evaluated baseline) by 5%, on average, and up to 28% across all the workloads. In the case of the DReAM-Online, 12 workloads satisfy the DReAM's threshold at run-time (i.e. improve the bit change rate by more than 7%) and for these workloads the DReAM-online outperforms the baseline by 4.5%, on average, and up to 23%.

Considering the results presented in Figure 5.24, *libquantum* achieved a significant performance improvement taking advantage of DReAM. To justify this outcome, it is useful to have a look at the extracted pattern for this workload presented earlier in Figure 5.13m. This figure shows that there is a high change rate for bit 14 (see the overshoot in this figure). As discussed in the corresponding section, this bit is mapped to the row address space. This increases the possibility of accessing different rows within the same bank (i.e. Page-Conflict) and so imposes a significant performance overhead. DReAM simply assigns this bit to the column address space by replacing it with a bit with a minimal change rate. In this situation, the excessive change rate of this bit increases the possibility of accessing different columns within the same row (i.e. Page-Hit) which improves the performance significantly. This is why this workload has achieved such a considerable performance improvement.

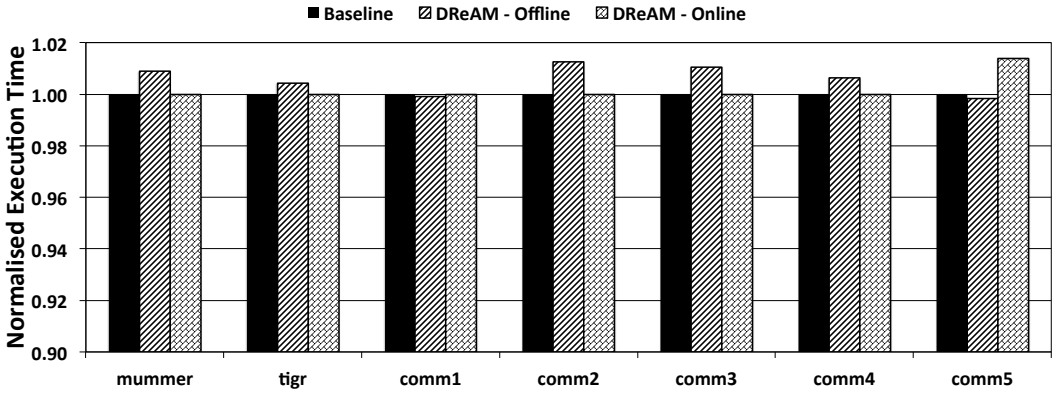


Figure 5.21: Execution time (normalised to baseline) achieved for BIOBENCH and COMMERCIAL benchmark suites.

Figure 5.25 depicts the execution time normalised to the baseline for the randomly selected multithread workloads presented in Table 5.3. These results show that DReAM can still predict a better address mapping scheme than the baseline even in the case of multithread workloads which produce a highly random memory access pattern.

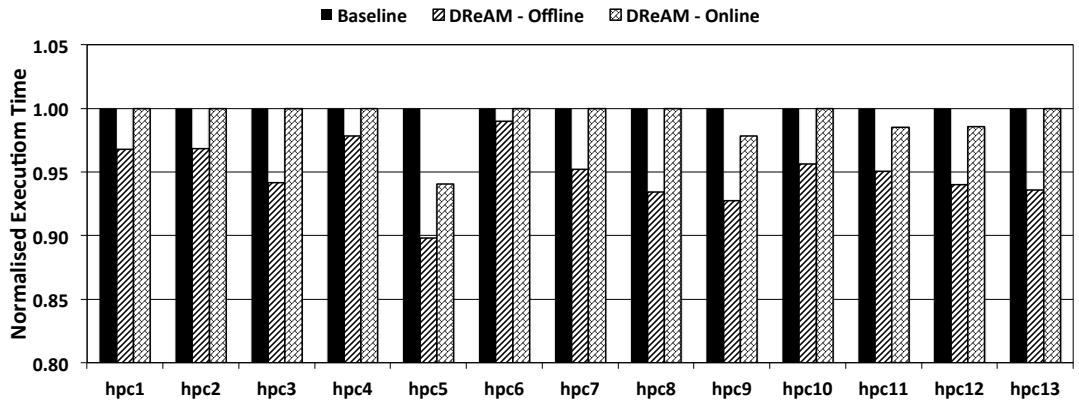


Figure 5.22: Execution time (normalised to baseline) achieved for HPC benchmarks.

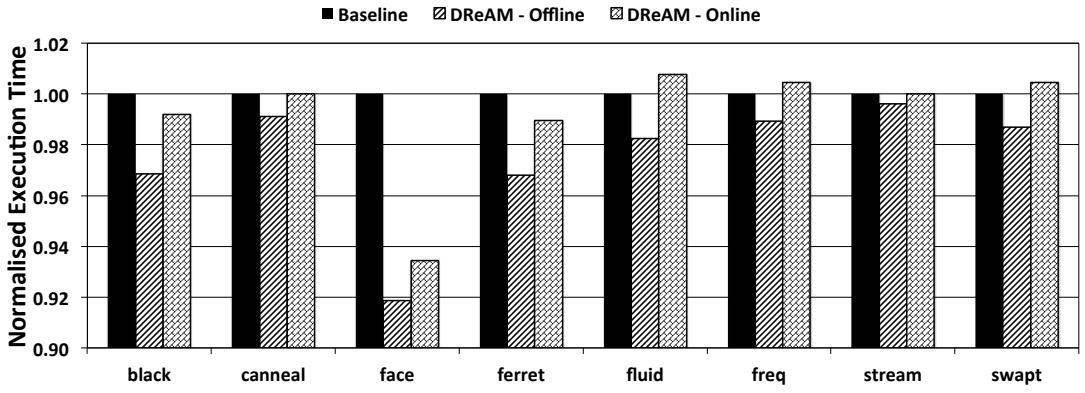


Figure 5.23: Execution time (normalised to baseline) achieved for PARSEC benchmark suite.

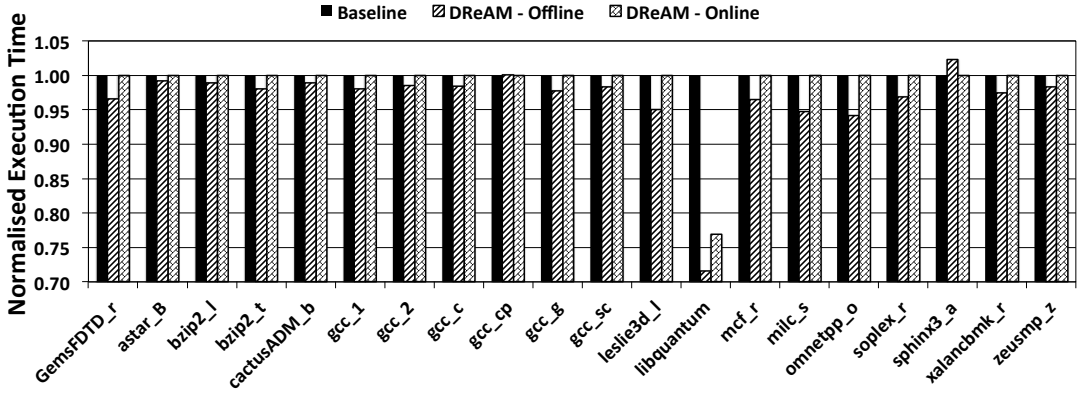


Figure 5.24: Execution time (normalised to baseline) achieved for SPEC benchmark suite.

To conclude, the DReAM performance is independent of single or multithread application and only depends on the final memory access pattern produced at the memory

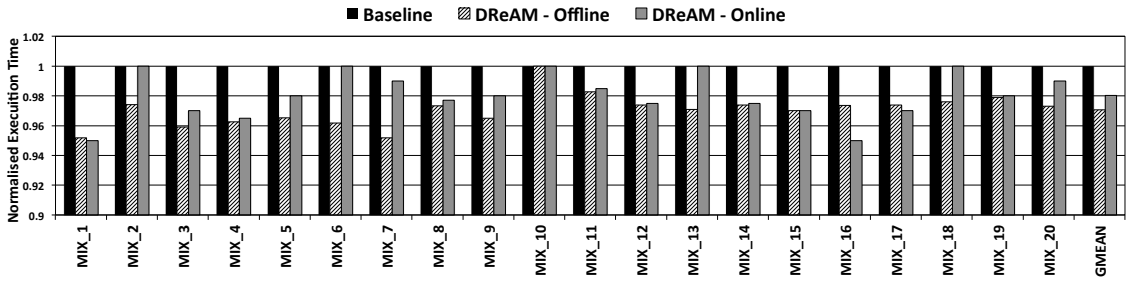


Figure 5.25: Final execution time (normalised to baseline) achieved for multithread benchmarks.

interface level. This pattern can either be produced by a single thread or multithread workloads. In this situation, depending on the extracted pattern from the dedicated physical address bit counters, DReAM might be able to predict a more efficient address mapping scheme than the baseline.

5.5.3 Data Relocation Analysis

As discussed, the data relocation required by DReAM is composed of two main scenarios: Migration and Rollback. To recap, migration happens as soon as a new address mapping scheme is detected, then the previously located data in DRAM needs to be migrated to the new location. On the other hand, if DReAM detects that the new address mapping does not outperform the baseline address mapping scheme then it rolls back the data to its original location. In the following some statistical analysis of migrations and rolls back required by DReAM will be discussed.

Migration vs. Rollback

The experimental results (presented in Figure 5.21 to Figure 5.24) show that 12 standard workloads undergo dynamic data relocation. Out of these 12 workloads only two workloads require data rollback which are ‘ferret’, with 10% of data relocation spent on data rollback, and ‘libquantum’, with 39% of data relocation spent on data rollback.

Inter Bank vs. Intra Bank Data Relocation

Figure 5.26 presents the intra and inter bank distribution of data relocation required by DReAM. According to this figure 87.5% of data relocation happens between banks (Inter-bank relocation) and 12.5% happen within banks (Intra-bank relocation). As

discussed, DReAM does not perform the Intra-bank scenarios to reduce the data relocation's cost.

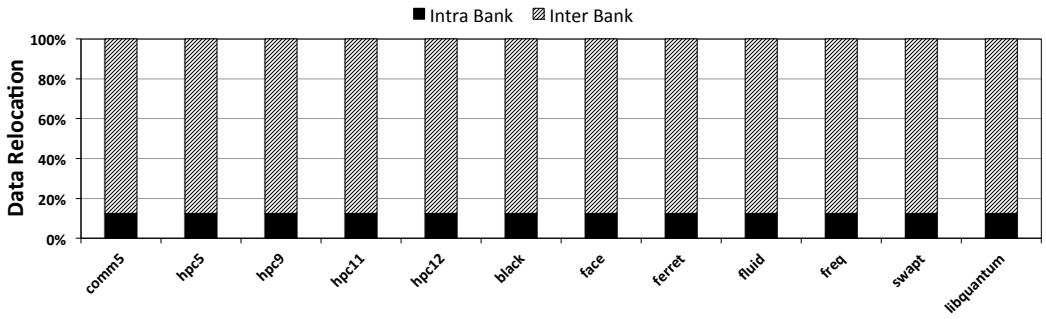


Figure 5.26: The analysis of inter and intra bank data relocation required by DReAM online.

5.5.4 Storage Overhead and Scalability

The storage overhead imposed by DReAM will be discussed in this section. As discussed, DReAM consists of two main phases: Address Mapping Prediction and Data Migration. The associated cost of implementation for both of these phases are as follows.

Address Mapping Prediction: As discussed in Section 5.3.1, there is only one counter and one XOR gate per physical address bit and one history buffer to keep track of the last access address is required to extract the monitoring pattern. Thus, assuming a sampling window of 250K memory requests, 18-bit counters times the number of physical address bits are the main storage overhead for the first phase of DReAM. Figure 5.27 presents the storage required by the address-mapping prediction phase of DReAM for different memory sizes. This is effectively negligible as an area overhead.

Data Migration: According to the discussion in Section 5.3.2, the first three data-migration scenarios do not require any storage overhead. However, the fourth scenario (i.e. online data migration) needs to keep track of migrated and swapped pages. Therefore the required MT and ST impose extra storage overhead to the overall memory system. Figure 5.28 depicts the overall storage overhead imposed by online data migration.

The result presented in Figure 5.28 is based on the assumption that all the existing rows in a DRAM will be migrated which is not a fair assumption. This is because the application access pattern of individual workloads has a limited 'memory footprint'.

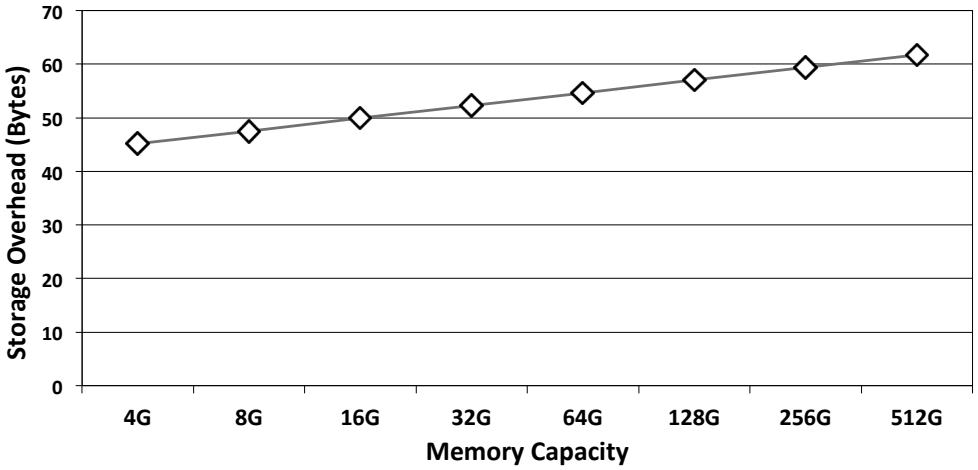


Figure 5.27: DReAM address-mapping prediction phase implementation cost.

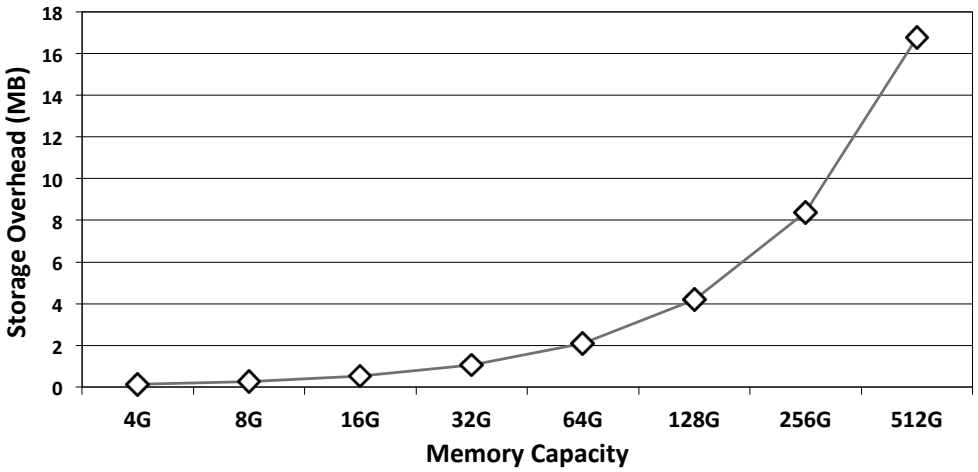


Figure 5.28: DReAM data migration phase implementation cost.

Here, the memory footprint is defined as the portion of memory space that each workload required.

In practice, it is possible to limit the total number of rows that can be relocated at run-time (*Partial Data Migration*). To investigate the probability of the partial data migration, Figure 5.29 presents the memory footprint for all the workloads evaluated in this chapter. This result shows only 3.7% average memory footprint for all the workloads. Moreover, it shows that 85% of the evaluated workloads access less than 5% of all the rows in the system.

Considering the above observations the associated cost of data migration in the case that there is 5% (to investigate average cases) and 25% (to investigate the corner

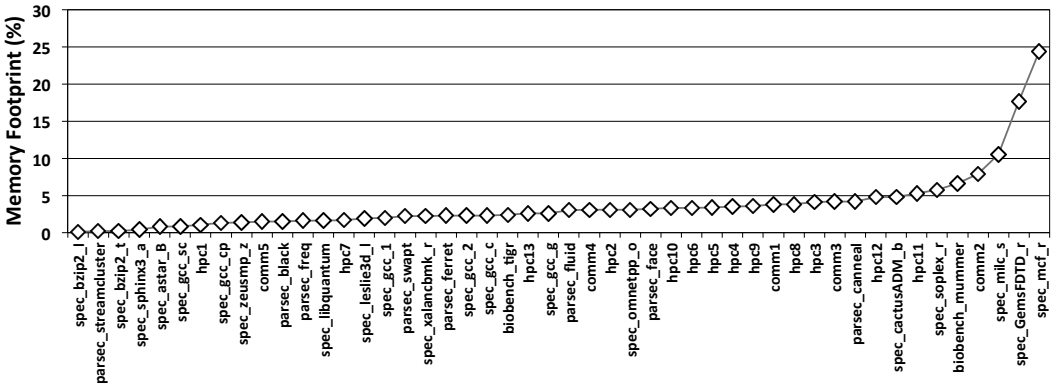


Figure 5.29: Memory footprint for all the evaluated workloads.

cases) limitation in the number of migrated pages were investigated (Figure 5.30). According to Figure 5.30 the storage cost of data migration can be reduced significantly by considering the partial data migration.

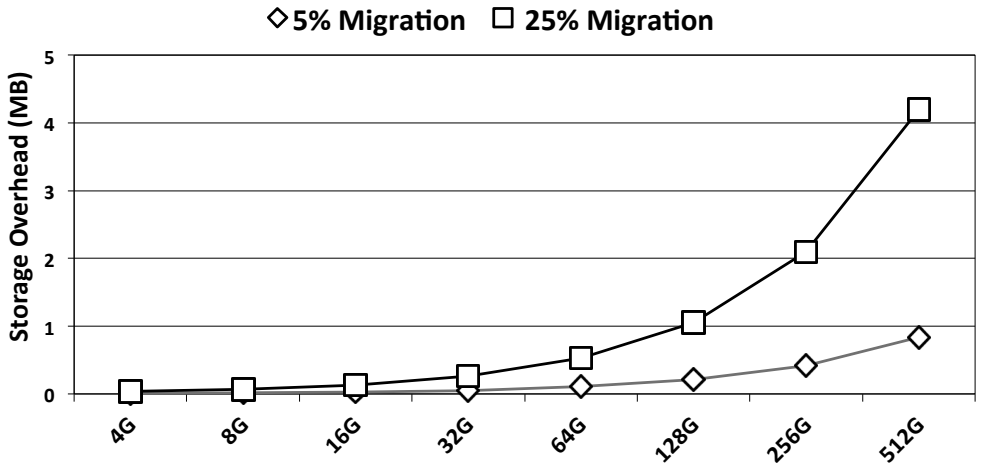


Figure 5.30: Associated cost of partial data migration for DReAM.

5.6 Related Work

The shortfalls of DRAMs with respect to page conflicts are widely recognised in the area of memory system design. Prior work proposed a wide range of different techniques such as memory interleaving schemes, scheduling algorithms and some architectural modifications to the current structure of DRAMs to mitigate this issue. For instance, Zhang *et al.* [ZZZ00] proposed a page interleaving scheme to reduce page

conflicts and exploit data locality. Hsu *et al.* [HS93] proposed another memory interleaving scheme to address the same issue. There are many other interesting works in the area of developing new scheduling algorithms[EMF+11, IMMC08, KHMHB10, KPMHB11, MM07, MM08, NALS06] that prioritise servicing certain memory requests to reduce page conflicts and improve the memory performance. Some other types of work in this area are those that propose either a new architecture for DRAMs or a small modification to the traditional structure of these memory systems. For instance, Sudan *et al.* [SCN+10] propose a technique to recognise the highly accessed data in DRAM and place them in the same row to improve the data locality. Kim *et al.* [KSL+12] proposed a technique to exploit the existing subarray level parallelism in DRAMs to improve the bank conflicts. PARDIS by Bojnordi *et al.* [BI12] is a programmable memory controller that can be configured using a specific instruction set architecture (ISA). Although the focus of this work was not on developing optimised address-mapping scheme they configured PARDIS by the application-specific address mapping heuristic achieved by offline profiling analysis and presented a good performance improvement in the memory system.

5.7 Summary

This chapter introduced DReAM (Dynamic Re-arrangement of Address Mapping) which is a novel hardware technique based on approximating the entropy of each memory address bit for a set of memory requests. DReAM identifies which bits are changing the most (higher entropy) and which change the least (lower entropy). DReAM presents three main contributions: first, a low-cost pattern recognition technique is developed to extract the memory access pattern at run time. Then, a methodology is proposed to estimate an optimised address-mapping scheme based on the detected access pattern. Finally, a technique is proposed for the on-the-fly migration of data within DRAMs to reduce page conflicts.

An extensive performance evaluation was carried out with 48 different workloads from 5 benchmark suites. By keeping the memory size constant while increasing the number of cores the randomness of the behaviour of the memory access pattern is increased significantly, which might impose a high rate of page conflicts in the memory system. In such a situation DReAM shows that it is still able to detect the application access pattern and estimate an optimised address-mapping scheme that improves the

performance of the memory systems in comparison with the baseline mapping. Overall, DReAM-Offline outperforms the permutation-based address-mapping scheme (the best evaluated baseline) by 5%, on average, and up to 28% across all the workloads. In the case of DReAM-Online, 12 workloads satisfy DReAM's threshold at run-time (i.e. improve the bit change rate by more than 7%) and for these workloads DReAM-online outperforms the baseline by 4.5%, on average, and up to 23%.

DReAM is complementary to existing schedulers in memory controllers and is the first on-the-fly mechanism capable of generating workload specific address-mappings without requiring running applications to be stopped.

Chapter 6

ARMOR: A Run-time Memory hot-row detectOR

6.1 Introduction

Despite nonvolatile memory technologies starting to make inroads, the main memory market for computer systems is dominated by DRAM. As DRAM manufacturers continue to move to smaller technology nodes, they are trying to optimize manufacturing costs and memory performance without degrading reliability [MDB+02, JHG13]. The increased density and smaller storage cells make DRAM cells more susceptible to different type of noise such as electromagnetic coupling between cells [FK91, MSY+90, TTK+92].

In June 2014 Kim *et al.* [KDK+14] empirically demonstrated that despite the best effort to mitigate disturbance errors, a high percentage of DRAM modules suffer from this phenomenon (i.e. 110 modules out of 129 DRAM modules sourced from three different memory manufacturers were tested). In particular for the most recent modules, those dated in the last two years, all but one module was affected. The corruption of stored data was achieved by memory requests requiring a particular DRAM row to be activated above certain thresholds; this has been called the *row hammer effect*.

When dealing with a row hammer scenario, the data corruption affects the physically neighbouring rows, not the highly activated row. Thus, it is possible to modify the contents of a physical DRAM row (typical sizes 1KB-4KB for DDR3) by activating a different row. In other words, segmented memory or page protection cannot guarantee isolation of two or more programs when they are using memory mapped to

adjacent physical rows. For example, consider a program which uses the SSE instruction `clflush` (not privileged) and allocates regions of memory which are page aligned and of size equal to the OS page size. If the program allocates half of the physical memory available but randomly frees whole OS pages, then the physical address space will be highly partitioned. Just hammering the end and beginning of each page by repeatedly calling clflush may be enough to corrupt the other programs' data with a high probability.

Google's Project Zero engineers have recently exploited the vulnerability of DRAMs to row hammer error to gain kernel privilege and run unauthorized code [Mara]. In another case study, they demonstrated how a Native Client (NaCl) [Chr] program can escalate privilege to escape from NaCl's x86-64 sandbox and obtain the ability to call the host OS's syscalls directly. They have also released c-code that can be run on personal computers to check if the system susceptible to row hammer error [Marb]. According to their experiments, 15 out of 29 evaluated laptops from different vendors were susceptible to the row hammer error.

Security attacks using the `clflush` instruction may not look threatening as the program would have had to already gain execution rights on the machine. However, more severe security implications would occur if simply accessing a web page was enough to generate row-hammer effects. Javascript applications have become ubiquitous on web pages (e.g. Google.com, Facebook.com, Youtube.com, Yahoo.com, Baidu.com, Wikipedia.com, Twitter, Amazon.com, etc.) and, by default in Chrome, Firefox, Safari and Internet Explorer, Javascript is enabled; less than 2% of web users disable Javascript. Despite Javascript being initially an interpreted language, sophisticated JIT compilers are now the norm; e.g. Chrome uses V8 [Goo]. Although Javascript applications will not have access to the `clflush` instructions, row-hammer could be achieved by taking into account the set-associativity and generating mapping misses in the Last Level Cache (LLC).

For simplicity, assume a LLC with 2-way set associativity and size 4KB. A C program with an array of integers X of size at least three times bigger than the LLC, e.g. 16KB, and the following inner loop would generate numerous evictions (mapping misses) and updates to main memory:

```
// C code
int X[]; // 16KB
...
X[i] += 1;
// occupy one entry in a given set
X[i + 1024] +=  2;
// occupy another entry in the same set
X[i + 2048] += X[constant] + X[constant + 1024]
// eviction due to mapping miss
...
```

The equivalent Javascript program uses Typed Arrays and would look very similar:

```
// Javascript code
var X[] = new Int32Array(4 * 1024); // 16KB
...
X[i] += 1;
// occupy one entry in a given set
X[i + 1024] +=  2;
// occupy another entry in the same set
X[i + 2048] += X[constant] + X[constant + 1024]
// eviction due to mapping miss
...
```

Current LLC from Intel, AMD, ARM, PowerPC and UltraSparc are predominantly 16-way or 8-way set associative with the largest LLC holding 30MB. With this information, it is possible to modify the above Javascript snippet to generate evictions and memory updates due to mapping misses.

Motivated by security, this section proposes an efficient and scalable technique, called ARMOR (A Run-time Memory hot-row detectOR), to overcome disturbance errors in DRAMs. This is the first work that detects hot-rows at run time with guarantees as well as allowing reduction in the level of accuracy in a controlled manner. The evaluation of ARMOR compares it directly with the PARA technique [KDK+14] for standard benchmarks as well as kernels representing malicious codes.

6.2 Background on Row Hammer Error

This section presents general background on the internal structure and operation of DRAMs to explain the row-hammer data corruption effect.

Figure 6.1 presents a low-level structure of a DRAM cell. Each DRAM cell consists of a transistor and a capacitor which are connected to the bitline wire and wordline wire. Typically, a fully charged capacitor holds logical '1' and fully discharged capacitor holds logical '0' value. A wordline is connected to all cells located in a row and a bitline is connected to all rows in a column. To access a row the corresponding wordline will be raised to a high voltage. This operation enables all the transistors in the target row and, as a result, the capacitors will be connected to their corresponding bitlines. In this way, the row data (capacitors' charges) is amplified by the sense amplifier, and is transferred to the row buffer ready for reading and updating. This entire operation is called *row activation*. In this process the capacitors will be discharged and restored to their former value. At the end, by lowering the wordline the transistors will be turned off and capacitors will be disconnected from bitlines [JNW10, Ito01, Kee08].

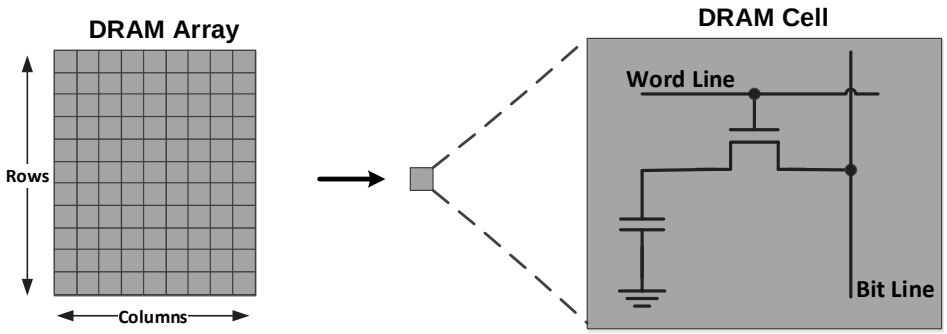


Figure 6.1: DRAM Cell.

6.2.1 DRAM Refresh

Considering the electronic circuit of a DRAM cell (Figure 6.1) the charge stored in these cells is not persistent. This is because capacitors will lose their electric charge over time due to various types of leakage [SHU+00, RMMM03]. Thus, each DRAM cell has a limited retention time. According to the DDR3 DRAM specification [SPE09], each DRAM cell has a retention time of *64 ms*, which means after 64 ms, DRAM cells are susceptible to lose their data. Therefore, all the DRAM cells must be *refreshed* within 64 ms to sustain data reliability.

To refresh a DRAM cell the memory controller issues a *Refresh* command at a specific time interval to make sure that all the rows within the system are refreshed at least every 64 ms. Originally, a refresh command was issued, by the memory controller, to each row in the system. However, as the size of memory and number of rows increase in modern DRAM devices issuing one refresh command per row becomes impractical. In this situation the memory controller only issues a fixed number of refresh commands within a refresh period (e.g. 8192 Refresh Commands) and each refresh command refreshes multiple rows at the same time.

6.2.2 Row Hammer Effect - Corrupting Data without Writing

The 'Row-Hammer' effect is not well known, but was highlighted by a test equipment company called Teledyne LeCroy [Mik, Mica] in the context of DDR4 DRAM. They observed that 'aggressive' activations of a specific row in a DRAM can corrupt adjacent rows' data. In reality this phenomenon is a specific type of Disturbance Error that occurs due to intensive interaction between electronic components that are supposedly isolated from each other [KDK+14].

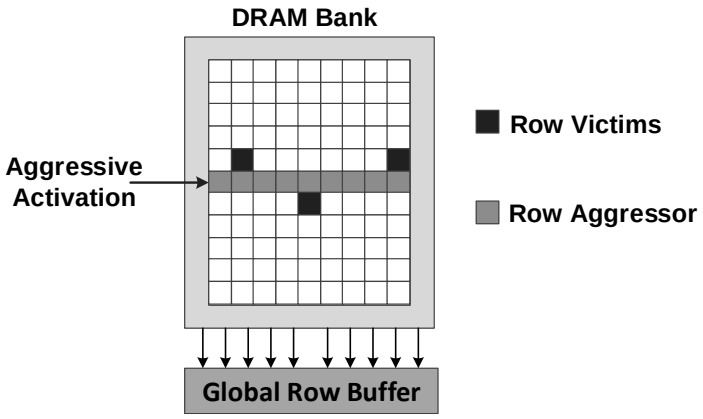


Figure 6.2: Row Hammer phenomenon.

Row-hammer in DRAMs can occur when a specific wordline of a DRAM is activated repeatedly within one refresh interval. In this situation the neighbouring cells leak charge at faster rate than expected. Thus, the retention time of such cells is reduced to less than 64 ms which means that these cells may lose their data (charge) before refresh happens. Subsequently, in refreshing these corrupted cells the wrong data will be read and written back again (figure 6.2). Moreover, ECC modules are not very efficient in this situation since they cannot detect multi-bit errors.

Kim *et al.* [KDK+14] provided the first empirical study in a peer-reviewed comprehensive study that demonstrated the existence of disturbance errors and more specifically the row-hammer effect in commodity DRAM devices. This chapter focuses on proposing a hardware solution to overcome the row-hammer issue instead of proving its existence.

Possible Intuitive Solution: One straightforward solution to mitigate *row-hammer* is to simply increase the refresh rate for all the rows in the memory system. Although, this approach might alleviate the row-hammer issue it imposes an unnecessary power and performance overhead to the system. Probably the most intuitive solution is to detect the rows with high activation value (i.e. 'hot' rows) and refresh their (physical) neighbours. A simple method to recognise hot rows in a DRAM is to dedicate a counter per row to keep track of the number of activations of each row. However, having one counter per row introduces a significant area and power overhead to the memory controller. Kim *et al.* [KDK+14] investigated general techniques used to find frequent items from a stream of items, such as: Bloom Filters [Blo70, CM03], Morris Counters [Mor78] and some other standard techniques [KSP03] to identify the hot rows. However, none of the mentioned techniques is scalable as the size of main memory increases or can guarantee successful row hammer identification/correction. Thus, Kim *et al.* proposed Probabilistic Adjacent Row Activation (PARA) to overcome row hammer. The key idea behind PARA is that every time a row is opened and closed one of its adjacent rows is refreshed (e.g. opened) with some low probability. Thus, if a particular row is opened and closed repeatedly then, statistically, the adjacent rows will be refreshed. However, due to the probabilistic nature of this technique and considering that the row hammer phenomenon is a security issue, PARA is not a reliable solution to address this error.

This chapter proposes a native memory hot row detector instead of trying to apply the mentioned traditional techniques to find the hot rows.

6.3 Row Hammer: Analytical Analysis

To develop an optimised solution for row-hammer it is important to understand when this phenomenon might happen. There are a few significant factors that should be taken into consideration when analysing row hammer problems that are briefly explained next.

Refresh Interval (RI): it has been discussed that every 64 ms the DRAM must refresh all the rows in the system. Therefore, RI is the maximum time period over which row-hammer events can occur.

Activation Threshold ($\mathbf{ACT_{th}}$): the minimum number of activations required to induce the error is called the ACT_{th}. Kim *et al.* [KDK+14] have done an extensive evaluation of row-hammer effects on a wide range of DRAM devices (i.e. 129 DRAM Modules) from three major DRAM manufacturers and they ended up with three different ACT_{th} for this set of DRAM devices which are presented in Table 6.1. ACT_{th} of 139k is used in the experiments presented in this chapter.

Modules	$\mathbf{ACT_{th}}$
Module A	139K
Module B	155K
Module C	284K

Table 6.1: ACT_{th} for different evaluated DRAM modules in [KDK+14].

Minimum Activation Interval ($\mathbf{MIN_{AI}}$): the minimum interval to activate a row within the DRAM device is limited to T_{RC} – that is the time interval between accessing a row and restoring data to the DRAM array plus the precharge time. In the experimental results presented in this chapter, MIN_{AI} is around 49 ns ($MIN_{AI} = T_{RC} = 49$ ns).

Maximum Possible ACT per RI ($\mathbf{MAX_{ACT}}$): considering the above definitions the maximum possible number of activation commands that can be issued to a bank within RI is:

$$MAX_{\text{ACT}} = \frac{RI}{MIN_{\text{AI}}} = \frac{64ms}{49ns} \approx 1.3\ M \tag{6.1}$$

Aggressor and Victim: the repeatedly opened rows are called 'aggressor rows' or 'hot-rows', if they reach the ACT_{th}, that cause the error and the neighbours of an aggressor row which may be affected by row hammering are called 'victim rows'.

Maximum Possible Aggressors Per RI ($\mathbf{MAX_{aggressor}}$): considering the MAX_{ACT} and the ACT_{th}, the maximum possible number of aggressor rows per RI, assuming $ACT_{\text{th}} = 139K$, can be calculated from equation (6.2).

$$MAX_{\text{aggressor}} = \frac{MAX_{\text{ACT}}}{ACT_{\text{th}}} = \frac{1.3M}{139K} \approx 10 \tag{6.2}$$

$MAX_{aggressor}$ suggests that there can only be a few aggressor rows (≈ 10 *aggressor rows*)

per bank per RI out of the millions which exist in the system. This leads to the fact that there must be a specific behaviour in the activation stream to create aggressor rows, which is one of the main principles behind ARMOR.

6.4 ARMOR: A Run-time Memory hot-row detectOR

It has been shown that there can only be a few potentially critical row aggressors within RI (e.g. $MAX_{\text{aggressor}} \approx 10$). Thus, a solution is to only keep track of potential row-aggressors' activation behaviour rather than monitoring all the available rows in the system. The main challenge is then how to detect the potential hot-rows in the system.

6.4.1 ARMOR - Basic Principles

In the previous section, it was suggested that there should be a specific behaviour in the stream of activation issued to a bank to produce row-hammer problem. To identify such an activation pattern at run time, assume that the activation stream to hot-rows follows a uniform distribution. This means that there is a fixed time interval between each activation that goes to a hot-row in a DRAM bank. In this situation, it is possible to define a minimum requirement that each row must satisfy to be identified as hot.

Potential Hot-Rows Condition: considering the previous assumption (e.g. uniform distribution of activation stream to a hot-row), a time window is defined, e.g. Hot Time Window (HTW), in which a row must be activated at least once to be identified as a potential hot-row within RI (Figure 6.3).

$$HTW = \frac{RI}{ACT_{\text{th}}} = \frac{64ms}{139K} \approx 460\ ns \tag{6.3}$$

As equation (6.3) shows, the RI is divided to 139K different time slots (of 460 ns each) which means that to have at least one hot-row within an RI, there should be at least one activation to a specific row in each time slot, otherwise the target row is not going to reach the ACT_{th}. Therefore, if there are no repeated activations to the target row within HTW it is not going to be a hot-row within the RI. However, if there are repeated activations to the target row within HTW then the target row *might* be a potential hot-row, depending on the activation stream behaviour in future time slots. In this way a simple structure can be developed to filter potential hot-rows at run time.

So far a uniform distribution of the activation stream is assumed for the sake of simplicity to explain the basic principle behind ARMOR; this is not always a valid

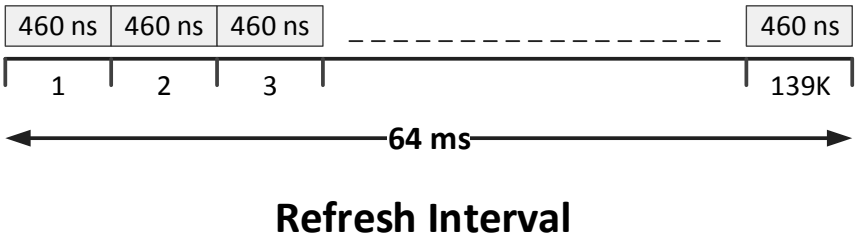


Figure 6.3: Hot time windows.

assumption. The activation stream depends on the application behaviour and memory access pattern. Considering multithread and multicore systems the application access pattern – and as a result the activation stream – is most likely 'random'. The next section explains how to use similar principles to detect hot-rows at run time for uniform and non-uniform distributions in the activation stream using a simple counter (i.e. a credit counter).

6.4.2 ARMOR - Overview of Architecture

In this section, the concepts presented so far are used to propose a novel and low-cost architecture to identify the hot-rows in DRAMs. Figure 6.4 presents an overview of the key components of ARMOR's structure. At a high level of abstraction there are two main phases to detect a hot row using ARMOR. In the first phase the potential hot-rows will be filtered using a Time-based Shift Register Filter (TSRF) and in the second phase a Dynamic Counter Allocator (DCA) will allocate one counter to each potential hot-row to keep track of the number of activations to these rows. Moreover, DCA will deallocate counters and evict the potential hot rows when they lose their eligibility to be a hot row at run time.

Time-based Shift-Register Filter (TSRF)

The first phase in identifying hot-rows is to detect a potential row that might reach the ACT_{th} or, in other words, filtering non-potential hot-rows to reduce the number of rows that need to be tracked.

To implement such a filter, a simple time-based shift register structure is proposed which is presented in figure 6.5. As its name implies, the contents of this shift register will be shifted to the left every specific time interval. This time interval is chosen to be equal to MIN_{AI} to capture activation stream behaviour in fine-grain time windows. Every MIN_{AI} the TSRF checks if an activation command has been issued in that period;

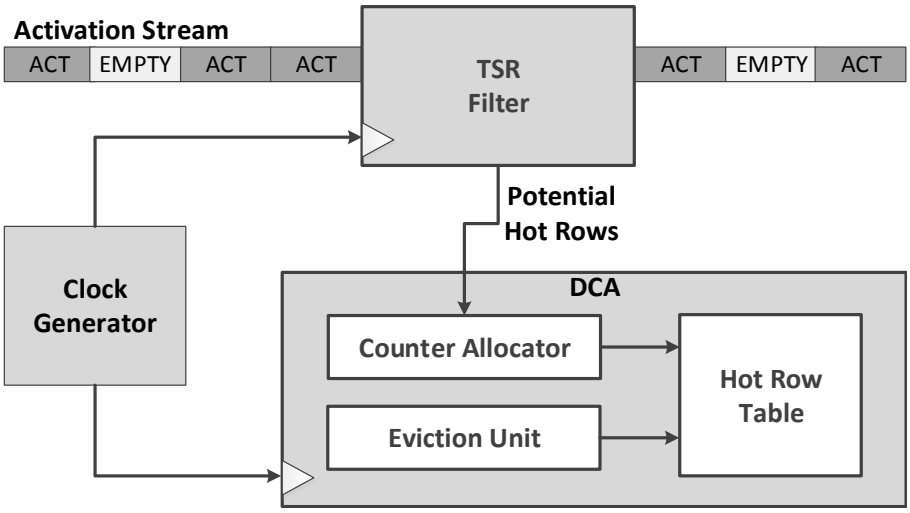


Figure 6.4: ARMOR overview.

if it has, the address of activated row will be stored in the last location of shift register and, if there is no activation command in that time period, the last location will be marked as empty and the contents of shift register will be shifted to the left by one location.

The shift register length is chosen in such a way that it accommodates the maximum number of possible hot-pages within RI (or $MAX_{aggressor}$) plus one (equation (6.4)). The extra register is to overlap the two consecutive HTWs. Thus, if an activation happens in an exact HTW time interval it will be captured by the TSRF. Moreover, since the registers of TSRF are updated every MIN_{AI}, this queue length is guaranteed to capture all the possible hot-rows if they happen in the same HTW ($\approx 460\ ns$). The TRFS' registers' size is made big enough to hold the required address-bits representing the activated row.

$$TSRF\ Size = MAX_{\text{aggressor}} + 1 = 10 + 1 = 11 \tag{6.4}$$

Having this structure, the physical address of the activated row will be kept in the queue for the maximum time of HTW. Every MIN_{AI}, before discarding the TSRF's last register, it will be compared to all the items in the queue to see if there is another activation to the same address, if there is, then it will be marked as a potential hot-row and sent to the next stage; otherwise it will be discarded. The output of the filter is therefore a stream of addresses which have been activated more than once within the sample window. This must be satisfied at least once in a refresh interval if a row is

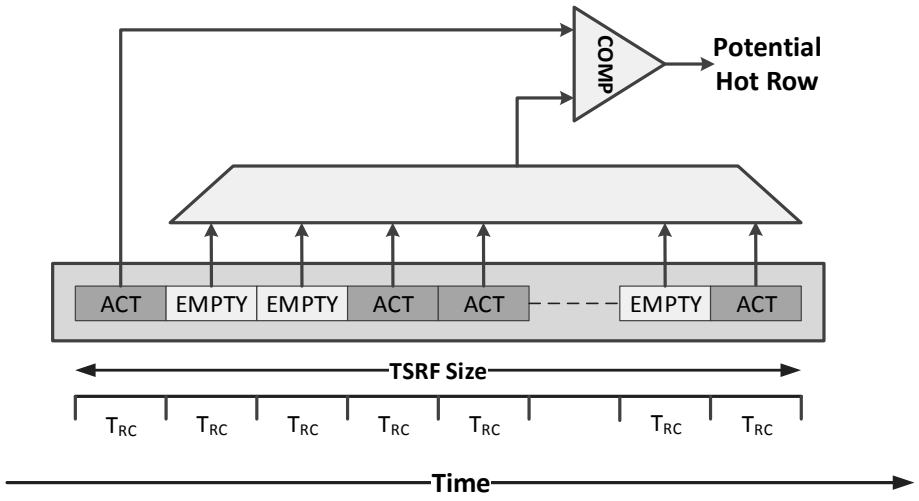


Figure 6.5: Time-based Shift Register.

to be 'hot'. Although, in this way, an initial filtering is carried out on the activation stream, some identified potential hot-rows might not remain hot-rows at the end of the RI.

Further Filtering Improvements: The experimental results show that the current structure of the TSRF delivers satisfactory results. However, the performance of this filter can be improved further considering a dynamic threshold to detect the hot-rows. It has been discussed that having a minimum of two activations to the same row within the HTW would be a minimum threshold to consider a row as a potential hot-row. However, this threshold is calculated based on the overall RI of 64 ms. It means that, as time progresses, the condition for a row to be a potential hot row will change as well. For instance if 32 ms of the RI has passed then, considering our uniform distribution assumption, the minimum number of row activations within the remaining HTW must be more than two (e.g. four) if the target row is to reach the ACT_{th} and be a hot row.

Dynamic Counter Allocator (DCA)

The DCA is in charge of allocation and deallocation of counters to detected potential hot-rows. It comprises a simple allocator and eviction unit plus a potential hot row table. An overview of the DCA structure is presented in Figure 6.6.

As soon as a potential hot-row is identified by the TSRF, its physical address will be sent to the DCA unit. The DCA consists of simple counters, timers and registers

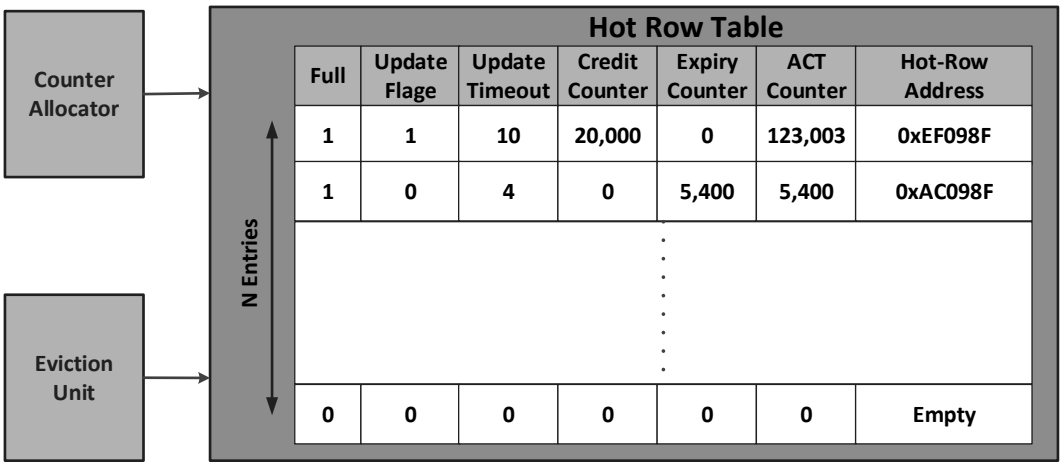


Figure 6.6: Dynamic Counter Allocator.

which are explained as follows.

Full Flag: is a 1-bit flag to specify if the current entry of the hot-row table is in use or free.

Hot-Row Address: is the address of the potential hot row.

ACT Counter: is a counter allocated to a potential hot-row to keep track of the number of activations of that row.

Update Flag: is 1-bit flag to show if the corresponding potential hot-row has been activated in the last MIN_{AI} or not.

Update Timeout: is a timer that keeps track of the HTW period. If there is no activation to a potential hot-row within the HTW then the potential hot-row becomes a candidate to be evicted from the hot-row table. This timer will be reset to its initial value after every activation to the corresponding potential hot-row.

Credit Counter: The presented architecture so far can only detect hot-rows with a uniformly distributed activation stream. The Credit Counter is a simple, intelligent way to extend the design to be able to detect a hot-row with a non-uniformly distributed activation stream. It has been discussed that one necessary requirement (but not the only one) for a specific row in DRAM to be a hot-row is to have at least two activations per HTW (i.e. every 460 ns) otherwise the potential row becomes a candidate for eviction from the hot-row table. However, depending on the activation stream distribution, there might be a case that there are several activations to the target row within one HTW but no activations for the next few HTWs (a non-uniform distribution). In this situation, the target row still has a potential to reach the ACT_{th} and become a hot row. In this

case, the Credit Counter comes into play. This counter keeps track of all the extra activations (i.e. more that two activations) within the HTW. This extra credit specifies the number of future HTWs that the target row can still remain a potential hot-row even if there is no activation to that row. This can be implemented simply by decrementing the credit counter every time that that there is no activation to the potential hot rows within a HTW (i.e. $Update\ Timeout = 0$).

Expiry Counter: is a counter that prioritises the order of eviction from the hot row table if there are several potential hot-rows which are flagged as candidates to be evicted. In this way, a candidate row with a higher number of activations will be evicted later than one with a lower number. It provides extra time credit to the row with higher number of activations to remain a potential hot-row if there are more activation commands to this row.

Finally, if a potential 'hot-row' in the table reaches the ACT_{th} it will be flagged as a hot-row and the corresponding signal will be sent to the memory controller for further actions which are described in the next section. Also, the detected hot-row will be evicted from the hot row table.

Hot-Row Table Size: considering the possibility of having the maximum number of hot rows within RI (e.g. $MAX_{aggressors} \approx 10$) the maximum required hot row table size to accommodate all the possible hot rows is equal to $MAX_{aggressors}$ per each bank. Thus, a maximum of 80 entries in the hot row table (820 Bytes) is required to support a 4 GB DRAM system populated in 8 banks. However, this number is calculated for the worst case scenario and the experimental results show that the possibility of having $MAX_{aggressors}$ per RI per bank is negligible. Therefore, a much smaller hot-row table would be enough to detect all hot rows in most situations.

The experimental results presented in section 6.7 show that ARMOR can identify all the possible hot-rows (based on a desired ACT_{th}) in the system with a high level of count accuracy of number of activation (i.e. 99.99%).

6.5 ARMOR Applications

In the previous section, ARMOR is proposed to detect hot-rows in DRAMs. In general, having a knowledge of which are the hot-rows in a memory system provides an opportunity to improve different aspects of functionality of these architectures. This section discusses how ARMOR can be used to improve the reliability and performance of DRAMs.

6.5.1 Target Row Refresh

It has been suggested that the most intuitive way to solve the row hammer issue is to detect the hot rows and refresh their neighbouring rows. Since ARMOR provides the solution to detect hot-rows in DRAMs, then the remaining component is to refresh the victim rows. This requires that the DRAM manufacturers disclose the mapping scheme of logical to physical rows to the memory controller. This enables the memory controller to issue the refresh command, or a simple activation command, to the physically adjacent rows. Alternatively, a new DRAM command can be issued which simply refreshes the neighbour rows of specific address. Such a capability has been added to the latest Micron DDR4 devices called 'Target Row Refresh (TRR)' Mode [Micd]. ARMOR is a perfect complement of a DRAM system equipped with TRR. The performance overhead imposed by ARMOR using TRR is discussed below.

Performance overhead: one of the main advantages of this work compared with the solution presented by Kim *et al.* [KDK+14] is that, since ARMOR detects the exact row-aggressors, it only refreshes the necessary victim rows. This means that, in the worst case, for each detected row-aggressor *two* adjacent rows need refreshing. Thus, this imposes a maximum cost of two extra page-misses (i.e. $\approx 2 \times T_{\mathrm{RC}}$) to the memory system. While, in the solution presented in [KDK+14] the refreshed rows might or might not be row victims. This imposes an unnecessary overhead to performance and power of the memory system. This extra overhead will be evaluated in section 6.7.

In the following a new approach is presented to overcome the Row-Hammer phenomenon using the unique capabilities provided by ARMOR.

6.5.2 ARMOR Cache Solution

The experimental results and mathematical analysis show that for most workloads, only a few specific rows are repeatedly flagged as hot-rows in consecutive time intervals. In this situation, other possible solutions to mitigate row hammer error include:

- Sharing the information about the hot rows with Last Level Cache (LLC) to avoid evicting cache-lines that are located in hot-rows.
- Using a simple buffer (Cache) in the memory controller to cache hot-rows. Thus, further activations to hot rows are serviced outside the DRAM module which, as a result, prevents the accessed row from being hammered. This also provides an opportunity to improve the performance of memory systems (since the cached

row can be accessed more quickly). Moreover, since these hot-rows only need to be cached for a short period of time (Refresh Interval - e.g. 64 ms) a small buffer would be sufficient to implement this solution.

In either case the memory request to the hot-rows, and as a result the possibility row-hammer occurrence, will be decreased. Additionally, memory latency will be reduced significantly since the highly accessed cache lines now either are cached in the memory controller or kept in the LLC. To evaluate the first (probably more elegant) solution an integrated, detailed memory controller with a full architecture simulator is required which is postponed to future work. However, to get an insight about the possible performance improvement that can be achieved using ARMOR the second solution has been investigated using a simple buffer in the memory controller.

The experimental and analytical results show that the number of hot rows is limited, to few, within specific time intervals (RI). This suggests that a small buffer with the right eviction algorithm would be enough to accommodate all the hot-rows during time intervals. To have an insight into the required size of such a buffer the basic principles described so far will be recapped here. The key point is that in the worst case scenario there are only a few rows that can reach a certain threshold (ACT_{th}) within the refresh interval (RI). Therefore, the maximum buffer size is defined by the maximum number of possible row-aggressors per RI times the size of row (to buffer the entire row). Moreover, reducing the size of this buffer to less than the maximum required size does not affect the accuracy of the ARMOR and it is just a trade off between area overhead and the performance improvement.

6.6 Evaluation Methodology

To investigate the performance of ARMOR, it is compared against the ground truth as well as the solution proposed by Kim *et al.* [KDK+14] (i.e. 'PARA'). The ground truth is calculated by having one counter per DRAM row to keep track of the exact number of activations within RI. All the counters are monitored and checked against the threshold which may corrupt data, ACT_{th}. When one counter reaches ACT_{th} (i.e. hammered rows are detected) ARMOR's hot-rows table is checked to see if the detected row is in the table and if so what is the predicted number of activations.

PARA also is evaluated using a similar methodology. This means that every time that a hot-row is detected using the embedded counters, for each row PARA is checked to see if it has issued any refreshes for the detected hot rows. If it does, it is assumed

that PARA has issued the correct refresh to the victim rows (due to its nature, it actually has only a 50% chance of issuing a refresh to the possible row victim).

The ACT_{th} value used by Kim *et al.* [KDK+14] is used in the experiments ($RI = 64ms$ and ACT_{th} of 139K).

Simulator: USIMM, as introduced in Chapter 3, is used as the main simulation platform in the experiments. Table 6.2 contains the parameters used to configure USIMM. ARMOR and PARA were implemented in USIMM to support direct comparison.

Model	Description	Value
Processor	Clock Speed	3.2 GHz
	Pipeline depth	10
	ROB size	128
Memory System	Bus Speed	800 MHz
	Number of Channels	1
	Ranks per channel	1
	Bank per rank	8
	Row per bank	65,536
	Cache lines per row	128
	Cache line size	64 Byte

Table 6.2: USIMM configuration parameters used in the experiments.

Workloads: To have an extensive evaluation several workloads are used from different benchmark suites as presented in Table 6.3:

- **Standard Benchmarks:** As described in Chapter 3, the standard benchmarks include a wide range of memory intensive applications (i.e. 48 workloads) from different benchmark suites (PARSEC [BKSL08], SPEC [Dix91], BIOBENCH [AJW+05], HPC and COMMERCIAL) and representative regions of interest for each application. To recap, Table 6.3 lists the workloads and their corresponding benchmark suites. An identifier is added to each application to facilitate the naming of multithread workloads constructed from these applications later.

- **Google's Project Zero code/attack:** To evaluate ARMOR's reliability against attacks similar to the one investigated by Google's Project Zero engineers the memory access pattern produced by this team's c-code is also analysed as well. Two different architectural simulators (Zsim [SK13] and GEM5 [BBB+11])

Benchmark Suites			
SPEC		**PARSEC**	**COMMERCIAL**
(a) GemsFDTD_r	(k) astar_B	(u) canneal	(D1) comm1
(b) bzip2_l	(l) bzip2_t	(v) streamcluster	(D2) comm2
(c) cactusADM_b	(m) gcc_1	(w) blackschols	(D3) comm3
(d) gcc_2	(n) gcc_c	(x) facesim	(D4) comm4
(e) gcc_cp	(o) gcc_g	(y) ferret	(D5) comm5
(f) gcc_sc	(p) mcf_r	(z) fluidanimate	**BIOBENCH**
(g) milc_s	(q) omnetpp_o	(A) freqmine	(E) mummer
(h) soplex_r	(r) sphinx3_a	(B) swaption	(F) tigr
(i) xalancbmk_r	(s) zeusmp_z	**HPC**	
(j) libquantum	(t) leslie	(C) hpc1 - hpc13	

Table 6.3: Evaluated workloads and benchmark suites.

were used to run this program and capture the memory access traces. The experimental results show that this code produces an intensive activation stream to random rows within a bank with a uniform distribution. This is used as a motivation to produce more synthetic kernels with similar types of memory access behaviour to model malicious codes with different access distribution described in the following.

- **Synthetic Kernels:** Since the Row Hammer phenomenon is a security issue in DRAMs, a more intense evaluation was carried out considering hand-crafted malicious codes. Around 500M instructions of different memory-intensive synthetic kernels (malicious code) were simulated and the corresponding memory traces were captured. Table 6.4 presents the configuration parameters of the synthetic kernel generator, described in Chapter 3, used to produce the synthetic kernels presented in this chapter. Using these kernels provides full control on modelling row-hammer faults (e.g. by producing hot rows) on a specific number of rows with specific access patterns. In each kernel, a specific number of rows (e.g. up to 20 rows) are targeted and repeatedly activated during execution time, more than other rows in the system. Three different random distributions are used, Uniform, Gaussian and Poisson, to access to these targeted (hot) rows. The main goal of such access patterns is to produce row hammer episodes with different memory access behaviours. The uniform distribution represents a variation of Google's Project Zero attack. Table 6.5 depicts the 36 synthetic kernels with different access distribution.

Synthetic Kernel Generator Parameters	
Parameters	**Value**
DRAM Configuration	1 Channel-1 Rank - 8 Banks - 65K Rows
Memory Trace Size	10 Million Accesses
Maximum Access Interval	1000 Clock Cycles
Memory Intensity	High (10% of Maximum Access Interval)
Read/Write Intensity	100% Read
Access Pattern	Hybrid (switch between sequential and random)
Application Phase Period	MIN=100 & MAX=1000 clock cycles
Number of Targeted Rows	Up to 20 rows
Distribution Access Pattern	Uniform, Gaussian and Poisson

Table 6.4: Synthetic kernel generator configuration parameters used to produce the synthetic kernels.

Benchmark Suites	
Distribution Access Pattern	**Synthetic Kernels**
Uniform Distribution	12 kernels – (UD1 - UD12)
Gaussian Distribution	12 kernels – (GD1 - GD12)
Poisson Distribution	12 kernels – (PD1 - PD12)

Table 6.5: Synthetic workloads.

- **Multithread Kernels:** To increase the randomness of memory access patterns 20 multithread workloads were investigated and the USIMM simulator was modified to map all the workloads to the same memory space. This increases the randomness of memory behaviour and it might also increase the vulnerability of system to the row hammer phenomenon if multiple workloads try to access to different rows within the same bank. To make it even more challenging, malicious kernels were also integrated into multithread workloads. Table 6.6 presents the multithread workloads sequence.

6.7 Results and Discussions

In this section, initially ARMOR is evaluated using standard benchmarks to investigate its performance in average case scenarios. However, since Row Hammer is a security issue, it is also evaluated against some hand-crafted malicious kernels.

Multithread Workloads	
MIX1: (y-PD10-r-UD11)	MIX11: (GD8-C9-C6-y-PD10-h-r-UD11)
MIX2: (GD8-C9-C6-h)	MIX12: (D2-PD3-D2-c-UD11-UD1-UD5-UD6)
MIX3: (D2-PD3-UD1-UD5)	MIX13: (E-D4-C10-x-y-PD1-m-d)
MIX4: (D2-c-UD11-UD6)	MIX14: (GD3-GD4-GD5-GD8-w-z-A-UD9)
MIX5: (E-C10-y-PD1)	MIX15: (GD4-C7-x-PD10-PD6-t-UD12-UD2)
MIX6: (D4-x-m-d)	MIX16: (E-GD2-C13-PD5-g-h-UD10-UD9)
MIX7: (GD3-GD4-w-A)	MIX17: (GD6-y-GD6-k-t-j-q-UD12)
MIX8: (GD5-GD8-z-UD9)	MIX18: (D2-D5-C3-C5-A-B-g-j)
MIX9: (GD4-x-PD10-PD6)	MIX19: (GD1-x-y-PD8-l-a-g-UD7)
MIX10: (C7-t-UD12-UD2)	MIX20: (C2-C4-PD6-n-g-UD10-UD3-UD4)

Table 6.6: Randomly generated Multithread workloads.

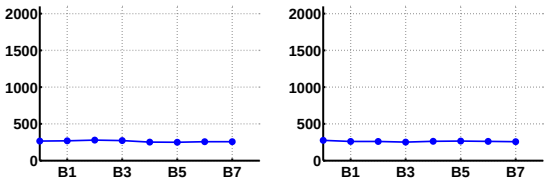
6.7.1 Benchmark Profiling

Standard Benchmark Profiling

Considering the definition of row hammer, it has a strong correlation with the activation interval in DRAMs. This means that lowering the activation interval increases the possibility of row hammer occurrence. Therefore, the benchmarks are profiled to investigate the activation patterns for different workloads. Figure 6.7 to Figure 6.11 present the Average Activation Intervals (AAI) for individual banks in the system. The Y-axis of all the graphs in these figures shows the AAI in *nanoseconds (ns)* and X-axis presents the bank IDs. These results show that there is a significant variation in AAI across different workloads (from around 100 ns for 'hpc6' to 2000 ns for 'stream'). These figures also depict that the memory accesses are distributed uniformly across different banks.

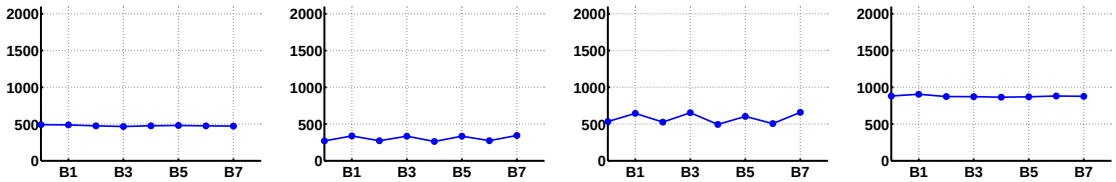
Synthetic Kernel Profiling

In this section, the 36 synthetic kernels, as discussed in Section 6.6 (12 kernels with Uniform distribution, 12 kernels with Gaussian distribution and 12 kernels with Poisson distribution), are profiled. Figure 6.12 presents the number of unique row-aggressors (some of rows might be flagged as row-aggressors several times within a RI) when increasing the number of targeted rows. This figure shows that, for a uniform access pattern, as the number of targeted rows increases the induced row-aggressors also increase linearly. However, when reaching a certain point (in this example, 10 targeted rows and assuming $ACT_{\text{th}} = 139K$) having a uniform access pattern causes none of

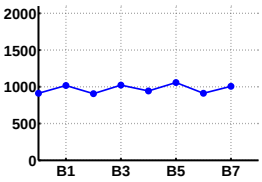


(a) mummer (b) tigr

Figure 6.7: Average activation intervals to each bank for BIOBENCH benchmark suite.



(a) Comm1 (b) Comm2 (c) Comm3 (d) Comm4



(e) Comm5

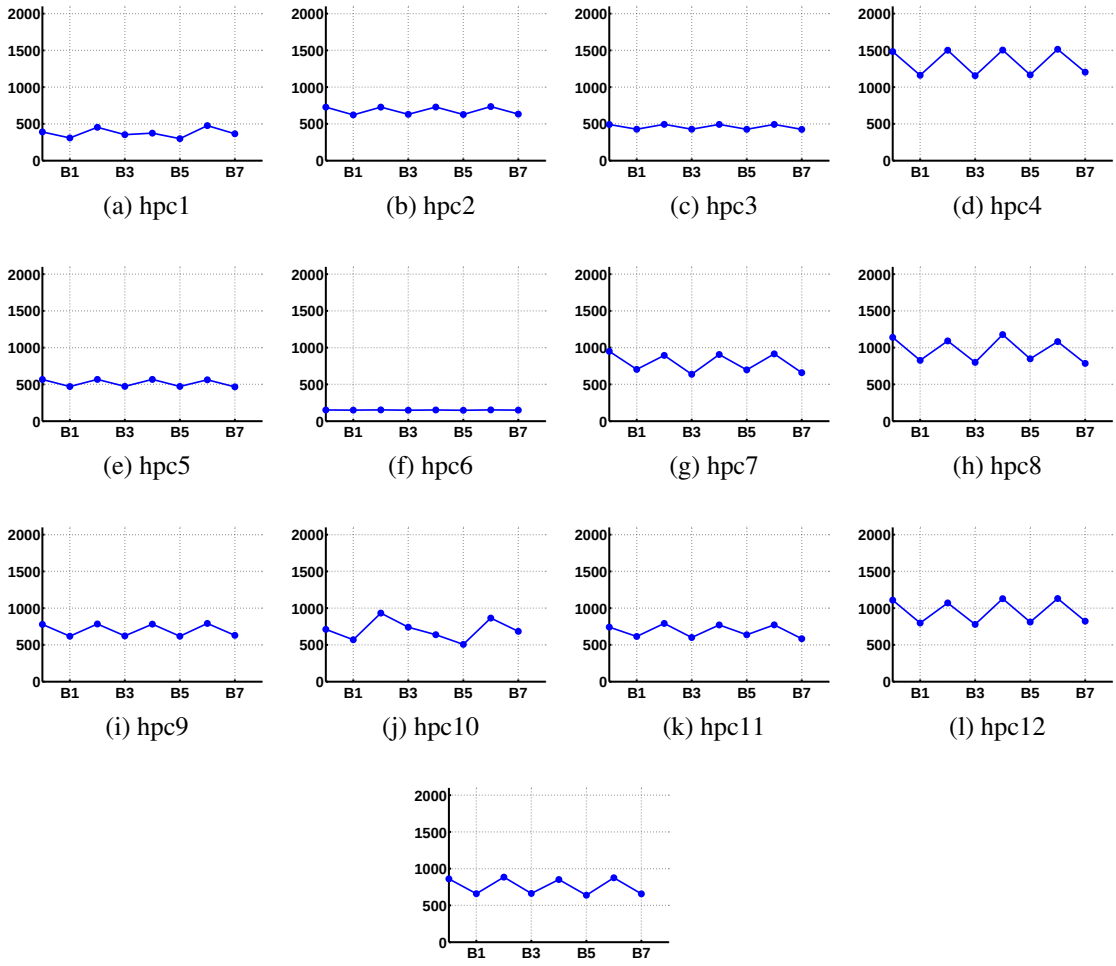
Figure 6.8: Average activation intervals to each bank for COMMERCIAL benchmark suite.

the targeted rows to reach the ACT_{th}. On the other hand, since Gaussian and Poisson distributions favour some targeted rows more than others there are still induced row-aggressors in the system even as the number of targeted rows increases.

Note that each row-aggressor can reach ACT_{th} multiple times within one RI period and, as a result, produce multiple row-hammer problems. Figure 6.13 presents the total row-hammer occurrences during the entire simulation time (i.e. two consecutive RI) using different random distributions.

Multithread Kernels Profiling

Figure 6.14 presents the average activation intervals to individual banks in the system when considering the multithread applications. This figure shows that the activation interval is significantly reduced in comparison with the single-thread workloads (presented in Figure 6.7 to Figure 6.11).



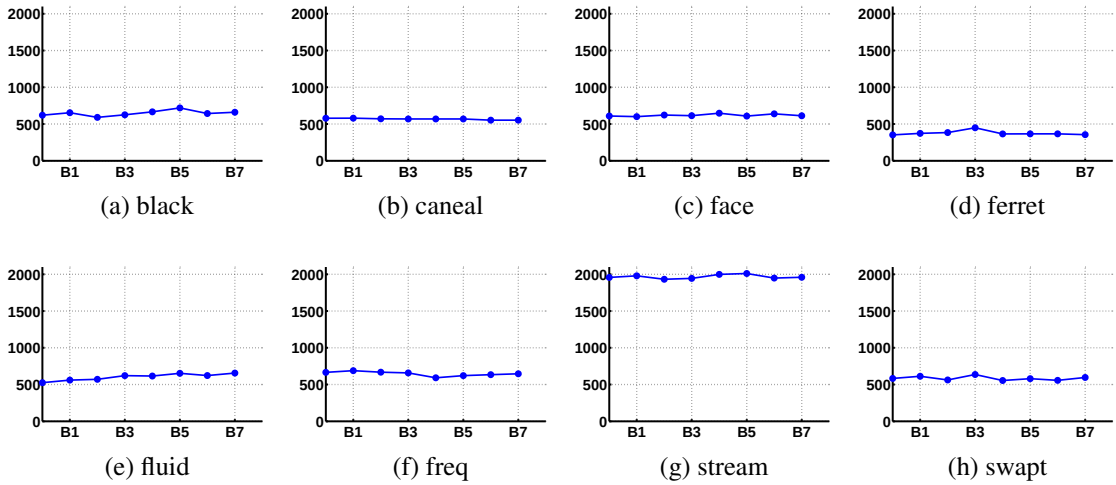
(a) hpc1 (b) hpc2 (c) hpc3 (d) hpc4

(e) hpc5 (f) hpc6 (g) hpc7 (h) hpc8

(i) hpc9 (j) hpc10 (k) hpc11 (l) hpc12

(m) hpc13

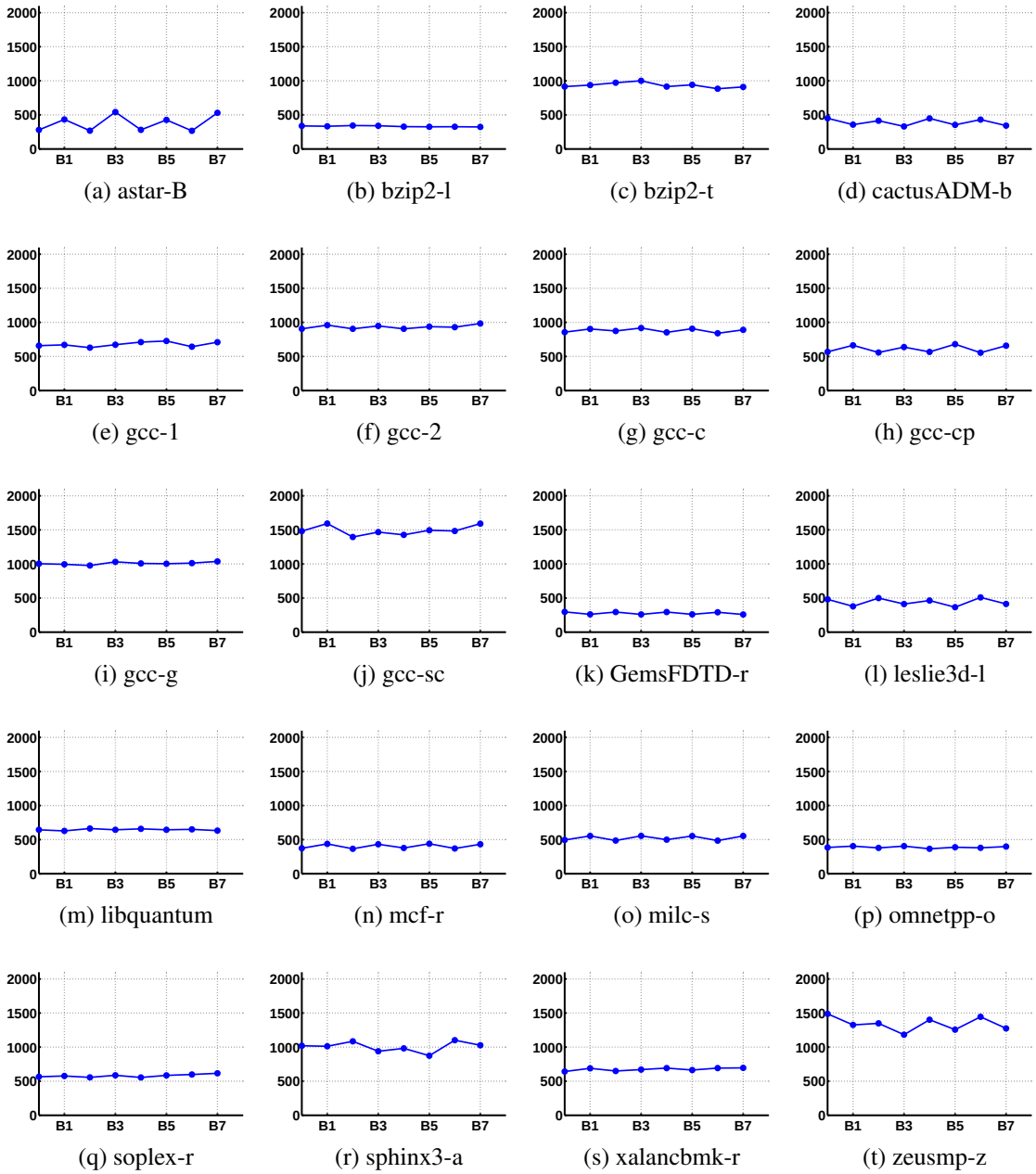
Figure 6.9: Average activation intervals to each bank for HPC benchmarks.



(a) black (b) caneal (c) face (d) ferret

(e) fluid (f) freq (g) stream (h) swapt

Figure 6.10: Average activation intervals to each bank for PARSEC benchmark suite.



(a) astar-B (b) bzip2-l (c) bzip2-t (d) cactusADM-b

(e) gcc-1 (f) gcc-2 (g) gcc-c (h) gcc-cp

(i) gcc-g (j) gcc-sc (k) GemsFDTD-r (l) leslie3d-l

(m) libquantum (n) mcf-r (o) milc-s (p) omnetpp-o

(q) soplex-r (r) sphinx3-a (s) xalancbmk-r (t) zeusmp-z

Figure 6.11: Average activation intervals to each bank for SPEC benchmark suite.

As discussed, reducing the AAI will increase the possibility of row hammer occurrence, albeit it does not guarantee that it will happen. The reason is that the AAI is the average interval value and it does not necessarily imply that a burst of activations has been issued to a specific row. The experimental results show that 8 mixes out of 20 multithread mixes evaluated (MIX2, MIX3, MIX5, MIX7, MIX10, MIX13, MIX16 and MIX19) manifest a row hammer error during execution time.

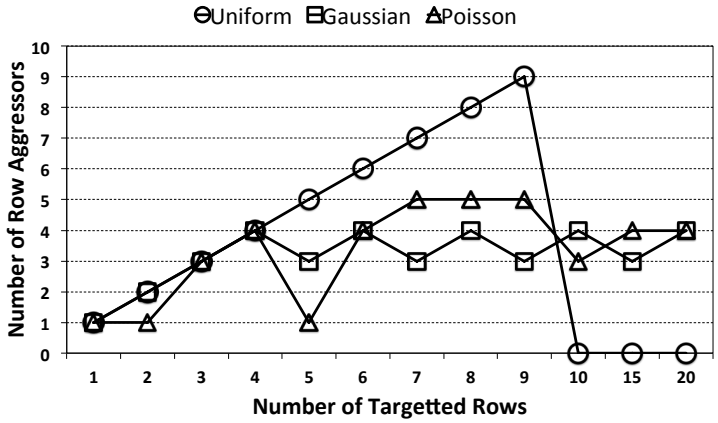


Figure 6.12: Induced unique number of row-aggressors.

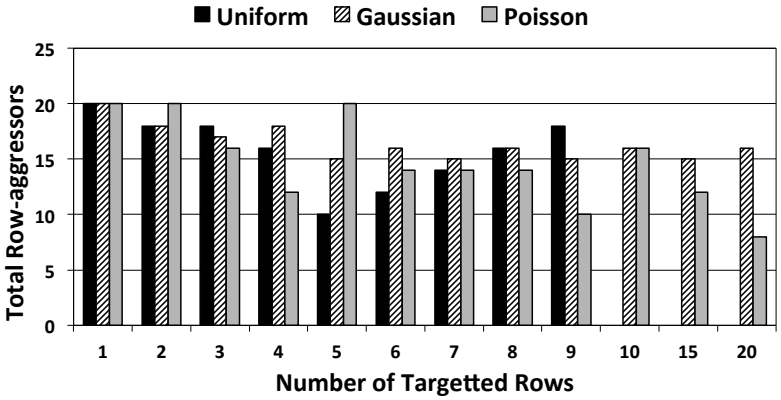


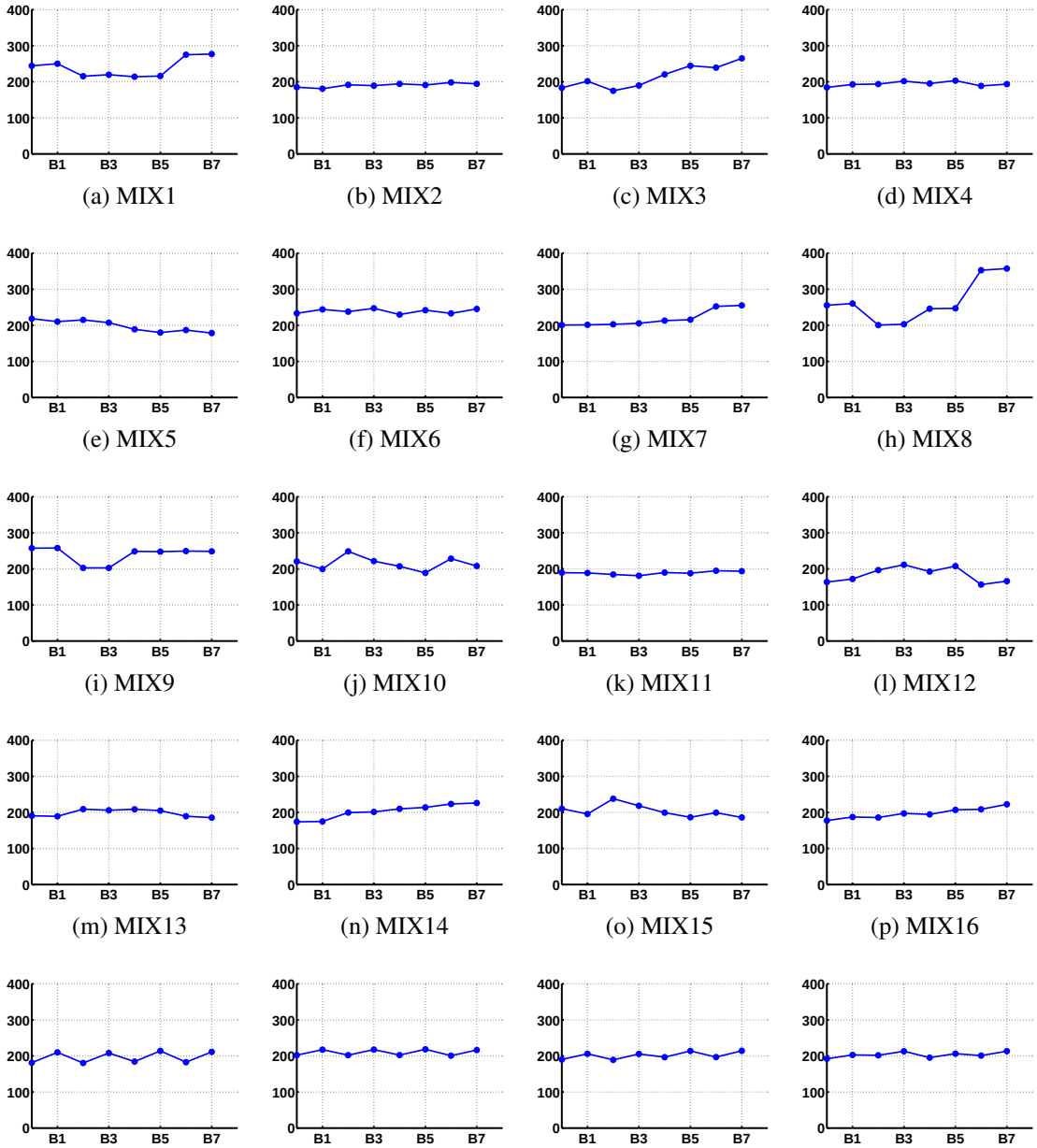
Figure 6.13: Total row aggressors during execution time.

6.7.2 Performance Analysis

Performance Analysis - Single-thread Standard Workloads

Figure 6.15 depicts the performance overhead of ARMOR and PARA, with probability values of 0.001 and 0.005 as proposed in [KDK+14], for standard benchmarks. However, experimental results have shown that PARA cannot protect DRAM from malicious attacks, this is demonstrated in the following section, using the probability value suggested by [KDK+14]. Therefore, the performance of PARA with higher probability values also was investigated and the result is presented in Figure 6.16.

Figure 6.15 and Figure 6.16 show that ARMOR has no performance overhead for standard benchmarks (since there is no row hammer error). On the other hand, although performance degradation of PARA is negligible (maximum around 1%-2%) when using small probability values it can degrade the performance up to 35% if using



(a) MIX1 (b) MIX2 (c) MIX3 (d) MIX4

(e) MIX5 (f) MIX6 (g) MIX7 (h) MIX8

(i) MIX9 (j) MIX10 (k) MIX11 (l) MIX12

(m) MIX13 (n) MIX14 (o) MIX15 (p) MIX16

(q) MIX17 (r) MIX18 (s) MIX19 (t) MIX20

Figure 6.14: Average activation intervals to each bank for HPC benchmarks.

probability values that can protect the DRAM against malicious code (i.e. P=0.2).

Performance Analysis - Single-thread Malicious Code

Although PARA delivers an acceptable performance overhead, due to its probabilistic nature it cannot *guarantee* preventing row hammer. On the other hand, ARMOR

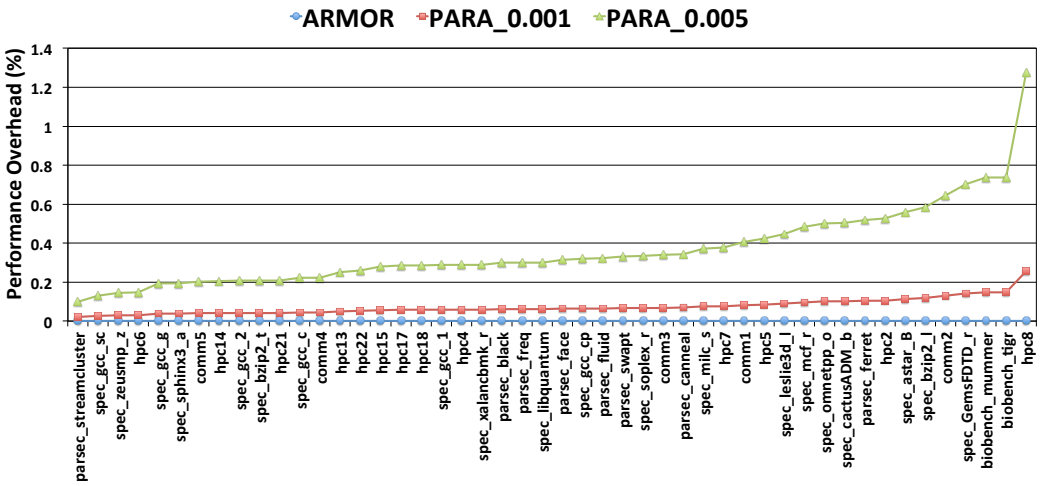


Figure 6.15: Performance overhead of ARMOR and PARA (with the suggested probability values by [KDK+14]) for standard workloads.

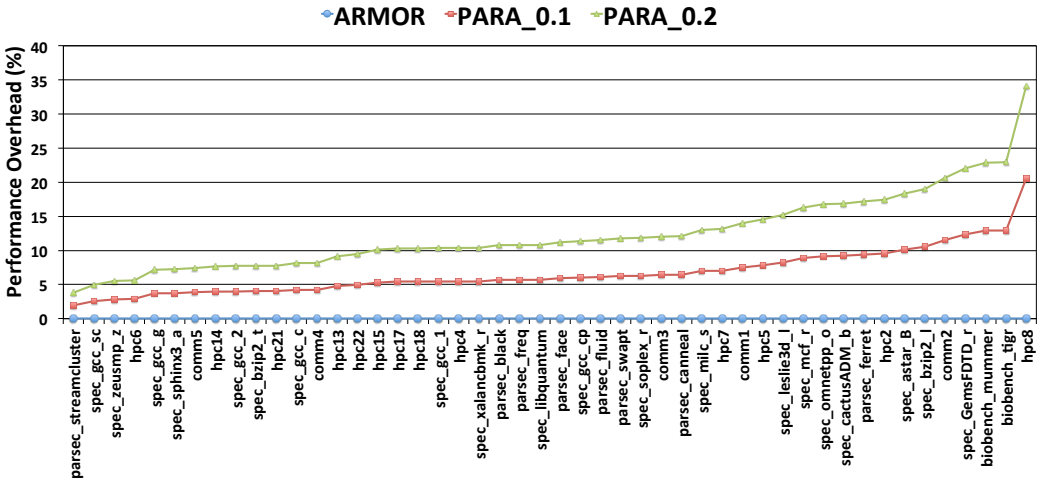


Figure 6.16: Performance overhead of ARMOR and PARA (with the higher Probability Values) for standard workloads.

offers a more robust solution to prevent this phenomenon, with a much lower performance overhead; it also imposes a small area overhead on the system. To investigate this more closely ARMOR and PARA both have been evaluated against the synthetic kernels. Figure 6.17 shows the performance overhead of ARMOR when it detects all possible row-aggressors in the system and refreshes both their adjacent rows. On average ARMOR imposes 0.0030% performance overhead across all the 36 synthetic kernels. According to this figure ARMOR has a higher performance overhead for workloads with a Poisson distribution when the number of targeted rows is equal to 2 or 5. The reason is that the performance overhead of ARMOR depends on the existing

number of row-hammer problems in the system and, according to figure 6.13, kernels with Poisson distribution induces more row hammer problems than the other situation (20 row hammer errors) when the number of targeted rows is 2 or 5.

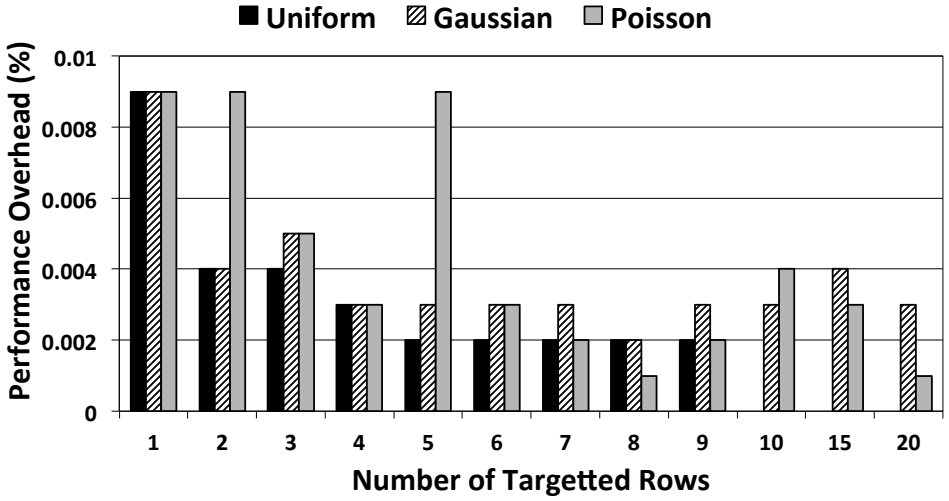


Figure 6.17: ARMOR overhead for synthetic kernels with various access distribution.

Similar experiments were run to evaluate PARA's performance for the synthetic kernels. The recommended probability values [KDK+14] (i.e. P=0.001 and P=0.005) are used in the experiment and these values were increased until PARA detected (refresh by probability) all the possible row-aggressors in the system. Figure 6.18 presents the performance overhead of PARA for different probability value. For P equal to 0.2 PARA has a miss-rate of 3.2%, 0.6% and 0.6% for synthetic kernels with Uniform, Gaussian and Poisson distributions respectively. Figure 6.19 depicts the PARA's miss-rate for different probability values for synthetic kernels.

Performance Analysis - Multi-thread Mixture of Standard Workloads and Malicious code

Figure 6.20 depicts the performance overhead of ARMOR and PARA while running multithread workload mixes. The total execution time of these multithread workloads covers up to 34 refresh cycle providing a long enough time period to induce row hammer error. According to this experiment ARMOR could successfully detect all the hot rows in the system and refresh the possible row victims with the average performance overhead of 2.02×10^{-5} % (up to 9×10^{-5} %) across all the 20 multithread workload mixes. On the other hand, whilst PARA could also randomly detect the row-aggressors,

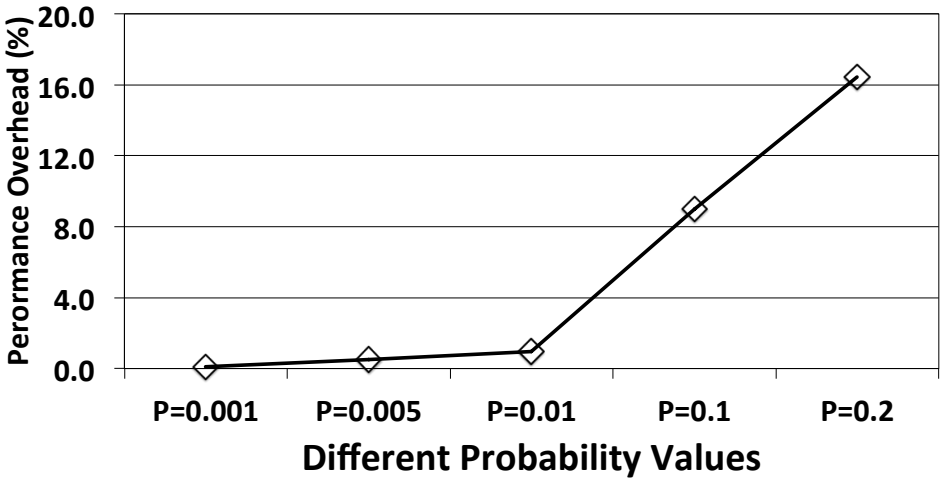


Figure 6.18: PARA overhead for synthetic kernels.

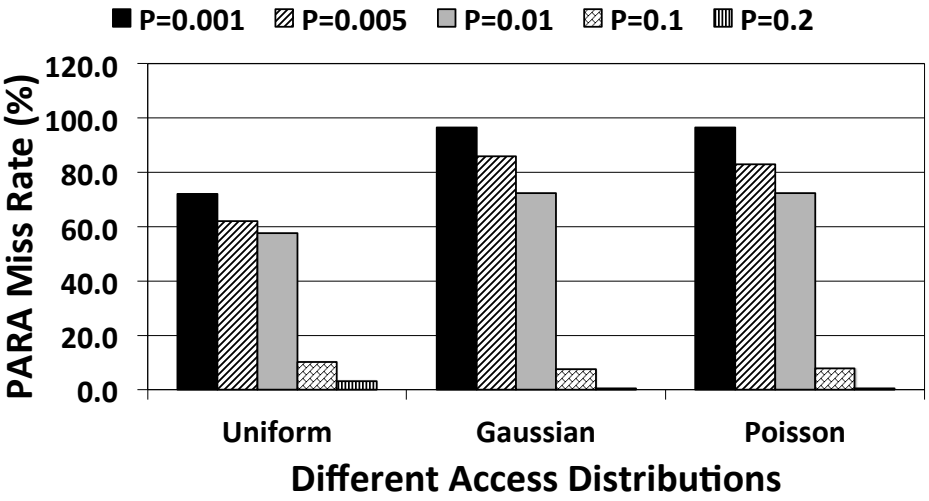


Figure 6.19: PARA miss-rate for synthetic kernels.

by increasing the probability value of PARA to 0.2, this gives an average performance overhead of 10 % (up to 15%).

6.7.3 Hot-Row Table Size vs Accuracy

In section 6.4 it was discussed that, theoretically, the maximum number of hot row table entries should be equal to the maximum possible hot rows that can exist within RI to guarantee that ARMOR detects all the row-aggressors. However, in practice, there is a very low probability that the maximum possible hot rows will occur within RI.

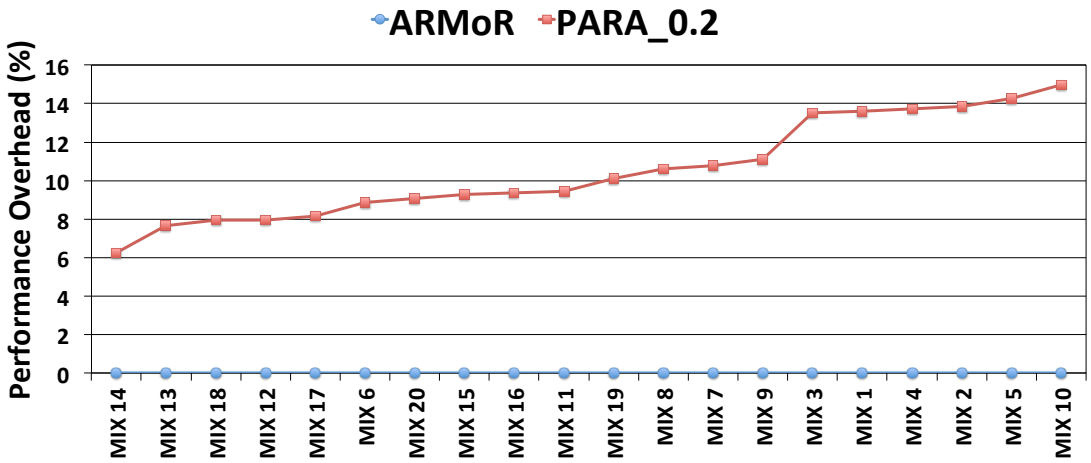


Figure 6.20: Performance overhead of ARMOR and PARA for multithreaded workload mixes.

To investigate this an experiment was carried out for the 36 synthetic kernels and the results show that overall 81% of kernels require fewer than 5 entries, 14% of kernels require 5 to 7 entries and 5% of kernels require 8 to 9 hot row table entries per bank for ARMOR to detect all the possible row-hammer errors in the system (Figure 6.21).

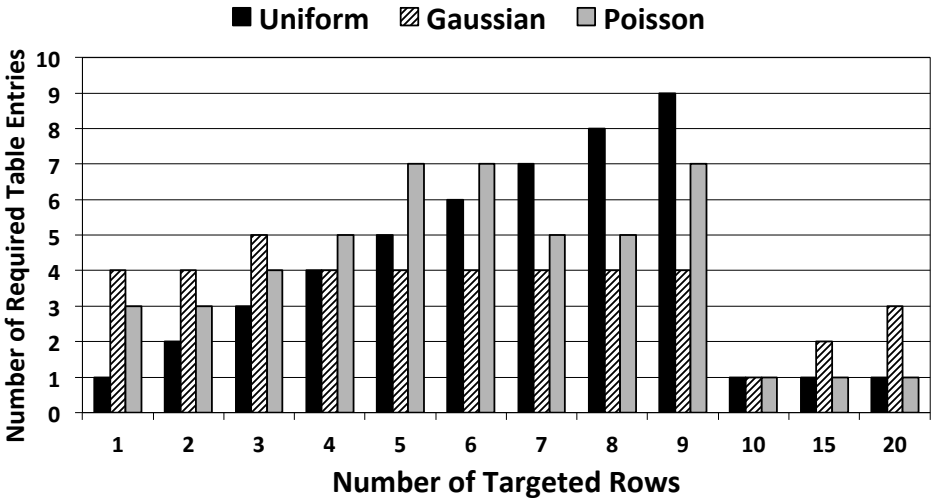


Figure 6.21: The required number of table entries to detect the possible row-hammer errors.

6.7.4 Scalability

To investigate the scalability of ARMOR the maximum imposed storage overhead of the proposal was profiled against various memory organisation populated with different memory sizes. To this end, the standard memory capacities and configuration from Micron's standard DDR3 memory modules [Micb, Micc] are used in this evaluation. Figure 6.22 presents the maximum required storage overhead for ARMOR to detect the maximum hot rows, considering the evaluated ACT_{th} of 139K, 155K and 284K [KDK+14], for a wide range of memory capacities. However, as discussed, this is the maximum theoretical requirement and, the experiment in the last section suggests, five entries per bank would be enough to cover possible row-hammer errors in the system. The corresponding storage overhead for Module A in Figure 6.22 can be considered to model the ARMOR implementation overhead with only five entries per bank.

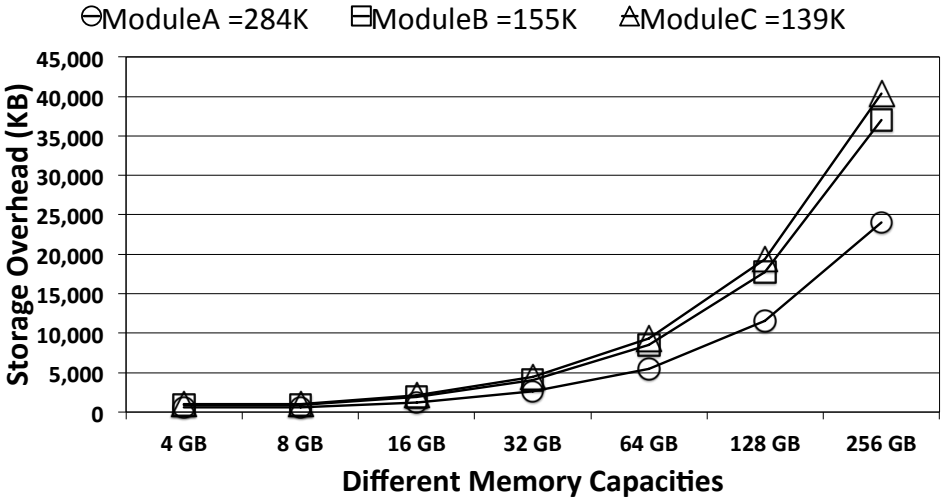


Figure 6.22: Storage overhead for different memory capacities.

6.7.5 ARMOR Cache Performance

To investigate the possible performance improvement that ARMOR can offer using a simple buffer structure, two different scenarios are considered at a high level of abstraction. In the first scenario, it is assumed that as soon as a row is flagged as 'hot' the entire row will be moved to an embedded buffer in the memory controller. There will be a performance overhead to copy the row but this scenario can still present an insight about the upper-bound improvement that can be achieved using this buffer. Figure 6.23 presents the improvement in execution time for all the synthetic kernels.

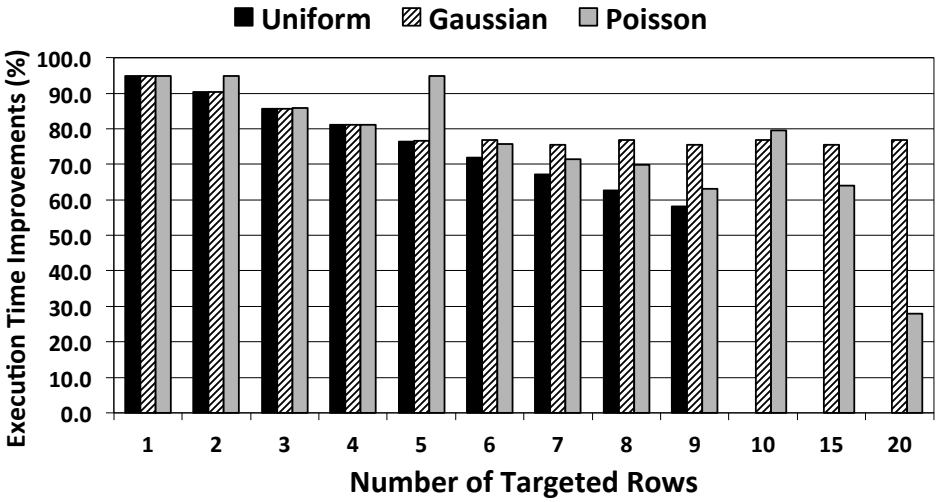


Figure 6.23: Execution time improvements considering buffering entire row.

In the second scenario, it is assumed that as soon as a row is flagged as 'hot' then only the cache-lines holding parts of this hot-row will be moved to an embedded buffer in the memory controller after they have been accessed. Figure 6.24 depicts the gained improvement in execution time for the synthetic kernels.

In both scenarios, the performance improvement that can be achieved has a strong correlation with the number of memory requests that can be serviced using ARMOR's buffer. The evaluated kernels are simulated for only *two* consecutive refresh intervals which means that only a few hot rows which are flagged in the first refresh interval are effective for improving the execution time.

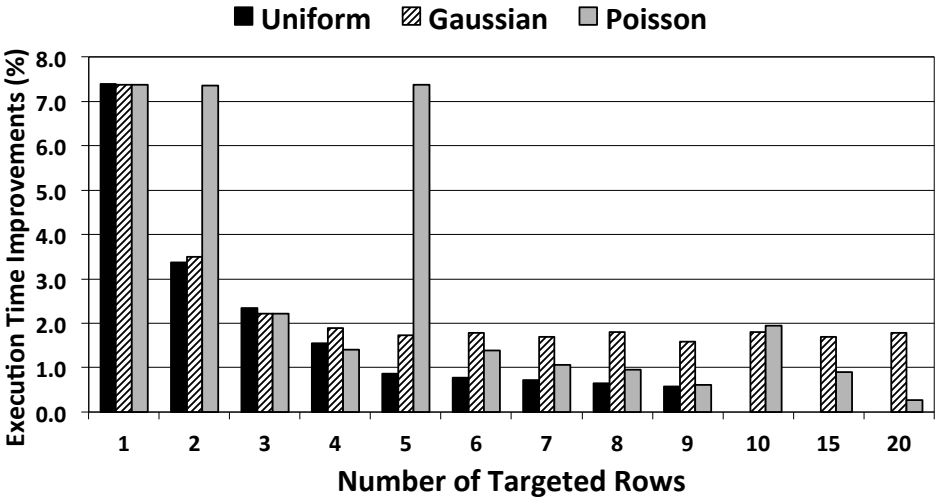


Figure 6.24: Execution time improvements considering buffering cache lines.

6.8 Related Work

In general, the DRAM fault and reliability problem has been studied [CMBR05, GCP+09, KCA+11, SL12]. However, more specifically, the row hammer problem has not been widely acknowledged in literature except by two recent works.

The Row-Hammer phenomenon was initially highlighted by a test equipment manufacturer, Teledyne LeCroy [Mik], at MemCon 2013 [Mica], specifically in the context of DDR4 DRAMs. They designed and developed a monitoring platform that could profile the internal operation of DRAMs (e.g. such as the number of activations per RI) at run time. This equipment can investigate the existence of the row-hammer issue based on programmable parameters (e.g. ACT_{th}).

At the time of writing, the most comprehensive academic work that investigated the existence of row-hammer issue in commodity DRAM devices is the paper by Kim *et al.* [KDK+14]. In this work, 129 DRAM modules from three different major DRAM manufacturers have been intensively evaluated for the row-hammer phenomenon using eight FPGA platforms. They showed that 110 out of 129 DRAM modules suffer from disturbance error (i.e. Row Hammer). Several possible solutions have been investigated in this work and they proposed PARA (Probabilistic Adjacent Row Activation) to overcome row hammer. The key idea behind PARA is that every time a row is opened and closed one of its adjacent rows is refreshed (e.g. opened) with some low probability. Thus, if a particular row is opened and closed repeatedly then, statistically, the adjacent rows will be refreshed.

The main advantages of ARMOR over PARA are as follows:

- First, PARA cannot *guarantee* detecting all possible row victims, since it is based on probabilistic estimation. Instead it significantly reduces the probability of row-hammer. However, since ARMOR can detect the exact row-aggressor the final result is much more accurate.

- Although, PARA has a very low overhead it still imposes some extra penalty by unnecessarily refreshing rows. Again, since ARMOR predicts the exact row-aggressor only the possible row-victims will be refreshed.

- ARMOR only refreshes adjacent rows when needed, thus reducing energy.

- Finally, having knowledge of exact row-aggressors' IDs offered by ARMOR provide an opportunity to improve the DRAM's performance as well as improve the reliability by overcoming the row-hammer phenomenon.

Outside the scientific publications, Intel has two patents [GBS+14, BH14] with an approach close to ARMOR. Note that ARMOR was developed independently without checking or searching the patent literature. The patents are based on the observation that there is a maximum number of rows, which is much smaller than the total number DRAM rows, that can reach the activation threshold. Intel proposed a single table with the number of entries fixed to accommodate the maximum possible row aggressors within a refresh interval. Then, for every activation command if the activated address is in the table then a counter to monitor the number of activations to that row will be incremented. If the activated address is not in the table, then the row ID will be inserted into the table. Thus the table is kept updated with each activation and only counts for row IDs stored. In addition, the table is kept sorted (e.g. high to low) according to the number of activations. When the table is full and a row not stored in the table is activated, the least activated entry in the table will be replaced with the newly activated address. However, the new row ID will still keep the corresponding number of activations for the replaced row ID, instead of resetting it. This attempts to retain the number of activations for the replaced row ID, if in the future the same address is activated again. On the other hand, this can over estimate the number of activations for rows leading to unnecessary refreshes. Albeit the patents can alleviate the row hammer problem there are key differences when compared with ARMOR. The patents do not propose the initial filtering phase offered by ARMOR (TSRF). Hence, instead of monitoring only the potential hot-rows, the patents monitor all the activation streams and remember only a subset of rows in a table. As there is not enough space in the table, rows (with smallest number of activations) are replaced from time to time. This introduces an error in the activation count that can cause a significant number of false alarms, and as a result imposes unnecessary refreshes to the system.

6.9 Summary

This chapter proposed a novel hardware technique called ARMOR which is capable of preventing data corruption in DRAMs due to the row hammer effect. Uniquely ARMOR is able to guarantee that the row-hammer effect cannot modify the contents of DRAM whilst having only modest hardware requirements. Given the datasheet parameters of DDR3 and DDR4 together with the minimum number of activations within a refresh interval required to trigger a row hammer effect, it has been shown how to derive mathematically the maximum number of rows that could be affected. This

number is small, no more than 10 per bank. ARMOR detects hammered-rows (rows with aggressive activation) at run time and either refreshes their adjacent rows or moves the hot rows to a local buffer in the memory controller to improve the performance of memory system.

ARMOR has been evaluated and compared directly against PARA [KDK+14] over 48 standard workloads, 36 synthetic memory intensive kernels (some of them representing the Google's Project Zero attack) with three different random access distributions (i.e. Uniform, Gaussian and Poisson) and 20 multithread workload mixes. Out of the 48 standard workloads, none exhibit an access pattern that would lead to row hammer data corruption. In this case, ARMOR delivers no execution time overhead while PARA would degrade the performance up to 35%. Using the synthetic kernels, it has been shown that PARA could miss row hammer episodes (up to 90%) while ARMOR detects all the row hammer errors with only 0.0030% performance overhead. The miss-rate of PARA is reduced to 0.6% by increasing its probability value to 0.2. In this case PARA imposes an average performance degradation of 16%. Considering the multithread mixes ARMOR prevents all the row hammer episodes with an average performance overhead of 2.02×10^{-5} % (up to 9×10^{-5} %) across all the 20 mixes. On the other hand, whilst PARA could also detect the row-aggressors, by increasing its probability value to 0.2, this added an average performance overhead of 10 % (up to 15%).

Overall, the experimental results show that ARMOR does not affect the execution time of the workloads while detecting all the possible row hammer errors in the memory system. Moreover, it has been shown that by using a small buffer in the memory controller, ARMOR provides an opportunity to improve the overall execution time of synthetic kernels by up to 7%.

Chapter 7

A Workload Adaptive Memory Controller

7.1 Introduction

So far, this thesis has presented three different techniques (i.e HAPPY, DReAM and ARMOR) to improve the performance of the three main components of the state-of-the-art memory controllers. To summarise, Chapter 4 (i.e. HAPPY) discussed how to reduce the implementation cost of dynamic page closure policy predictors, and improve their scalability, while maintaining the prediction accuracy. Chapter 5 (i.e. DReAM) presented how a workload-specific address mapping scheme can be detected and employed at run time that outperforms the baseline address-mapping schemes and improves the performance of the memory system. Finally, Chapter 6 (i.e. ARMOR) proposed a hardware solution to detect and prevent the row hammer phenomenon in DRAMs, improving the reliability of the memory system. However, each of these techniques was evaluated individually in the absence of the other techniques. In this chapter, the performance of an adaptive memory controller utilising all three of the proposed techniques is investigated. Moreover, this chapter analyses the overall implementation cost of a state-of-the-art memory controller employing these three techniques.

7.2 Evaluation Methodology

This section describes a methodology to investigate how a memory controller can take advantage of HAPPY, DReAM and ARMOR in a unified system.

7.2.1 Baseline vs. Adaptive Memory Controller

The final evaluation of this research considers two different memory controllers. A standard memory controller equipped with HAPPY, DReAM and ARMOR, hereafter called the *'adaptive'* memory controller and a state-of-the-art memory controller configured to use the previous best page closure policy, best address mapping scheme and best existing row hammer protection technique evaluated in this PhD thesis, hereafter called the *'baseline'* memory controller. Figure 7.1 depicts an overview of the adaptive and baseline memory controllers and Table 7.1 presents different configurations for them.

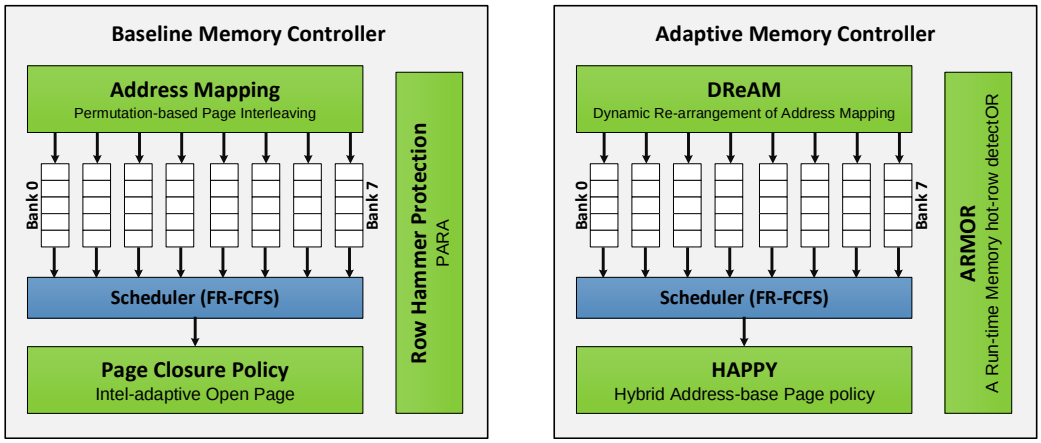


Figure 7.1: An overview of the adaptive and baseline memory controllers.

DReAM is configured to limit the total number of migrations to 5% of the rows (as discussed in Section 5.5.4). Also, the migration threshold of DReAM is set be 7%, as before.

To recap and justify the configuration presented in Table 7.1, Chapter 4 determined that Intel-adaptive delivers the best execution time compared with other page closure policies. Chapter 5 showed that the Permutation address-mapping scheme outperforms other address-mapping schemes. Finally Chapter 6 presented that PARA is the only publicly available solution (at the time of writing) to mitigate row hammer in DRAMs and, based on the experimental results in this thesis, it can detect the row hammer errors

with a probability value equal to 0.2. For this reason the baseline memory controller is configured with these parameters.

Memory Controller	Baseline	Adaptive
Scheduler	FR-FCFS	FR-FCFS
Page Closure Policy	Intel-adaptive	Intel-adaptive-HAPPY
Address Mapping	Permutation	DReAM (Offline - Online)
Row Hammer Protection	PARA (p=0.2)	ARMOR

Table 7.1: Implementation options for standard and adaptive memory controller.

7.3 Results and Discussions

The focus of this section is to analyse the performance and implementation cost of the proposed adaptive memory controller in comparison with an optimised state-of-the-art memory controller, as a baseline.

7.3.1 Performance Analysis

Figure 7.2 presents the performance comparison between the baseline and the adaptive memory controller. The results depicted in this figure are normalised to the execution time of the baseline memory controller. These results show that the adaptive memory controller outperforms the baseline, in all the cases, by 18% (on average) and up to 35% for some workloads.

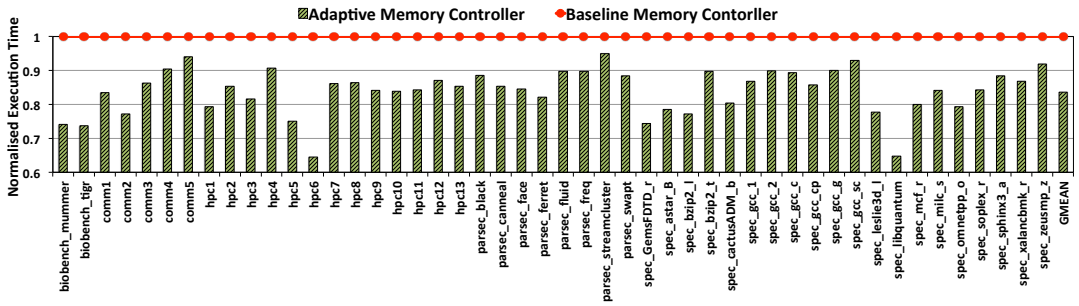


Figure 7.2: Execution time comparison between the adaptive and the baseline memory controller (normalised to the baseline execution time).

A more detailed analysis was carried out to investigate the performance of the baseline and the adaptive memory controller. To this aim, the execution time was profiled based on three main contributors to performance:

- **Normal Execution Time:** This is the time required to execute each workload ignoring the data relocation cost and the imposed penalty by refreshing the victim rows to prevent row hammer effect.
- **Row Hammer Protection Overhead:** This is the extra penalty imposed by refreshing the victim rows to prevent row hammer errors.
- **Data Migration Overhead:** This is the time required to relocate data inside DRAMs imposed by the online data migration.

Figure 7.3 and Figure 7.4 depict the performance results profiled based on these three main factors contributing to the execution time for the baseline and the adaptive memory controller respectively.

As Figure 7.3 presents, since the baseline memory controller employs a fixed address mapping scheme (i.e. Permutation-based page interleaving) then there is no data migration penalty associated with it. On the other hand, as this memory controller takes advantage of PARA (with the probability value of 0.2), to avoid the row hammer errors, then there is in average 13% (up to 36%) performance penalty due to refreshing random rows.

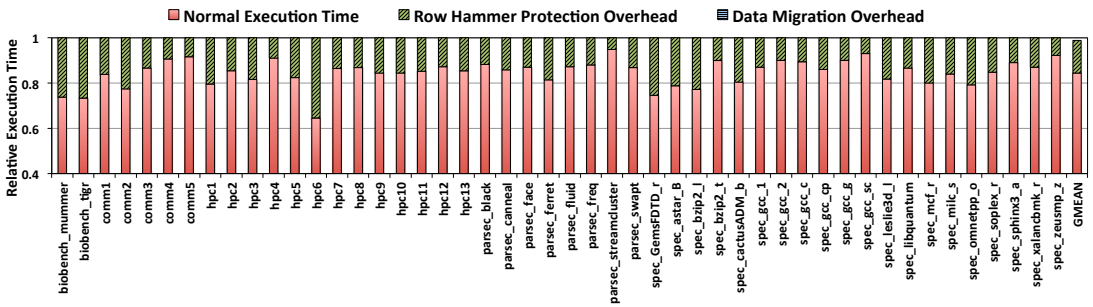


Figure 7.3: The profiled execution time for the baseline memory controller.

Similarly, Figure 7.4 presents the profiled execution time of the adaptive memory controller. In this case, since the adaptive memory controller employs ARMOR to prevent row hammer then there is no performance penalty associated with it. Note, that ARMOR would impose a negligible performance penalty (as discussed in Chapter 6) if there was a row aggressor in the system; Thus, since there are no aggressors detected for the benchmarks evaluated in this chapter then ARMOR imposes no performance penalty. On the other hand, since the adaptive memory controller uses DReAM to detect a workload specific address-mapping scheme at run time, there is an extra cost

associate with the online data migration. However, as discussed in Chapter 5, DReAM employs the predicted address mapping scheme only if it can improve the bit-change rate over a predefined threshold (i.e. 7% in this experiment). This is why only a few applications are paying a penalty for online data migration according to this Figure.

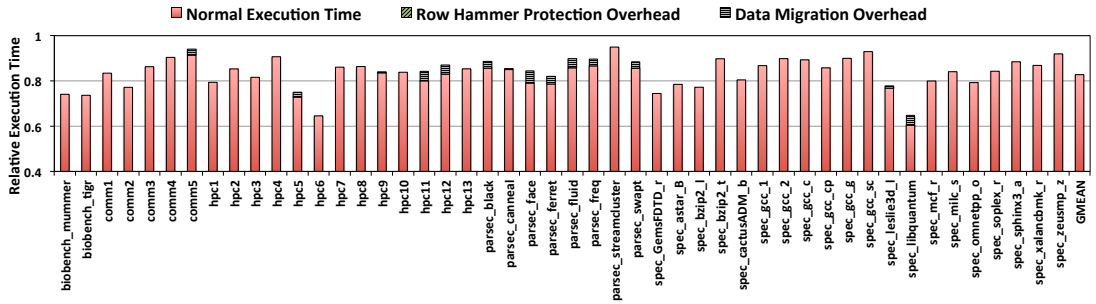


Figure 7.4: The profiled execution time for the adaptive memory controller.

The performance of the page closure policies, Intel-adaptive for the baseline memory controller and Intel-adaptive-HAPPY for the adaptive memory controller, are tightly integrated and manifest themselves in the normal execution time of each workload.

Since one might be interested in integrating the offline version of DReAM (as discussed in Chapter 5) into the adaptive memory controller an experiment was carried out comparing the performance of the baseline memory controller against two versions of the adaptive memory controller: 'Adaptive-Offline', which employs the offline version of DReAM, and 'Adaptive-Online' which employs the online version of DReAM. Figure 7.5 to Figure 7.8 present the results associated with this experiment for all the workloads categorised based on their corresponding benchmark suite.

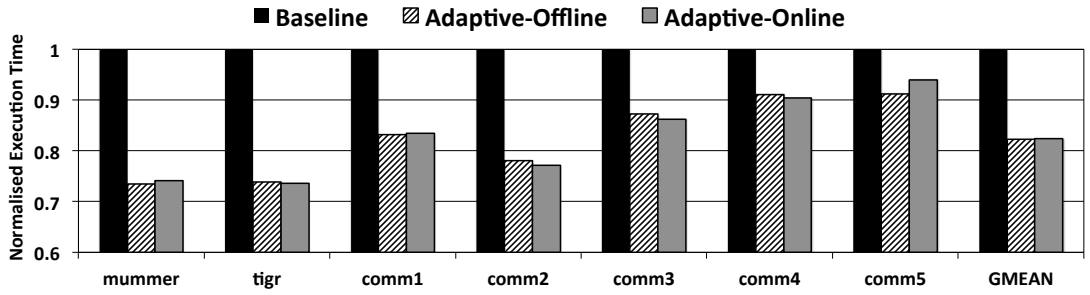


Figure 7.5: Final performance improvement achieved for BIOBENCH and COMMERCIAL benchmark suites.

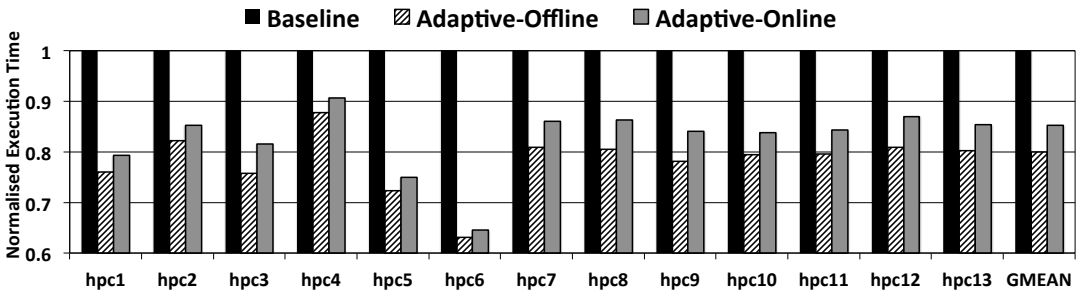


Figure 7.6: Final performance improvement achieved for HPC benchmarks.

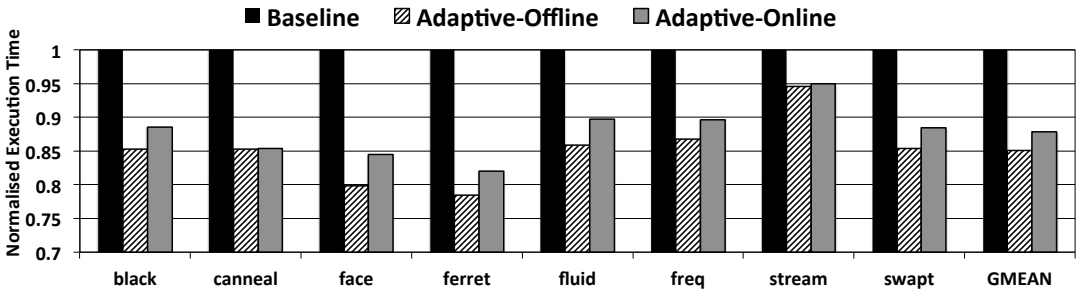


Figure 7.7: Final performance improvement achieved for PARSEC benchmark suite.

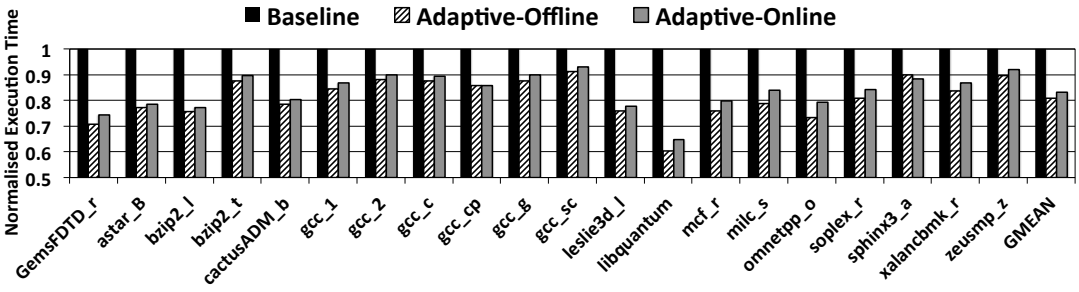


Figure 7.8: Final performance improvement achieved for SPEC benchmark suite.

7.3.2 Implementation Cost Analysis

This section investigates the implementation cost of the adaptive memory controller in comparison with the baseline. The focus of this section is to analyse the differences in the storage overhead imposed by each memory controller implementing the page closure policy, address mapping and row hammer protection units.

Figure 7.9 depicts the storage overhead imposed by the baseline and the adaptive memory controller. Since, DReAM imposes a significantly higher storage overhead than the other two proposals (i.e. HAPPY and ARMOR), the figure also presents two versions of the adaptive memory controller, one version considering the storage

overhead imposed by DReAM and one version ignoring it. This figure also shows the scalability of each memory controller against the different memory capacities.

To recap, the baseline memory controller employs a fixed address mapping scheme and PARA as a protection scheme against the row hammer effect. Therefore, there is no storage overhead above a standard memory controller to implement these two units (as discussed in the corresponding chapters). Thus, the storage overhead for the baseline memory controller in Figure 7.9 is mainly related to the Intel-adaptive page closure policy.

On the other hand, the adaptive memory controller requires additional storage overheads to implement HAPPY, DReAM and ARMOR. The implementation costs of these three proposals were investigated in detail in their corresponding chapters. Figure 7.9 presents the overall storage overhead from these three techniques for the adaptive memory controller. Overall, these results show that a 6 KB to 900 KB storage overhead is required to design an adaptive memory controller to support a wide range of memory capacities (up to 512 GB). However, excluding DReAM (the most expensive part) from the implementation can reduce the storage overhead to around 1 KB to 100 KB.

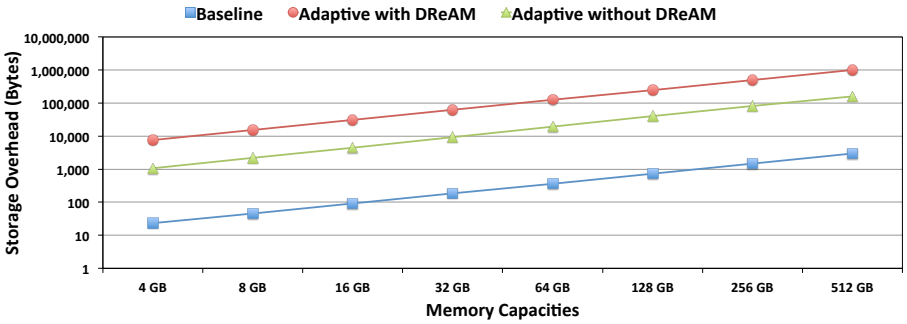


Figure 7.9: Implementation cost comparison between the baseline and the adaptive memory controller.

For a more detailed analysis of the adaptive memory controller's implementation cost Figure 7.10 depicts the profiled storage overhead based on the three proposed techniques. This result shows that HAPPY requires the minimum storage cost of the three proposed techniques with less than 100 bytes across all the memory capacities. DReAM imposes the maximum storage overhead with up to 800 KB for the 512 GB memory. ARMOR imposes a modest storage overhead of 1 KB for the 4 GB and up to 100 KB for the 512 GB memory.

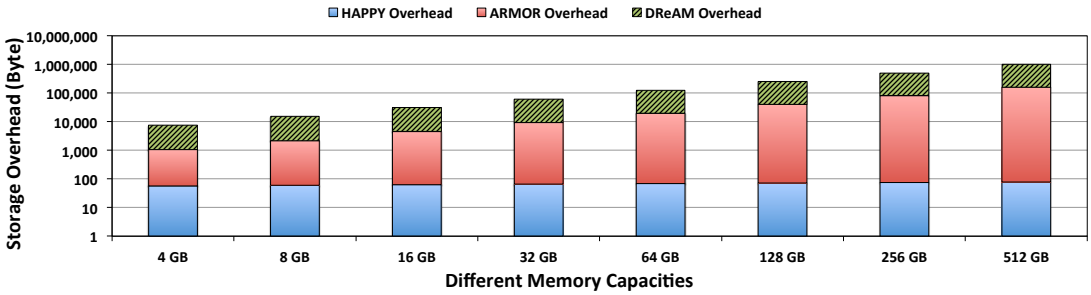


Figure 7.10: The implementation cost of the adaptive memory controller profiled base on the cost of HAPPY, DReAM and ARMOR.

7.4 Summary

This chapter summarised the overall improvement that a memory controller can achieve employing the techniques proposed in this PhD thesis (i.e. HAPPY, DReAM and ARMOR). To this aim the proposed techniques were integrated in a standard memory controller, called the adaptive memory controller, and its performance and implementation cost were compared with an optimised state-of-the-art memory controller, as a baseline.

Overall the experimental results show that the adaptive memory controller outperforms the baseline memory controller, on average, by 18% and up to 35% for some workloads. Moreover, the execution times were profiled based on the three main contributors to the performance. The results show that the baseline memory controller spends around 13% of its time refreshing random rows to avoid the row hammer error. However, since the adaptive memory controller employs ARMOR, it does not pay such a penalty to protect against this phenomenon. Moreover, since the adaptive memory controller includes DReAM it can also gain an extra performance boost for some workloads by taking advantage of a workload-specific address-mapping scheme predicted at run time. However, for the same reason, the adaptive memory controller also pays a penalty to perform the online data migration.

Investigating the implementation cost, the adaptive memory controller imposes more storage overhead to the system than the baseline. Overall, the experimental results show that the adaptive memory controller requires around 6 KB-900 KB more storage than the baseline memory controller to support a wide range of memory sizes. A detailed profiling of the implementation cost showed that DReAM is the most expensive part of the adaptive memory controller with up to 800 KB storage overhead. Therefore, as an implementation option, if a designer decides to exclude DReAM the

overall storage cost of the adaptive memory controller will be around 1 KB-100 KB. Moreover, it has been shown that HAPPY with the 100 Bytes of storage overhead is the cheapest technique amongst the three proposed here. Also, it has been discussed that ARMOR delivers a modest storage overhead of 1 KB-100KB for 4 GB and 512 GB memories respectively.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this PhD research different approaches have been investigated to improve the crucial performance aspects of DRAM controllers such as Performance, Reliability and Scalability. To this end, three key components in a DRAM controller have been recognised and three different techniques have been proposed to improve their performance. In particular this PhD thesis made the following contributions.

DRAM performance depends on the memory access pattern and, more specifically, the number of page-hit and page-conflicts that occur at run time. Modern DRAM controllers employ advanced page closure policy predictors to improve performance trying to transform page-conflicts into page-empty (e.g. by closing the last accessed row at the "right time"), and page-empty cases into page-hits (e.g. by keeping open the last accessed row for longer time). However the main challenge is to balance the prediction accuracy of these predictors with manageable hardware overheads (scalability) as the size of DRAM is increased. HAPPY – a compact and efficient binary-encoding technique – was proposed to alleviate the scalability problem of DRAM page closure predictors. The main intuition behind HAPPY is based on the observation that addresses that are spatially close together tend to have a similar page-closure policy preference. Thus, the physical address bits carry the information that a memory controller requires to predict the page-hit or page-conflict for a particular access. HAPPY is devised to exploit such information by fine-grain monitoring of physical address bits' access pattern. Considering this, the required performance counters and monitoring units needed by the page closure prediction algorithms can be encoded from the physical address bits. Doubling the size of DRAM only implies one extra physical address bit. This means

that with HAPPY only two extra monitoring units are required to predict the DRAM page closure policy when the size of memory is doubled. In other words, HAPPY offers a scalable hardware solution to implement dynamic DRAM page closure predictor algorithms. HAPPY was evaluated by applying it to a traditional Hybrid page closure policy, as well as the state-of-the-art Intel-adaptive open page policy included in Intel Xeon X5650. The experimental results show that the HAPPY implementation of Intel-adaptive page policy can reduce the cost of implementation by 5× for the evaluated 64 GB memory size (up to 40× for a memory size of 512 GB) while maintaining the prediction accuracy. The other case study shows 182,000× reduction in cost of implementation for the evaluated 64 GB memory size (up to 1.2M× for memory size of 512 GB). The experiments have also reported the accuracy of the predictors and have studied the sensitivity towards the memory address-mapping. In both scenarios, HAPPY maintains its key advantage of offering no degradation of prediction accuracy while reducing significantly the hardware overhead.

One of the main contributors to the performance of the memory system is the address-mapping scheme employed to map the physical address bits from processors to the internal structure of the DRAM chip. Thus, the address-mapping scheme controls the data location in the memory system. Due to the 3D structure of DRAMs, accessing different locations of the memory system implies different access latencies. The performance of DRAMs depends on the bit mapping and the memory access pattern of each workload. Usually, memory controllers use a predefined and fixed address-mapping scheme; if the memory access pattern of an application does not fully exploit this the performance of the memory system will be degraded. To mitigate this issue, this thesis introduces DReAM (Dynamic Re-arrangement of Address Mapping) which is a novel hardware technique based on approximating the entropy of each memory address bit for a set of memory requests. DReAM identifies which bits are changing the most (higher entropy) and which ones change the least (lower entropy) and maps the lowest entropy bits such that they occur within a row, hence minimising page conflicts. DReAM presents three main contributions: first, a low-cost pattern recognition technique was developed to extract the memory access pattern at run time. Then, a methodology was proposed to estimate an optimised address-mapping scheme based on the detected access pattern. Finally, a technique is proposed for on-the-fly migration of data within DRAMs, to reduce page conflicts. An extensive performance evaluation was carried out with 48 different workloads from 5 benchmark suites. By keeping the

memory size constant while increasing the number of cores the 'randomness' of the behaviour of the memory access pattern is increased significantly, which might impose a high rate of page conflicts to the memory system. It was shown that, in such a situation DReAM is still able to detect the application access pattern and estimate an optimised address-mapping scheme that improves the performance of the memory systems in comparison with the best evaluated baseline mapping. Overall, DReAM-Offline outperforms the permutation-based address-mapping scheme (the best evaluated baseline) by 5%, on average, and up to 28% across all the workloads. In the case of DReAM-Online, 12 workloads satisfy the DReAM's threshold at run-time (i.e. improve the bit change rate by more than 7%) and for these workloads DReAM-online outperforms the baseline by 4.5%, on average, and up to 23%. DReAM is the first mechanism capable of generating workload specific address-mappings on-the-fly without requiring to stop the running applications.

Reliability of DRAMs is another important factor that has been studied in this PhD thesis. Due to the volatile nature of DRAM technologies, memory controllers need to issue refresh commands at strict time intervals to avoid corruption of stored data. A less well known means of corrupting stored data within a refresh interval is to have a sequence of memory requests causing a DRAM row to be repeatedly activated above certain thresholds; the *row hammer effect*. This PhD thesis proposed a novel hardware technique called ARMOR which is capable of preventing data corruption in DRAMs due to the row hammer effect. It has been shown how to derive, mathematically, the maximum number of rows that could be affected. This number is small; no more than 10 per bank. ARMOR accurately detects hammered-rows (rows with aggressive activation) at run-time which is important when addressing security problems. ARMOR has been evaluated and compared directly against PARA [KDK+14] over 48 standard workloads, 36 synthetic memory intensive kernels (some of them representing Google's Project Zero attack) with three different random access distributions (i.e. Uniform, Gaussian and Poisson) and 20 multithread workload mixes. Out of the 48 standard workloads, none exhibits access patterns that would lead to row hammer data corruption. In this case, ARMOR has no execution time overhead while PARA would degrade the performance by up to 35%. Using the synthetic kernels, it has been shown how PARA could miss row hammer episodes (up to 90%) while ARMOR detects all the row hammer errors, with only 0.0030% performance overhead. Considering the multithread mixes ARMOR prevent all the row hammer episodes with an average performance overhead of 2.02×10^{-5} % (up to 9×10^{-5} %) across all the 20 mixes.

Overall, the experimental results showed that ARMOR does not affect the execution time of the workloads while detecting all the possible row hammer issues in the memory system. Moreover, it has been shown that by using a small buffer in the memory controller, ARMOR provides an opportunity to improve the overall execution time of synthetic kernels by up to 7%. The ARMOR technique is now the basis of a patent application.

As a final evaluation of this PhD thesis, an adaptive memory controller was developed integrating HAPPY, DReAM and ARMOR into a standard memory controller. The performance and the implementation cost of such an adaptive memory controller were compared against a state-of-the-art memory controller, as a baseline. Overall the experimental results show that the adaptive memory controller outperforms the baseline, on average, by 18% and up to 35% for some workloads while requiring around 6 KB-900 KB more storage than the baseline to support memory sizes in the range from 4 GB to 512 GB. A detailed profiling of the implementation cost shows that DReAM is the most expensive part of the adaptive memory controller, with up to 800 KB storage overhead, ARMOR has a modest storage overhead of 1 KB-100 KB for 4 GB and 512 GB memories respectively whilst HAPPY, with 100 bytes of storage overhead, is the cheapest of the three techniques proposed in this thesis.

To conclude, this thesis presented how the performance, scalability and reliability of memory controllers can be improved by employing various workload adaptive techniques in different parts of a standard memory controller. It has been shown that the physical address bits carry useful information, that can be extracted to improve different algorithms employed by a DRAM controller. HAPPY used such information to improve the page closure policy and DReAM took advantage of similar data to predict a workload-specific address mapping scheme. The main advantage of such algorithms that are devised based on the information extracted from the physical address bits is their scalability. This is due to nature of the physical address bits that are scale with $\log_2$ the memory size (i.e. there is only one extra bit required to address a doubled size of memory). Moreover, this PhD thesis addressed how to identify frequently accessed items in an stream of data (ARMOR) which is an important challenge in different areas of research .

8.2 Future Work

Although, this PhD thesis presented some novel hardware architectures to improve the performance of a DRAM controller, this has also opened up a window to what can be done to improve the performance of these memory systems even further. Future work that might build upon the work presented in this thesis include:

- The evaluation methodology employed in this thesis can be improved furthermore by integrating the proposed techniques in a full system simulator to the investigate the effect of process stalls on memory systems.
- Investigating the potential power saving to be gained from the proposed techniques. This requires a reasonable power modelling of the proposed architectures which is postponed for future work.
- Investigating the effect of technique proposed in this thesis on the future memory technologies such as DDR4 and HMC.
- Although the techniques proposed by this PhD thesis target specific parts of a DRAM controller, the methodology that is used in any of three proposals can be used in different area of memory system design as well. Pre-fetchers for DRAMs are one example of such an area that might employ pattern recognition techniques similar to those proposed for HAPPY or DReAM to predict the next memory request in advance. The methodology proposed by ARMOR to detect hot-rows in DRAMs might be used to improve the evictions algorithms, and hence the performance, of caches.

Chapter 9

Publications, Patent and Commercialisation

Publications - Under Review

- **M. Ghasempour**, J. Garside, A. Jaleel and M. Luján."DReAM: Dynamic Rearrangement of Address Mapping in DRAMs" in International Symposium on Microarchitecture (**MICRO**),Waikiki, HI, USA, 2015.
- **M. Ghasempour**, J. Garside and M. Luján."ARMOR: A Run-time Memory hot-row detectOR to prevent Row-Hammer data corruption in DRAMs" in International Conference on Parallel Architectures and Compilation Techniques (**PACT**), San Francisco, CA, USA, 2015.
- **M. Ghasempour**, A. Jaleel, J. Garside and M. Luján."HAPPY: Hybrid Address-based Page PolicY in DRAMs" in International Conference on Parallel Architectures and Compilation Techniques (**PACT**), San Francisco, CA, USA, 2015.

Publications - Accepted

- **M. Ghasempour**, Jonathan Heathcote, Javier Navaridas, Luis A. Plana, J. Garside and M. Luján."Accelerating the Analysis of Interconnection Networks using Reconfigurable Hardware - SpiNNaker as a Case Study" to appear in International Symposium on Field-Programmable Custom Computing Machines (**FCCM**), Vancouver, British Columbia, Canada, 2015.
- O. Arcas, G. Ndu, N. Sonmez, **M. Ghasempour**, A. Armejach, W. Song, J. Mawer, J. Navaridas, O. Unsal, M. Luján, A. Cristal."An Empirical Evaluation

of Hardware Design Languages and Tools for Database Acceleration" in International Symposium on Field-Programmable Logic and Applications (**FPL**), Munich, Germany, 2014.

- **M. Ghasempour**, M. Luján, J. Garside."SoC Simulator on FPGA using Bluespec System Verilog" in UK Electronics Forum (**UKEF**), Newcastle, UK, 2012.

Patent (Filed)

A patent application has been filed to protect ARMOR on behalf of The University of Manchester by The University of Manchester Intellectual Property (UMIP).

- Title: MONITORING DEVICE
- Inventors: **Mohsen Ghasempour**, Mikel Luján and Jim Garside
- GB Application No: 1500446.8

Commercialisation

ARMOR is under process of commercial exploitation with help from UMIP (Figure 9.1 shows ARMOR's logo). More information can be found at ARMOR official website [GLG].

Figure 9.1: ARMOR logo.

Bibliography

[AJW+05] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, C-W Tseng, and Donald Yeung. BioBench: A benchmark suite of bioinformatics applications. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 2–9. IEEE, 2005.

[ANBD11] Manu Awasthi, David W Nellans, Rajeev Balasubramonian, and Al Davis. Prediction based dram row-buffer management in the many-core era. In *International Conference on Parallel Architectures and Compilation Techniques (PACT), 2011.*, pages 183–184. IEEE, 2011.

[BBB+11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen5, Korey Sewell8, Muhammad Shoaib, Nilay Vaish, Mark Hill5, and David Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[BH14] K.S. Bains and J.B. Halbert. Row hammer monitoring based on stored row hammer threshold value. `https://www.google.com/patents/US20140156923`, June 5 2014. US Patent App. 13/690,523.

[BI12] Mahdi Nazm Bojnordi and Engin Ipek. PARDIS: A programmable memory controller for the DDRx interfacing standards. In *39th Annual International Symposium on Computer Architecture (ISCA), 2012*, pages 13–24. IEEE, 2012.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques*, pages 72–81. ACM, 2008.

[Bla13] Matthew Blackmore. A Quantitative Analysis of Memory Controller Page Policies. 2013.

[Blo70] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[CBS+12] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, S Pugsley, A Udipi, Ali Shafiee, Kshitij Sudan, Manu Awasthi, and Zeshan Chishti. USIMM: the utah simulated memory module. *University of Utah, Tech. Rep*, 2012.

[Chr] Chrome. Welcome to Native Client. `https://developer.chrome.com/native-client`. [Accessed: 28-April-2015].

[CLC+14] Kevin Kai-Wei Chang, Donghyuk Lee, Zeshan Chishti, Alaa R Alameldeen, Chris Wilkerson, Yoongu Kim, and Onur Mutlu. Improving DRAM Performance by Parallelizing Refreshes with Accesses. In *30th Annual International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[CM03] Saar Cohen and Yossi Matias. Spectral Bloom filters. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 241–252. ACM, 2003.

[CMBR05] Qikai Chen, Hamid Mahmoodi, Swarup Bhunia, and Kaushik Roy. Modeling and testing of SRAM for new failure mechanisms due to process variations in nanoscale CMOS. In *Proceedings 23rd IEEE VLSI Test Symposium.*, pages 292–297. IEEE, 2005.

[Diaa] Diablo Technologies. MemoryChannel Storage. `http://www.diablo-technologies.com/memory-channel-storage`. [Accessed: 28-April-2015].

[Diab] Diablo Technologies. NanoCommit Technology. `http://www.diablo-technologies.com/nanocommit-technology`. [Accessed: 28-April-2015].

[Dix91] Kaivalya M Dixit. The SPEC benchmarks. *Parallel computing*, 17(10):1195–1209, 1991.

[Dod06] J.M. Dodd. Adaptive page management. http://www.google.com/patents/US7076617, July 11 2006. US Patent 7,076,617.

[EMF+11] Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, José A Joao, Onur Mutlu, and Yale N Patt. Parallel application memory scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 362–373. ACM, 2011.

[eTe14] eTeknix. Price of 2GB DDR3 modules hit all time low. http://www.eteknix.com/price-of-2gb-ddr3-modules-hits-all-time-low, 2014. [Accessed: 28-April-2015].

[FK91] John A Fifield and Howard L Kalter. Crosstalk-shielded-bit-line DRAM, April 23 1991. US Patent 5,010,524.

[GBS+14] Z. Greenfield, K.S. Bains, T.Z. Schoenborn, C.P. Mozak, and J.B. Halbert. Row hammer condition monitoring. http://www.google.com/patents/US20140006704, January 2 2014. US Patent App. 13/539,417.

[GCP+09] Zheng Guo, Andrew Carlson, Liang-Teck Pang, Kenneth T Duong, Tsu-Jae King Liu, and Borivoje Nikolic. Large-scale SRAM variability characterization in 45 nm CMOS. *IEEE Journal of Solid-State Circuits*, 44(11):3174–3192, 2009.

[GLG] Mohsen Ghasempour, Mikel Luján, and Jim Garside. ARMOR: A Hardware Solution to Prevent Row Hammer Error in DRAMs. http://apt.cs.manchester.ac.uk/projects/ARMOR/RowHammer/index.html. [Accessed: 28-April-2015].

[Goo] Google. Chrome v8. https://developers.google.com/v8/. [Accessed: 28-April-2015].

[HLHL06] Dandan Huan, Zusong Li, Weiwu Hu, and Zhiyong Liu. Processor directed dynamic page policy. In *Advances in Computer Systems Architecture*, pages 109–122. Springer, 2006.

[HS93] W-C Hsu and James E Smith. Performance of cached DRAM organizations in vector supercomputers. *ACM SIGARCH Computer Architecture News*, 21(2):327–336, 1993.

[IBM14] IBM. Phase Change Memory. `http://www.research.ibm.com/labs/zurich/sto/pcm/`, 2014. [Accessed: 28-April-2015].

[IFSK12] Takakazu Ikeda, Naoki Fujieda, Shimpei Sato, and Kenji Kise. Request Density Aware Fair Memory Scheduling. In *3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC)*, 2012.

[IMMC08] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *35th International Symposium on Computer Architecture, 2008. ISCA'08.*, pages 39–50. IEEE, 2008.

[Int] Intel. Intel xeon processor x5650. `http://ark.intel.com/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz`. [Accessed: 28-April-2015].

[Ito01] Kiyoo Itoh. *VLSI memory chip design*, volume 5. Springer New York, 2001.

[JD04] B.G. Johnson and C.H. Dennison. Phase change memory, September 14 2004. US Patent 6,791,102.

[JHG13] Yoon H Jung, Hunter C Hillery, and Tressler A Gary. Flash and DRAM Si Scaling Challenges, Emerging Non-Volatile Memory Technology Enablement. In *Flash Memory Summit*, 2013.

[JNW10] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[Joh] John Ousterhout. RAMCloud. `https://ramcloud.stanford.edu/wiki/display/ramcloud/RAMCloud`. [Accessed: 28-April-2015].

[KCA+11] Daeyeon Kim, Vikas Chandra, Robert Aitken, David Blaauw, and Dennis Sylvester. Variation-aware static and dynamic writability analysis for voltage-scaled bit-interleaved 8-T SRAMs. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pages 145–150. IEEE Press, 2011.

[KDD10] Karthik Kumar, Martin Dimitrov, and Kshitij Doshi. Energy efficient DRAM row buffer management for enterprise workloads. In *International Conference on Energy Aware Computing (ICEAC)*, pages 1–4. IEEE, 2010.

[KDK+14] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *41st International Symposium on Computer Architecture (ISCA)*, pages 361–372. ACM/IEEE, 2014.

[Kee08] Brent Keeth. *DRAM circuit design: fundamental and high-speed topics*, volume 13. John Wiley & Sons, 2008.

[KHMHB10] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2010.

[KPMHB11] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling. *Micro, IEEE*, 31(1):78–89, 2011.

[KSJ11] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy Kurian John. Minimalist open-page: A DRAM page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 24–35. ACM, 2011.

[KSL+12] Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu. A case for exploiting subarray-level parallelism (SALP) in DRAM. In *39th Annual International Symposium on Computer Architecture (ISCA)*, pages 368–379. IEEE, 2012.

[KSP03] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)*, 28(1):51–55, 2003.

[LCC+07] Ki-Won Lee, Joo-Hwan Cho, Byoung-Jin Choi, Geun-Il Lee, Ho-Don Jung, Woo-Young Lee, Ki-Chon Park, Yong-Suk Joo, Jae-Hoon Cha,

Young-Jung Choi, et al. A 1.5-V 3.2 Gb/s/pin Graphic DDR4 SDRAM with dual-clock system, four-phase input strobing, and low-jitter fully analog DLL. *Journal of Solid-State Circuits*, 42(11):2369–2377, 2007.

[LWWX12] Chongmin Li, Dongsheng Wang, Haixia Wang, and Yibo Xue. Priority Based Fair Scheduling: A Memory Scheduler Design for Chip-Multiprocessor Systems. *Tsinghua National Laboratory for Information Science and Technology*, 2012.

[Mara] Mark Seaborn - Google Project Zero. Exploiting the DRAM rowhammer bug to gain kernel privileges. `http://googleprojectzero.blogspot.co.uk/2015/03/exploiting-dram-rowhammer-bug-to-gain.html`. [Accessed: 28-April-2015].

[Marb] Mark Seaborn - Google Project Zero. Program for testing for the DRAM "rowhammer" problem. `https://github.com/google/rowhammer-test`. [Accessed: 28-April-2015].

[MAW01] Seiji Miura, Kazushige Ayukawa, and Takao Watanabe. A dynamic-SDRAM-mode-control scheme for low-power systems with a 32-bit RISC CPU. In *Proceedings of the International Symposium on Low power Electronics and Design*, pages 358–363. ACM, 2001.

[MC07] Chiyuan Ma and Shuming Chen. A DRAM Precharge Policy Based on Address Analysis. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools.*, pages 244–248. IEEE, 2007.

[MDB+02] Jack A Mandelman, Robert H Dennard, Gary B Bronner, John K DeBrosse, Rama Divakaruni, Yujun Li, and Carl 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, 2002.

[Mica] Mike Micheletti. Tuning DDR4 for Power and Performance. `http://cdn.teledynelecroy.com/files/whitepapers/tuningddr4_for_power_performance.pdf`. [Accessed: 28-April-2015].

[Micb] Micron Technology Inc. RDIMM. `http://www.micron.com/products/dram-modules/rdimm`. [Accessed: 28-April-2015].

[Micc] Micron Technology Inc. RDIMM: Registered Memory Modules. http://www.micron.com/-/media/documents/products/data/20sheet/modules/parity_rdimm/jszs72c2g_4gx72pz.pdf. [Accessed: 28-April-2015].

[Micd] Micron Technology Inc. Target Row Refresh Mode. http://www.micron.com/products/datasheets/3d323c4d-6bc7-4193-908d-e99ad746aa4e?page=13. [Accessed: 28-April-2015].

[Mic14a] Micron Technology Inc. DDR3 to DDR4, 2014.

[Mic14b] Micron Technology Inc. DDR4 SDRAM. http://www.micron.com/products/dram/ddr4-sdram, 2014. [Accessed: 28-April-2015].

[Mic14c] Micron Technology Inc. DDR4 SDRAM - MT40A1G4HX-083E. http://www.micron.com/parts/dram/ddr4-sdram/mt40a1g4hx-083e, 2014. [Accessed: 28-April-2015].

[Mic14d] Micron Technology Inc. Hybrid Memory Cube. http://www.micron.com/products/hybrid-memory-cube, 2014. [Accessed: 28-April-2015].

[Mic14e] Micron Technology Inc. Hybrid Memory Cube Consortium. http://www.hybridmemorycube.org, 2014.

[Mic14f] Micron Technology Inc. Phase Change Memory Innovations. http://www.micron.com/about/innovations/pcm, 2014. [Accessed: 28-April-2015].

[Mik] Mike Micheletti. Teledyne LeCroy Upgraded DDR Protocol Analyzer adds Row Hammer Reporting and System Memory Mapping. http://cdn.teledynelecroy.com/files/pressreleases/09042013.pdf. [Accessed: 28-April-2015].

[MKK+12] Young-Suk Moon, Yongkee Kwon, Hong-Sik Kim, Dong-gun Kim, Hyungdong Hayden Lee, and Kunwoo Park. The Compact Memory Scheduling Maximizing Row Buffer Locality. In *3rd JILP Workshop on Computer Architecture Competitions (JWAC-3): Memory Scheduling Championship (MSC)*, 2012.

[MM07] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 146–160. IEEE Computer Society, 2007.

[MM08] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 63–74. IEEE Computer Society, 2008.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[MSY+90] Dong-Sun Min, Dong-Il Seo, Jehwan You, Sooin Cho, Daeje Chin, et al. Wordline coupling noise reduction techniques for scaled DRAMs. In *Symposium on VLSI Circuits, Digest of Technical Papers.*, pages 81–82, 1990.

[NALS06] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *39th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-39.*, pages 208–222. IEEE, 2006.

[OAE+10] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[ORS+11] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[Paw11] J Thomas Pawlowski. Hybrid Memory Cube (HMC). In *Hotchips*, volume 23, pages 1–24, 2011.

[PP03] Seong-Il Park and In-Cheol Park. History-based memory mode prediction for improving memory performance. In *Proceedings of the International Symposium on Circuits and Systems. ISCAS'03.*, volume 5, pages V–185. IEEE, 2003.

[Raj] Rajinder Gill. Everything you always wanted to know about SDRAM memory but were afraid to ask. `http://www.anandtech.com/show/3851/everything-you-always-wanted-to-know-about-sdram-memory-but-were-afraid-to-ask/6`. [Accessed: 28-April-2015].

[RMMM03] Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage current mechanisms and leakage reduction techniques in deep-submicrometer CMOS circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003.

[Sch97] Reinhard C Schumann. Design of the 21174 memory controller for DIGITAL personal workstations. *Digital Technical Journal*, 9:57–70, 1997.

[SCN+10] Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. Micro-pages: increasing DRAM efficiency with locality-aware data placement. *ACM Sigplan Notices*, 45(3):219–230, 2010.

[SHU+00] 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 *Electron Devices Meeting, 2000. IEDM'00. Technical Digest. International*, pages 837–840. IEEE, 2000.

[SK13] Daniel Sanchez and Christos Kozyrakis. ZSIM: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 475–486. ACM, 2013.

[SK14] Vasily A Sartakov and Rudiger Kapitza. NV-Hypervisor: Hypervisor-based Persistence for Virtual Machines. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–659. IEEE, 2014.

[SKF+13] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B Gibbons, Michael A Kozuch, and Todd C Mowry. Rowclone: Fast and energy-efficient in-DRAM bulk data copy and initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–197. ACM, 2013.

[SL12] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2012.

[SM04] Vladimir V Stanković and Nebojša Milenković. Access latency reduction in contemporary dram memories. *Facta universitatis-series: Electronics and Energetics*, 17(1):81–97, 2004.

[SM05a] Vladimir V Stanković and Nebojsa Z Milenkovic. DRAM Controller with a Close-Page Predictor. In *Computer as a Tool, 2005. EUROCON 2005. The International Conference on*, volume 1, pages 693–696. IEEE, 2005.

[SM05b] Vladimir V Stanković and NZ Milenkovic. DRAM controller with a complete predictor: Preliminary results. In *7th International Conference on Telecommunications in Modern Satellite, Cable and Broadcasting Services.*, volume 2, pages 593–596. IEEE, 2005.

[SNS+13] Kyomin Sohn, Taesik Na, Indal Song, Yong Shim, Wonil Bae, Sanghee Kang, Dongsu Lee, Hangyun Jung, Seokhun Hyun, Hanki Jeoung, Ki-Won Lee, Jun-Seok Park, Jongeun Lee, Byunghyun Lee, Inwoo Jun, Juseop Park, Junghwan Park, Hundai Choi, Sanghee Kim, Haeyoung Chung, Young Choi, Dae-Hee Jung, Byungchul Kim, Jung-Hwan Choi, Seong-Jin Jang, Chi-Wook Kim, Jung-Bae Lee, and Joo Sun Choi. A 1.2 V 30 nm 3.2 Gb/s/pin 4 Gb DDR4 SDRAM with dual-error detection and PVT-tolerant data-fetch scheme. *Journal of Solid-State Circuits*, 48(1):168–177, 2013.

[SPE09] SPECIFICATION, DDR3 SDRAM. JEDEC STANDARD. 2009.

[SW11] Claude Sammut and Geoffrey I Webb. *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[TTK+92] Akira Tanabe, Toshio Takeshima, Hiroki Koike, Yoshiharu Aimoto, Masahide Takada, Toshiyuki Ishijima, Naoki Kasai, Hiromitsu Hada, Kentaro Shibahara, Tahemitsu Kunio, et al. A 30-ns 64-mb dram with built-in self-test and self-repair function. *Solid-State Circuits, IEEE Journal of*, 27(11):1525–1533, 1992.

[Vika] Viking Technology. ArxCis-NM: Non-Volatile Memory Technology. http://www.vikingtechnology.com/arxcis-nv. [Accessed: 28-April-2015].

[Vikb] Viking Technology. NV-DIMM: Fastest Tier in Your Storage Strategy. http://www.vikingtechnology.com/uploads/nvdimm_tiered_storage.pdf. [Accessed: 28-April-2015].

[Vikc] Viking Technology. NVDIMM vs SSD: A Performance and ROI Comparison. http://www.vikingtechnology.com/uploads/NVDIMM_Technical_Comparison.pdf. [Accessed: 28-April-2015].

[weba] Diablo Technologies. http://www.diablo-technologies.com. [Accessed: 28-April-2015].

[webb] Viking technology. http://www.vikingtechnology.com. [Accessed: 28-April-2015].

[WM95] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[WRK+10] H-SP Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase Change Memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[XAD09] Ying Xu, Aabhas S Agarwal, and Brian T Davis. Prediction in dynamic SDRAM controller policies. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 128–138. Springer, 2009.

[ZZZ00] Zhao Zhang, Zhichun Zhu, and Xiaodong Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual International Symposium on Microarchitecture*, pages 32–41. ACM, 2000.