THE PERFORMANCE AND POWER IMPACT OF USING MULTIPLE DRAM ADDRESS MAPPING SCHEMES IN MULTICORE PROCESSORS

by

RAMI JADAA
B.S. University of Sharjah, 2006

A thesis submitted in partial fulfillment of the requirements
for the degree of Masters of Science in Electrical Engineering
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, FL

Fall Term
2011

Major Professor: Mark Heinrich

© 2011 Rami Jadaa

ABSTRACT

Lowest-level cache misses are satisfied by the main memory through a specific address mapping scheme that is hard-coded in the memory controller. A dynamic address mapping scheme technique is investigated to provide higher performance and lower power consumption, and a method to throttle memory to meet a specific power budget. Several experiments are conducted on single and multithreaded synthetic memory traces -to study extreme cases- and validate the usability of the proposed dynamic mapping scheme over the fixed one. Results show that applications' performance varies according to the mapping scheme used, and a dynamic mapping scheme achieves up to 2x increase in peak bandwidth utilization and around 30% higher energy efficiency than a system using only a single fixed scheme Moreover, the technique can be used to limit memory accesses into a subset of the memory devices by controlling data allocation at a finer granularity, providing a method to throttle main memory by allowing un-accessed devices to be put into power-down mode, hence saving power to meet a certain power budget.

I dedicate this work to the three pillars of my life: Allah, my family, and my friends.

Those whom God has guided have the true guidance, but those whom He has caused to go astray are certainly lost. All thanks belong to Allah. Whatever good I may achieve is certainly form God, and whatever I suffer is from myself.

To my family that always provided all the support in the past and present to brighten my future. I hope this work lightens a candle in my future, like you lightened my life.

To my friends who cared, tolerated and brought joy to my journey. I can't imagine my life without them.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor, Dr. Mark Heinrich, for giving me the support, direction, and knowledge delivered through many courses, that paid off well in this thesis. I want to thank you for the unbounded guidance, understanding, tolerance and motivation.

I am thankful as well for Dr. Mingjie Lin's orientation through my initial readings. He was indeed an essential part of the research group, and never hesitated to provide direction whenever required.

My gratitude goes to my heroes of the Memory-Systems research group at the University of Maryland for providing such a powerful, yet simple and well-written simulator.

My deepest appreciation goes to Bruce Jacob, Spencer W. Ng, and David T. Wang, the authors of "Memory Systems: Cache, DRAM, Disk" book, which represented the bible of my study in this research.

TABLE OF CONTENTS

TABLE OF CONTENTS ii

LIST OF FIGURES iv

CHAPTER 1: INTRODUCTION TO DRAM AND MEMORY CONTROLLERS 1

1.1 DRAM Basics: Architecture, Organization, and Operation 2

1.1.1 DRAM Architecture and Capacity 3

1.1.2 DRAM Operation and Address Mapping Schemes 7

CHAPTER 2: RELATED WORK 17

2.1 Merged Logic DRAMs 17

2.2 Smart Memory Controllers 20

2.3 DRAM Power Management 24

CHAPTER 3: RESEARCH DESIGN AND METHODOLOGY 27

3.1 Motivation 27

3.2 DRAMSim2 Simulator 28

3.3 Experimental Setup 32

3.3.1 Metrics 33

3.3.2 Scheme-Matching Synthetic Trace Generator 34

3.3.3 Simulation Configuration 37

3.3.4 Dynamic Address Mapping Scheme Design 40

3.3.5 Set of Experiments 41

CHAPTER 4: RESULTS AND ANALYSIS 43

4.1 Single-Scheme Device with Scheme-Matching Synthetic Trace 43

4.2 Single-Scheme Device with Multi-Threaded Synthetic Traces 48

4.3 Default Scheme vs. Dynamic Multiple Mapping Schemes 51

CHAPTER 5: CONCLUSION AND FUTURE WORK 55

5.1 Summary 55

5.2 Future Work 56

LIST OF FIGURES

Figure 1-1: PC System Organization 2

Figure 1-2: JEDEC-Style DRAM module 3

Figure 1-3: Several devices connected in parallel to the bus 4

Figure 1-4: DRAM Internals, Memory cells arranged in rectangular arrays 5

Figure 1-5: Multiple arrays staked. The number of arrays defines the device width 6

Figure 1-6: SDRAM module of 4 banks with the control circuitry 8

Figure 1-7: Row selected spanning all DRAM devices 10

Figure 1-8: Example of parallel access of same bank, row and column in all devices 12

Figure 1-9: Example of mapping physical address into a mapping scheme 15

Figure 2-1: Overall organization of a FlexRam-based system 19

Figure 2-2: Conventional vs. PAR-BS DRAM request scheduling 22

Figure 2-3: Impulse vs. conventional MC data mapping and prefetching 23

Figure 3-1: DRAMSim2 Functional Flow Diagram 30

Figure 3-2: Sample DRAMSim2 Epoch Output 31

Figure 3-3: Per-thread address mapping 41

Figure 4-1: BW utilization of SchemeB-matching trace with different read/write ratios 44

Figure 4-2: Energy/Op of SchemeB-matching trace with different read/write ratios 44

Figure 4-3: BW utilization on a single-scheme device configuration 47

Figure 4-4: Energy/Op on a single-scheme device configuration 48

Figure 4-5: BW utilization of mixed synthetic traces on a single AMS 49

Figure 4-6: Energy/Op for mixed synthetic traces on a single AMS 50

Figure 4-7: Per-Bank bandwidth from different runs on a single scheme device 51

Figure 4-8: BW utilization of mixed synthetic traces on a DMS 52

Figure 4-9: Energy/op of running mixed synthetic traces on a DMS 53

Figure 4-10: Per-Bank bandwidth from different runs on a DMS 54

Figure 5-1: Utilization of some multimedia benchmarks 57

LIST OF TABLES

Table 1: Three different confiugrations for a 4GB DRAM module 7

Table 2: DRAM access timings 13

Table 3: Device Configurations and Electrical Parameters 38

Table 4: Timing parameters for the used device in simulations 39

Table 5: Address mapping schemes studied 42

Table 7: Bandwidth Utilization.. 46

Table 8: Energy/Op 46

CHAPTER 1: INTRODUCTION TO DRAM AND MEMORY CONTROLLERS

The memory system is an essential part of a computer system. All modern software based on the von Neumann architecture relies on the storage of programs (code and data) in the memory system at different levels of the memory hierarchy. The hierarchal design of memory into caches, DRAMs, and disks is very important to the operation of the processor to approximate the objectives of an ideal memory system:

- *Infinite capacity*: Storing large data sets and programs
- *Infinite bandwidth*: Supplying the processor with its data and code
- *Clock-cycle latency*: Preventing processor stalls
- *Low cost*: Minimizing cost for large data storage

Cache memory represents the closest memory unit to the processor, communicating directly with it at different levels (L1, L2). They are characterized by very low latency (1 to 10 processor cycles) and high bandwidth. However, because of their limited capacity, caches rely on the locality principle (temporal and spatial) to feed the processor with instructions and data at the rate it requires. Due to size and mapping limitations, cache misses are inevitable, and must be satisfied by the main memory, typically constructed from DRAM (Dynamic Random Access Memory).

DRAM provides a high aggregate bandwidth, relatively low latency (on the order of 100s of processor cycles), high capacity, and low cost memory system. To understand memory

hierarchy better, Fig-1 shows a typical PC system organization, in which the processor has two levels of cache, and communicates with external systems via the memory controller in the North Bridge chipset. Whenever a memory request is not satisfied by the caches –i.e. a cache miss, it goes to the DRAM through the Memory Controller.

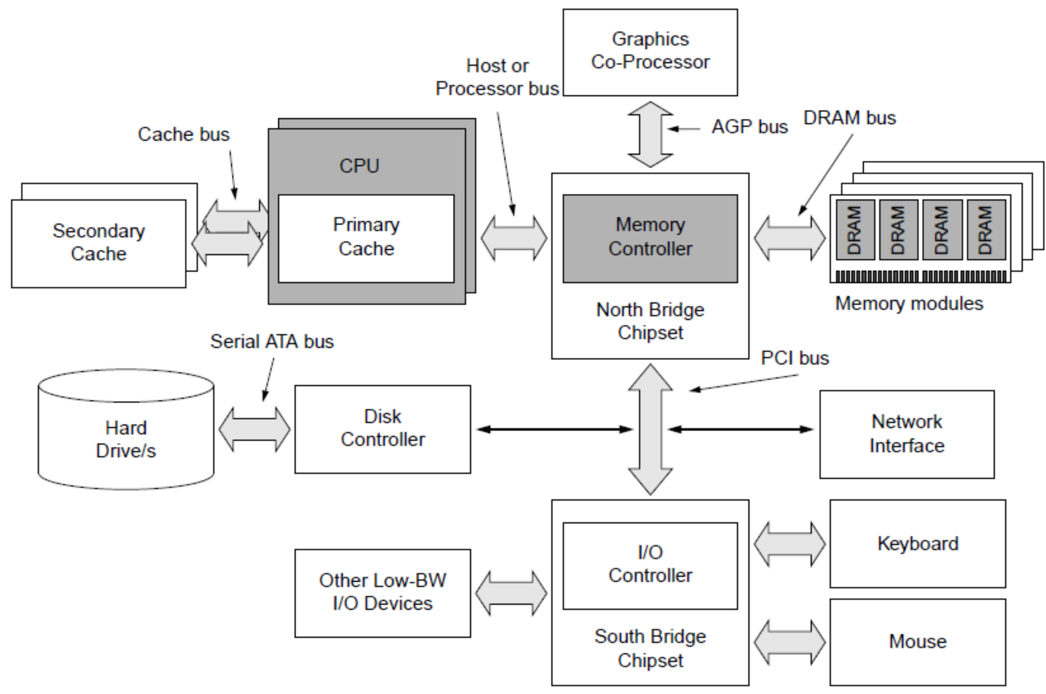

Figure 1-1: PC System Organization

The Operating System (OS) controls the memory allocation of programs by controlling page allocation in main memory. Whenever a request is received by the memory controller, it converts the memory address into a physical location in the DRAM.so that it can transfer the required data back to the processor cache. The handling of a cache miss is a crucial process both in terms of power and performance, as the processor could be stalled until such a request is

returned. The process of handling a cache miss relies on many factors in the design of a memory controller, DRAM organization, and page allocation in memory.

Our research here focuses on minimizing the stall time of a cache miss request to DRAM, by a better utilization of Memory Level Parallelism (MLP) and hence provide higher level of energy efficiency by reducing the energy required for each operation (read/write) To understand our contribution, we need to understand the details of DRAM architecture and organization, and the role of the Memory Controller in handling a processor cache miss. The next section provides a detailed introduction of DRAM technology, followed by introduction description of memory controllers and their operation. Chapter 2 discusses related work, in particular the contributions made by merged logic design, Active Memory controllers and better DRAM bank management. Chapter 3 explains our technique of dynamic address mapping schemes and its many implications in our research. Chapter 4 discusses our simulation results, and analyzes the data collected to understand how a dynamic address mapping scheme can provide better performance and energy efficiency.. Finally, Chapter 5 summarizes the results, provides a conclusion and lays the ground for future work.

1.1 DRAM Basics: Architecture, Organization, and Operation

There are two main types of Random Access Memory (RAM): Static and Dynamic. DRAM belongs to the dynamic memory class (**D**ynamic **R**andom **A**ccess **M**emory), in which the *dynamic* term refers to the requirement of refreshing the stored data periodically at certain *refresh* intervals.

Usually, DRAMs are explained in a bottom-up fashion, in which the smallest storage element is explained first, then going up in the hierarchy. However, a top-down description is followed here as it provides a better understanding of how data is organized without using confusing terminology. The next section provides a detailed top-down explanation of the internal DRAM architecture.

1.1.1 DRAM Architecture and Capacity

Fig. 1-2 shows the hierarchy of typical JEDEC[1]-style DRAM modules. Several DRAMs are grouped on a printed-circuit- board to form a **D**ual **I**n-line **M**emory **M**odule (DIMM). The capacity of the DRAM memory system is determined by the number of storage devices it has. A set of storage **devices** are organized to form a **rank**, in which they operate in unison on a single channel connected to the data bus. A single DIMM usually has 1 or 2 ranks, depending on the configuration used.

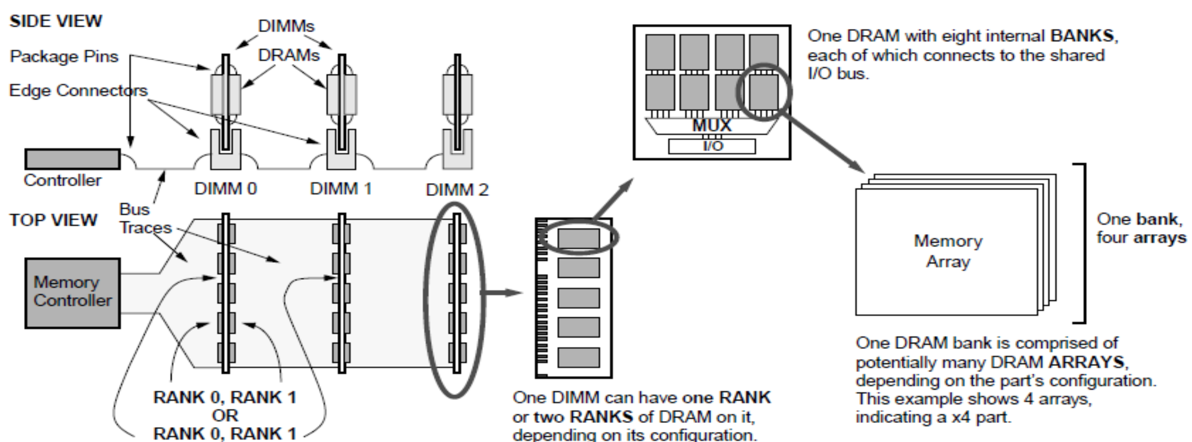

Figure 1-2: JEDEC-Style DRAM module

[1] **J**oint **E**lectron **D**evices **E**ngineering **C**ouncil is is an independent semiconductor engineering trade organization and standardization body

The DIMMs are connected to the memory controller through a bus. JEDEC-style buses are classified according to their function into: *command* (read, write, refresh...etc), *address* (address at which data is read/written) and *data*, in which data bus width is standardized to be 64-bit wide. Each storage device has a certain width, called the **device width**, which constitutes a part of the 64-bit wide bus. Hence, the device width defines the number of storage devices required on an n-bit bus system by dividing the bus width by the device width For example, a **x4** DRAM (pronounced as by 4) has a device width of 4 bits, in which 16 devices (64/4) act in unison to provide the 64-bit data required for a read/write operation. Modern DRAMs are usually x16 or x32. Fig. 1-3 shows how devices are connected in parallel in a single rank to the bus.

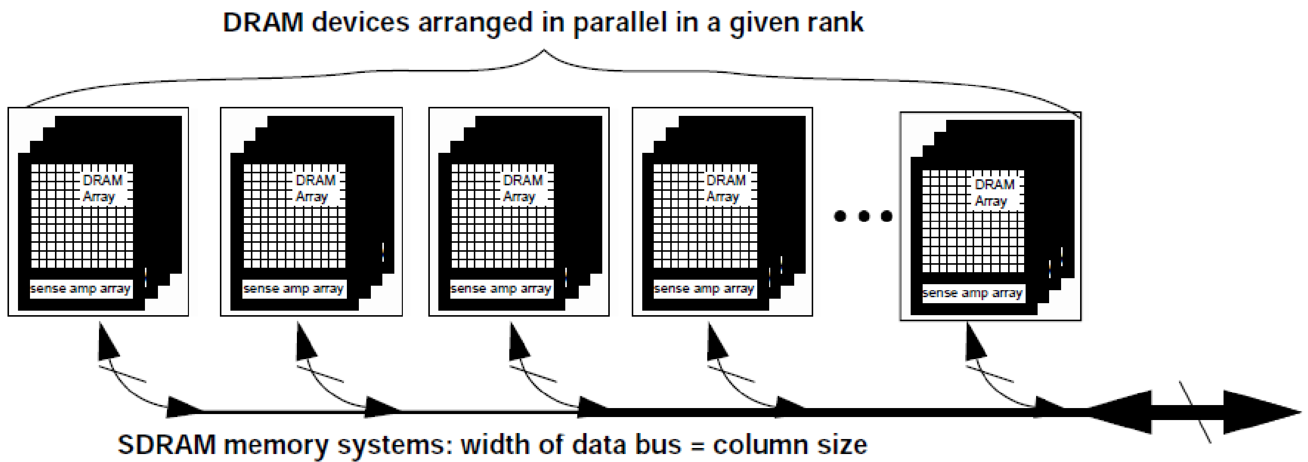

Figure 1-3: Several devices connected in parallel to the bus

Each storage device is internally **banked**. Each bank is made up of rectangular arrays of the basic-storage blocks (memory cells). The memory cell is a single-bit dynamic storage element in which a transistor-capacitor pair is used to store a single bit of data. It is dynamic in the sense that the capacitor is not an ideal device, and current leakage will cause the capacitor

charge (and hence the data stored) to be lost if not *refreshed* periodically. Fig.1-4 shows how memory cells are arranged in a rectangular array. Each cell is accessible by activating the corresponding horizontal (**row**) and vertical (**column**) lines.

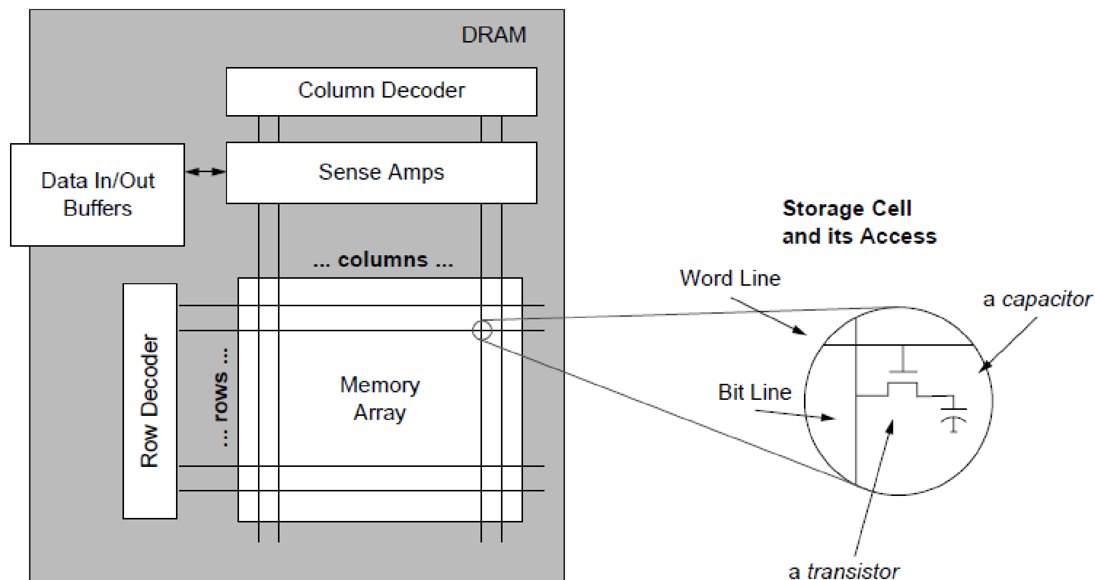

Figure 1-4: DRAM Internals, Memory cells arranged in rectangular arrays

To read the data stored in a memory cell, the corresponding row of the cell is activated (set high), switching the transistor on, and providing a path for the charge stored in the capacitor to be sensed by the Sense Amplifiers in the columns. Then a column address strobe (CAS) is sent to decide which column(s) to read the data from. The row line is usually called a **Word** or **Page line** as it activates multiple bits across several storage devices, while the column line is called a bit-line as each activates a single-bit memory cell. With the architecture provided in Fig.1-4, a single bit is read with each CAS. However, to increase the storage density, *n*-arrays are usually stacked, in which a single CAS will result in sensing *n*-bits at a time (Fig.1-5). In fact, the

number of stacked arrays represents the device width. The data read by a single CAS represent the smallest addressable unit in the DRAM.

From the above, the location of the data stored in a memory cell is identified by the *row* and *column*, then the *bank* it exists in and finally the *rank*. The total number of memory cells in a DRAM represents the total storage capacity. It can be determined by the number of ranks the DIMM has, the number of storage devices in a rank, and finally the storage capacity of a single storage device.

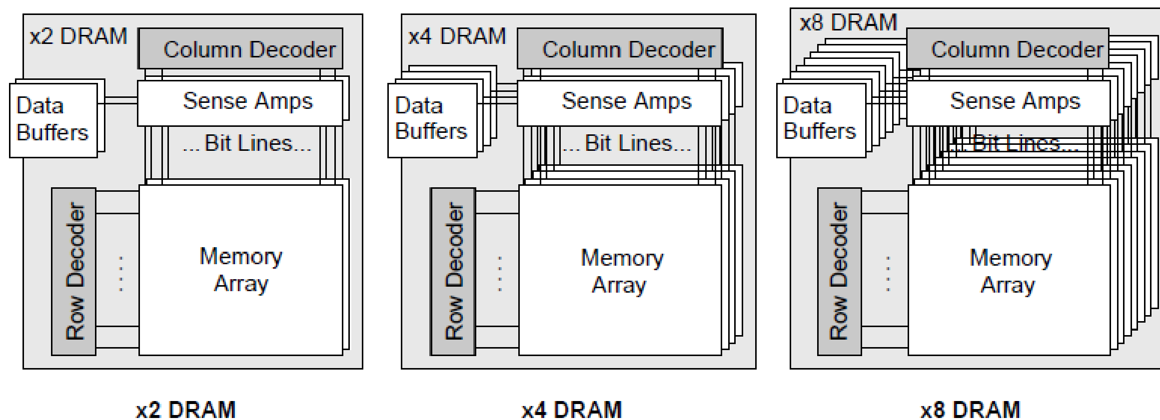

Figure 1-5: Multiple arrays staked. The number of arrays defines the device width

Table 1 shows different configurations of the same storage capacity of a 4GB DRAM module. A DRAM module capacity is determined by the following three main configurations:

- Device Density: This represents the single storage device capacity. It's the number of banks, multiplied by the number of rows and columns, and the device width.
- Number of Devices: Obtained by dividing the data bus width by the device width.
- Number of Ranks.

Finally, the full DRAM capacity is obtained by multiplying the above three values. The differences in the configurations lead to different bits-per-bitline, row activation, and/or bits transmitted at every transaction. This has an impact on the DRAM cost, application performance and power consumption with different configurations.

Table 1: Three different confiugrations for a 4GB DRAM module

Number of Ranks	Number of Devices	Device Density	Device Width	Number of Banks	Number of Rows	Number of Columns
1	8	4 GBit	x8	8	32768	2048
1	4	8 GBit	x16	8	65536	1024
2	16	1 GBit	x4	8	16384	2048

1.1.2 DRAM Operation and Address Mapping Schemes

Old DRAM models used to be asynchronous, in which the DRAM chip is not tied to the system clock. Newer systems rely on clocked Synchronous DRAMs (SDRAM) as they provide faster performance (via pipelining) and a simpler interface. Fig.1-1 shows how DRAM modules are connected through a bus to the **M**emory **C**ontroller (MC). The memory controller serves as the intermediate communicator between the CPU and other interfacing peripherals, including the DRAM modules. For example, when a missed cache-line request arrives to the memory controller, it fetches the data from the DRAM modules, handling all the required control logic to satisfy the request without processor intervention. It is an involved process as the memory controller must know the location of the data to be read/written and handle the timing constraints

of the different row/column/bank states of the attached DRAM. Fig.1-6 shows the control circuitry of a DRAM module of 4 banks, interfaced with the MC.

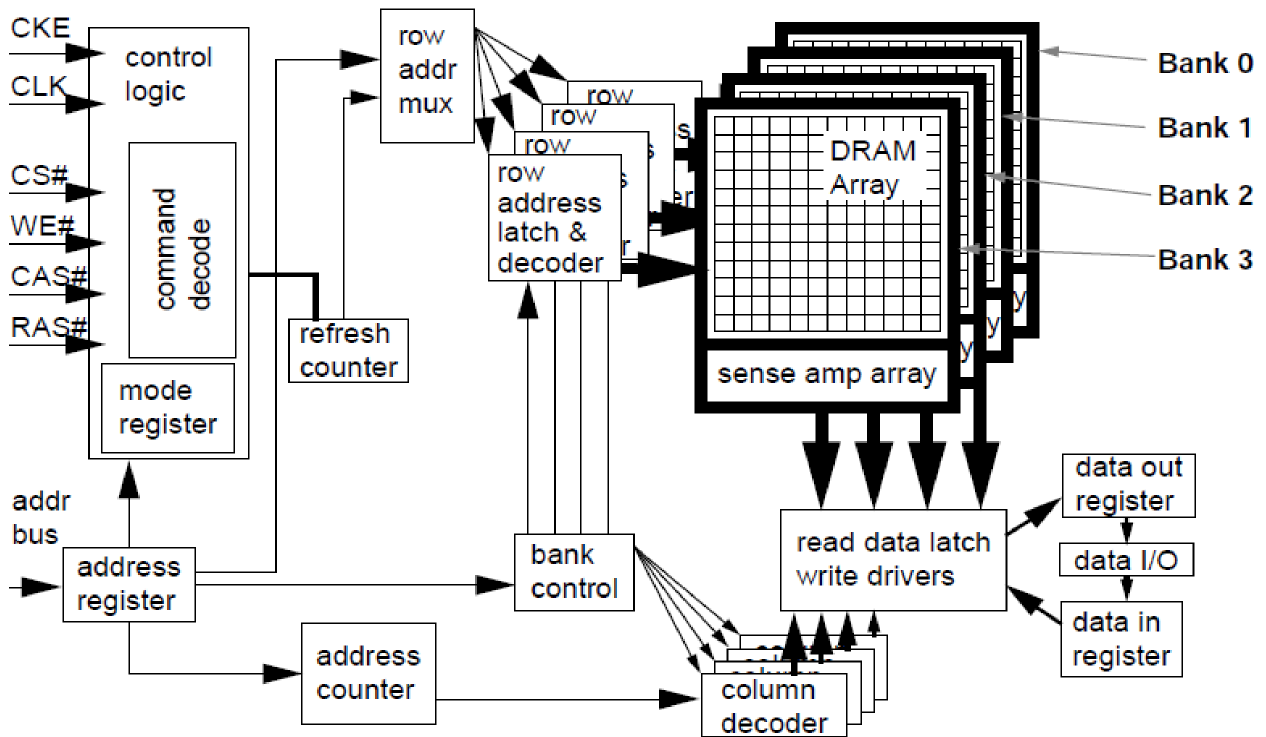

Figure 1-6: SDRAM module of 4 banks with the control circuitry

To simplify the process of data access in DRAMs, the following steps explain the different states and main timing constraints that need to be met during the process:

1- A data access request arrives at the memory controller and is inserted into a pending request queue.

2- Requests in the queue are then served according to the queuing policy, usually on First Come First Services (FCFS) basis, and the MC determines the *command* to be sent (Read, Write) on the command bus, the *address* of data accessed on the address

bus, and finally forwards the data from/to the DRAM/processor according the data access type.

3- The memory controller decodes the address into rank, bank, row and column from the physical address of the cache line received according to the *mapping scheme* (explained in section 1.1.2.2). A chip-select command activates the desired rank through chip-select logic. The MC should be aware of the bank states to coordinate timings accordingly. Each bank could be in one of the following sequential states, ordered by the least time required to accomplish the data access :

    a. *Row Active*: A certain row is precharged, and ready for column access. This depends on the page policy used (open vs. closed). Page policy is explained later in this section.
    b. *Precharging*: Precharging a row for data access.
    c. *Idle*: Ready to receive requests, and once received will precharge the row required.
    d. *Refreshing*: Going through refresh cycle to retain data values.
    e. *Power Down*: A state at which the bank draws minimal current to reduce power consumption. Satisfying a data access in this state has the highest latency, as the DRAM must power up and move back through states c, b, and c sequentially.

4- .Once the correct bank is selected (using bank select logic) the corresponding row is selected through the row decoder, precharged and activates the transistors of the memory cells, allowing the stored charged to be read. This process is called the Row

Access Strobe (RAS), and it happens after t_{RAS} of bank activation. It is important to note that the row activates all bitlines spanning all storage devices, and is hence referred to as DRAM **page**. Fig. 1-7 shows an example of the row select.

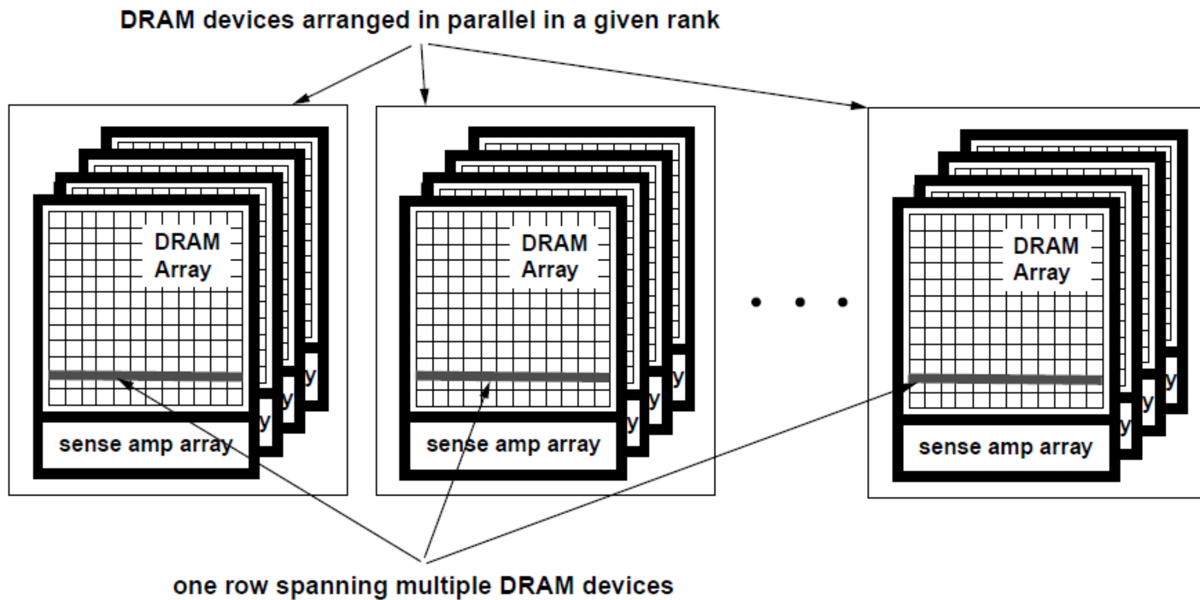

Figure 1-7: Row selected spanning all DRAM devices

5- For read accesses, the sense amplifies read the charges stored in the capacitors, determine the value of each (0 or 1), and keep the values read in the row-buffer. For write accesses, the same process happens, but the sense amplifiers are forced to a voltage high/low depending on the value to be written, which drives the charge on the capacitors to the value required.

6- There are two page policies adopted here:

a. *Open-Page*: Keeping the read row active by keeping sense amplifiers open and holding the read page for further accesses. It favors memory accesses to the same row, exploiting data locality.

b. *Closed-Page*: Keeping the read row active for one access only then discharges the row, getting the bank ready for other row accesses. It favors memory accesses to different rows, and is used usually in the context of multi-core/threaded applications, where data locality is not exhibited.

7- After t_{RCD} passes (Row to Column Delay), the column address is sent to the decoder, and the corresponding column value is read/written from the activated page. This process is referred to as Column Access Strobe (CAS). Data will be available on the bus after t_{CAS} passes.

8- Since a single CAS is an *N*-bit access (device width), a single CAS is not enough to provide the value of the cache line. Therefore, a **burst** CAS happens, in which, starting at the address of the first column decoded, a number of consecutive columns are accessed consecutively (between 4 and 8) to provide the data for the cache line required. The narrower the channel the higher the burst value is. A Column to Column Delay (t_{CCD}) is added after each CAS.

9- We should keep in mind that the same bank, row and column in all devices are accessed in parallel, and data is then sent in *beats* across the data bus. In Double Data Rate DRAMs (DDR DRAM), two beats happen per cycle. Fig.1-8 provides an example of the parallel access of devices for an example address.

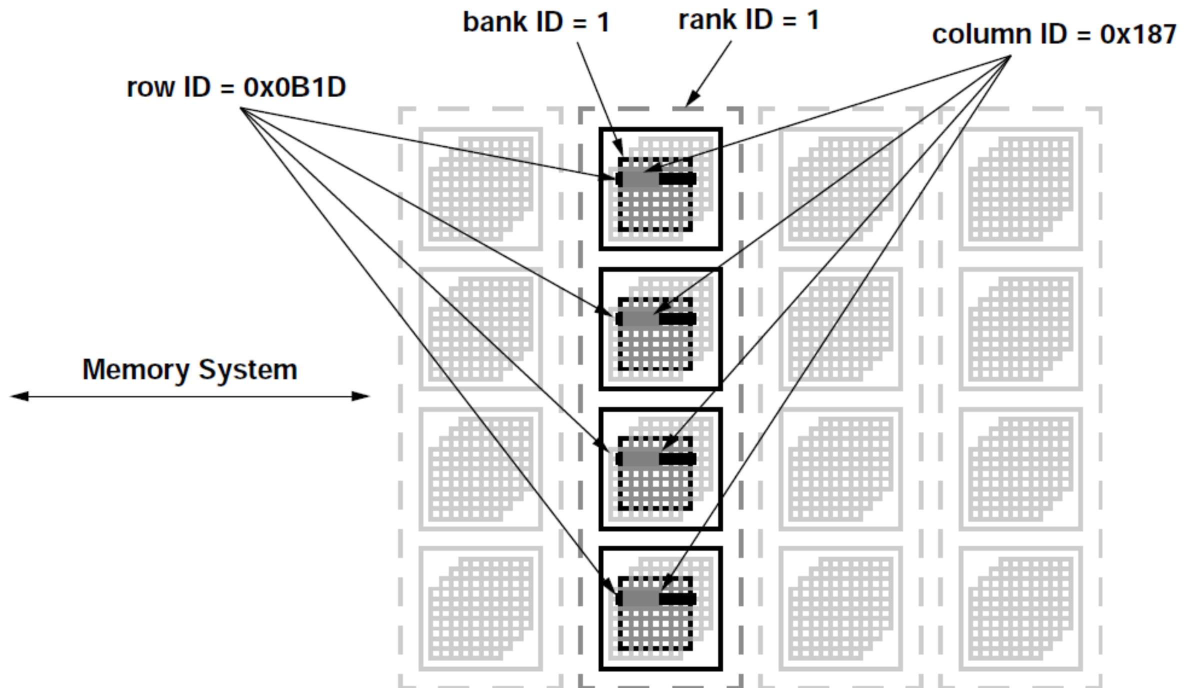

Figure 1-8: An example of parallel access of same bank, row and column in all devices

Table 1-2 lists all the important timings during the DRAM access. Each timing value is technology/industry-specific and can be obtained from the DRAM manufacturer data sheets. For example, a typical 4 GB DDR3 SDRAM by Micron [29] has a t_{RCD} (Row-to-Column command delay) = t_{RP}(Row Precharge) = t_{CAS} ranging from 13.1-15 ns, depending on the value of t_{CK} (clock speed of the SDRAM) device. Typical values of t_{CK} are 1.1 to 1.87 ns, providing data rates from 1066 to 1866 MT/s (**M**ega **T**ransfers per **s**econd, a measurement of bus and channel speed in effective cycles per second, as data is transferred at a higher rate than the clock frequency [30]).

Table 2: DRAM access timings

Parameter	Description
t_{AL}	Added Latency to column accesses, used in DDRx SDRAM devices for posted CAS commands.
t_{BURST}	Data burst duration. The time period that data burst occupies on the data bus. Typically 4 or 8 beats of data. In DDR SDRAM, 4 beats of data 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. Multiple internal bursts are used to form longer burst for column reads. t_{CCD} is 2 beats (1 cycle) for DDR SDRAM, and 4 beats (2 cycles) for DDR2 SDRAM.
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 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_{OST}	ODT Switching Time. The time interval to switching ODT control from rank to rank.
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 time interval between accesses to different rows in a bank. $t_{RC} = t_{RAS} + t_{RP}$.
t_{RCD}	Row to Column command Delay. The time interval between row access and data ready at sense amplifiers.
t_{RFC}	Refresh Cycle time. The 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_{RTRS}	Rank-to-rank switching time. Used in DDR and DDR2 SDRAM memory systems; not used in SDRAM or Direct RDRAM memory systems. One full cycle in DDR SDRAM.
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 read command starts.

1.1.2.1 Sources of Parallelism

Since data is stored in ranks, banks, rows, and columns, it is critically important to understand the sources of Memory Level Parallelism (MLP) available in the DRAM. Each unit is explained briefly here:

- *Channel*: The channel provides the highest level of parallelism in which consecutive cache lines are distributed to different channels, and operations happen in parallel as no timing delay is required between different channel accesses.
- *Rank*: Usually, one to two ranks occupy the same channel. Hence, for ranks residing on the same channel, parallelism is limited by the availability of the bus for the rank to be accessed (command, address, and bus). In SDRAMs, there is a high timing penalty for rank-to-rank switching.
- *Bank:* Bank parallelism suffers from the same bus availability problem existing in the ranks. However, accessing several banks is favorable as no latency is required for sequential commands sent on the bus. Read requests are grouped and sent to different banks to speed up cache line data collection and hide latency associated with row-switching if the data exists in the same bank (i.e. bank conflict). Bank conflicts cause accesses to wait until the bank is free, which reduces bandwidth and increases the energy-per-operation.
- *Row:* In a single bank, only one row can be active. Hence, no parallelism is exploited in row-access. Row-switching is an expensive process; hence it is

favorable to limit the number of row switches, especially with an open-page policy.

- *Column*: Only one column is accessed at a time. No parallelism can be obtained from column accesses. However, it is preferable to satisfy many columns from the same row –especially with an open-page policy.

1.1.2.2 Address Mapping Schemes

In step 3 of the DRAM access operation, the MC relies on the address mapping scheme to decode the received address into the corresponding channel, rank, bank, row and column. It is the process of specifying the number of bits required to decide each level of hierarchy of DRAMs, and then decoding the received physical address according to the order devised. For example, for a DRAM system with 1 channel, 1 rank, 4 banks, 1024 rows and 512 columns, we need: 2 bits for the bank, 10 bits for the rows, and 9 for the columns. The address then can be decided to be *rank***:***bank***:***row***:***column*, or any permutation of them. Fig. 1-9 shows an example of a 32-bit physical address mapped to a single rank, multiple channel system using the following mapping scheme: *row*:*bank*:*col*:*chan*

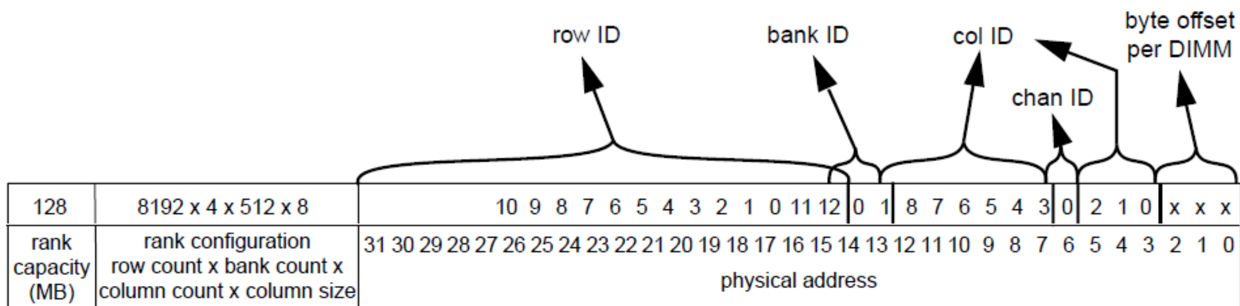

Figure 1-9: Example of mapping physical address into a mapping scheme

Manufacturers of DRAMs rely on the sources of parallelism discussed in the previous section when devising the fixed scheme. For example, banks are chosen to be at lower bit order than ranks to limit rank switching and promote multiple bank accesses. Ranks are chosen to be high to reduce row-switching and make use of data-locality. Columns are chosen to be at the low end to promote the sequential access of data. However, not all applications manifest the same data locality and parallelism, and other schemes *may* fit them better.

The mapping scheme is traditionally considered to be fixed and cannot be *dynamically* adjusted. Our research is the first to explore the effects of dynamically changing the address mapping scheme and studying its impact on both power and performance across a variety of applications.

CHAPTER 2: RELATED WORK

Many DRAM-related studies focused on mainly enhancing DRAMs performance, namely bandwidth and stall time. Some others investigated the power-saving options in DRAMs. Traditionally, DRAMs are not accounted for much of the system's power or heat dissipation (for example, heat sinks are required on DRAMs). However, with the recent advancements in DRAM technology, as in modern DRAMs with inter-chip signaling rates exceeding 1 Gbps/pin, like fully buffered DIMM (FBDIMM), DRAM power consumption becomes an issue [15]. Better utilization of DRAMs affect power indirectly as well by reducing stall time for processers, and hence less power to accomplish a task.

Up to the time of this thesis, no other research investigated the use of dynamic address mapping schemes. A list of related research is reviewed here that focuses on memory controller techniques to meet the aforementioned objectives. They are grouped into the different techniques they belong to.

2.1 Merged Logic DRAMs

Since processors waste large number of cycles transferring data from/to storage units, it became appealing to combine the processing units into the DRAM storage to hide such latency. This approach is referred to as Intelligent Memory (IRAM) [1], or Processors-in-Memory (PIM) [2], through a process that has considerable technological challenges, called Merged Logic DRAM (MLD) [3].

Many approaches followed here in which a combination of a processor, caches, and big DRAM is made on a single chop to be the main compute engine [2, 4, 5]. The problem with this

approach is that it provides incremental speedups, as the DRAM behaves closely to a large on-chip L2 cache. Some other tried to tackle this problem by adding a vector processor [6] or many-cores [5], but such systems became very hard to program.

Another approach is to use MLD to build a specialized engine that can process vector applications [7] or data besides the disk [8]. The third option followed is to replace the memory chips in workstations by a PIM chip to process the most memory intensive applications, like multimedia and data mining [9, 10]. We focus here on a technique that belongs to this approach and called as FlexRAM [12].

FlexRAM is characterized by extracting high bandwidth from the DRAMs using many small processing element embedded in the DRAM units, the ability to run legacy code by using the chip as a regular DRAM unit when recompiling the code is not possible and being a general purpose computational unit –i.e. not bound to specific algorithms. Fig. 2-1 shows the overall system organization of the FlexRam-based memory system.

The system organization of Fig. 2-1 shows a similar hierarchy for a regular computational system. However, the DRAM chips in FlexRAM include an array of small processing engines (PEs), each of 32-bit width and can support integer operations only and share a single multiplier, with small instruction caches but no data caches as they rely on the DRAM row buffers instead. They are grouped into arrays with ring connections to allow neighboring computations, forming Processing Array engines (P. Arrays). P. Arrays operate in Single Program Multiple Data (SPMD) mode for applications that shows high levels of computational parallelism.

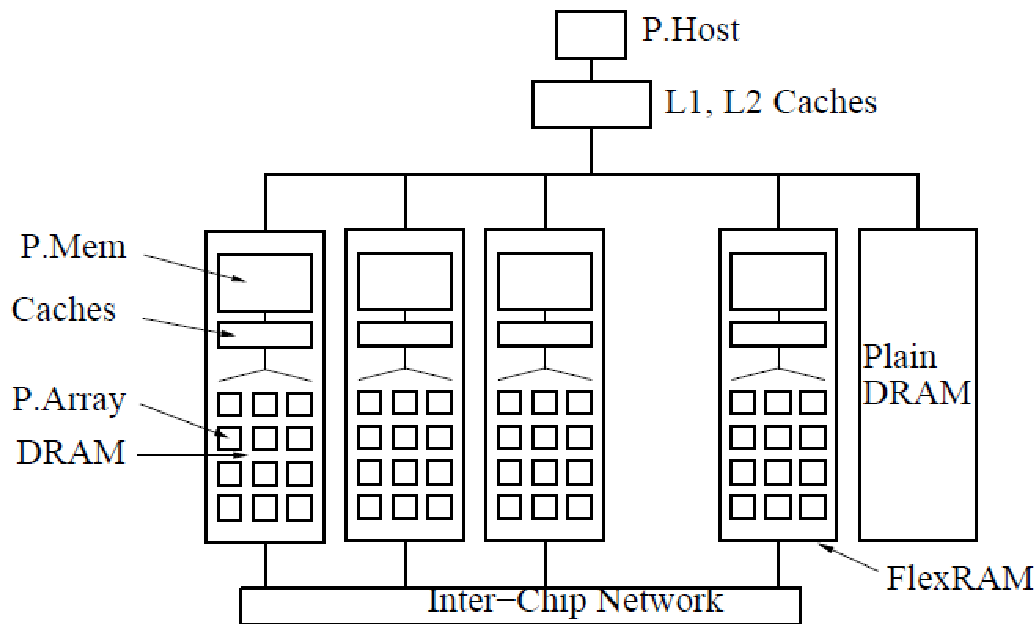

Figure 2-1: Overall organization of a FlexRam-based system

The P. Mem represents a 2-issue superscalar process with floating-point support and has a single data and instruction cache. It's clocked at 400 MHz and supports messaging queues (for message-passing –like parallel computation between the different P.Mem's). There is no support for cache coherence, and programmers are responsible to maintain data integrity when intra-chip level communication happens.

The DRAM unit is made of 64 banks of 1-MB size each, single-ported with the P.Array – which might cause some contention. It uses a Rambus interface as it supports two-way control signal between, which is necessary for the inter-chip communication between the P. Mem and P. Host, and power and ground signaling to allow on-chip processing [12, 13].

Though FlexRAM is intended to be general purpose, it's favorable for memory-intensive applications, such as Data mining (tree generation and deployment, and neural networks),

computational biology and multimedia applications. Communication happens at the inter-chip level in which global communication between P.Arrays on a single FlexRAM chip occur through the P.Mem coordination. This requires that the P.Mem to be able to access each P.Array memory for read/write operations. Inter-chip communication is required when the application can't fit on a single FlexRAM chip, and hence P.Mem units can access memory of other chips by using a plain DRAM storage as no single P.Mem can be the master of the bus.

The FlexRAM is estimated to consume around 36W at the worst case estimate –assuming all blocks are active- with the above configurations, or 0.7W in normal DRAM operation. Testing FlexRAM with several applications characterized by high miss rates, like GTree (Tree generation), DTree (Tree deployment), on BLAST (Protein machine) on a workstation, a speedup between 7and 11 is achieved, with 50noticed on GTree –due to the poor performance on a plain workstation.

Though FlexRAM achieves good speedup results, it requires hand-distribution of the workload on the different P.Arrays and P.Mems. This makes the programming model not very easy. It is very similar to GPU computational models, like CUDA [11], which is more mature, well supported, provides similar speedups, and easier to program.

2.2 Smart Memory Controllers

Other DRAM studies focus on building a smart memory controller design that makes it an active/smart controller, with the objective of reducing the latency incurred by the memory system in applications that exhibit poor locality–i.e. high miss rates-, like sparse matrix codes, applications that heavily use linked-lists, databases, and CAD applications) [16].This ranges

from simple MC design modifications to exploit MLP, like in parallelism-aware batch-scheduling (PAR-BS) [22], to more complicated designs like the Impulse memory controller [23] or the active memory controller [24]

PAR-BS MC design is used in multi-threaded application to preserve each thread's bank-level parallelism through request reordering and batch scheduling. Fig. 2-2 shows how a conventional and PAR-BS scheduler behaves upon receiving requests from different threads. In a conventional scheduler, when T0-Req0 (thread 0, request 0) and T1-Req1 are received to different banks at time t = 1, they are serviced in 1 bank access latency. Then T1-Req0 and T0 – Req1 are serviced at t=2. This causes cores 0 and 1 to wait for the same amount of time before the requested data is received. On the other hand, a PAR-BS scheduler groups such requests into a batch, and implements in-batch request reordering, satisfying all T0 requests (Req0 and Req1) in 1-bank access latency time, and T1 in another cycle.

As noticed in the previous example, request re-ordering preserves the inherent application bank level parallelism, and dividing them into batches divides the problem into finer-grained size to prevent starvation of other requests from different threads. The PAR-BS scheduler is similar to the idea of our research in maintaining MLP, but relies on the inherent parallelism of the application, and does not solve the bank conflict problem among different threads. In fact, such reordering techniques are orthogonal to the dynamic address mapping scheme devised in our research and can be used in conjunction with it to benefit from both.

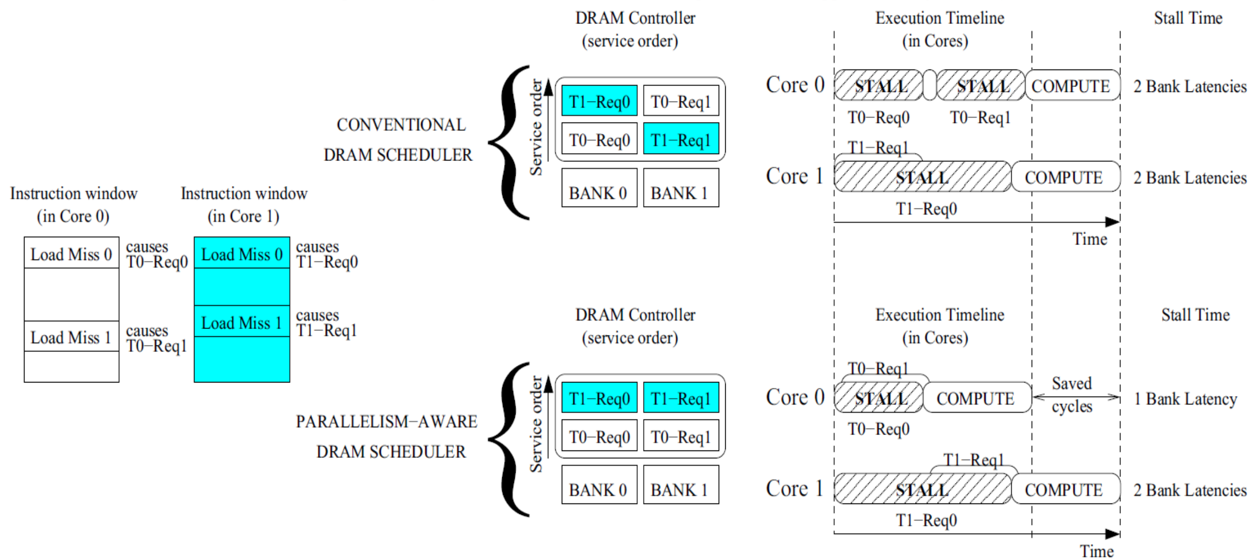

Figure 2-2: Conventional vs. PAR-BS DRAM request scheduling

The Impulse memory controller [23] relies on hiding the latency of DRAM access and reducing the number of transactions (and hence bus bandwidth) in two main ways: it supports and extra stage of address remapping, and performs prefetching of data at the memory controller. Address re-mapping utilizes the *unused* physical address available in the system as a *shadow address space* to be remapped into a real physical address. This is implemented through application (or compiler) and OS modifications, without any hardware changes.

To understand what is meant by unused (or shadow) address space and how it is important to hide memory access latency, consider a system with 4GB address space, with 1 GB available (installed) physical space. The 3 GB represents an unused address space. When an application requires access of diagonal matrix elements (Fig. 2-3), it will cause multiple cache misses that result in fetching the whole row of data into the cache, while only one element (word) is going to be accessed. This results in poor bandwidth utilization as useless data is

transferred across the bus and stored in the caches, through multiple transactions. Impulse MC resolves this issue by allowing the application to re-map the diagonal data in the physical space, pre-fetches and gathers the diagonal elements into a single cache line(s), and transmits them with minimal data bus transactions.

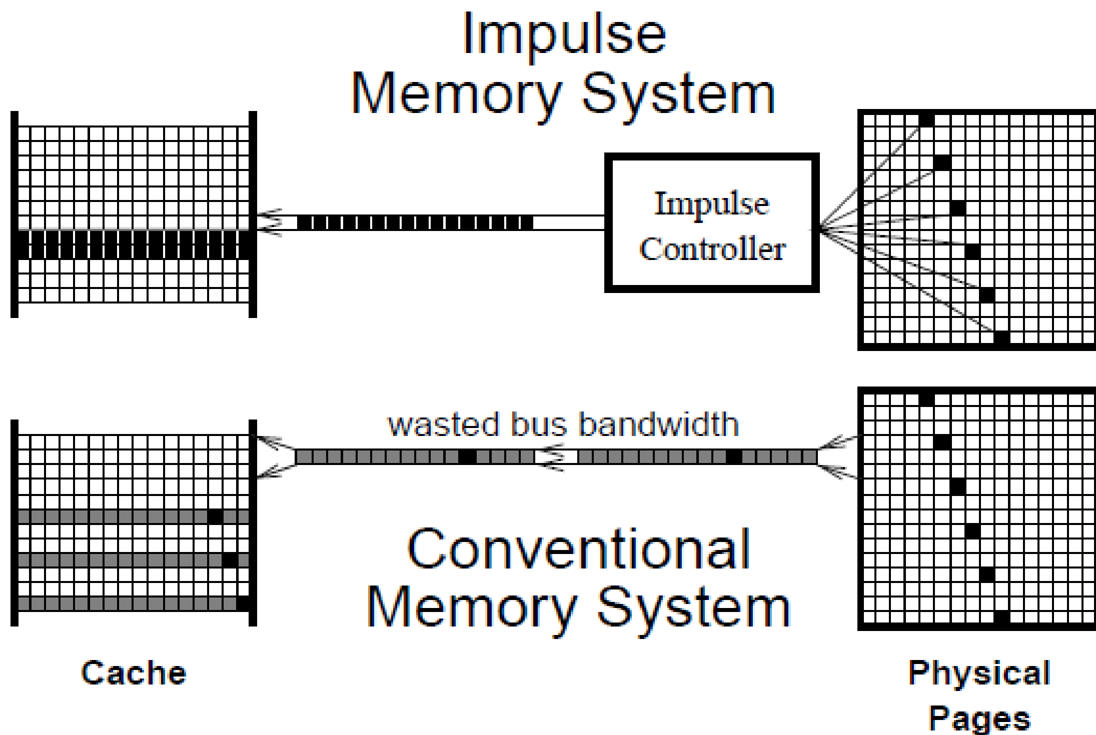

Figure 2-3: Impulse vs. conventional MC data mapping and prefetching

A major problem with Impulse MC is the violation of cache coherence (CC) when allowing the processor to refer to the same data through more than one address. This requires cache flushes whenever correctness is a concern, which represents an expensive overhead especially in the context of multiprocessors.

Another approach followed in [24] addresses this cache coherence problem by extending the coherence protocol already running at the memory controller to selectively invalidate lines to ensure that the processor cache only caches one space at a time. For example in the matrix transpose case, matrix A is transposed in the memory controller into A' and placed in the shadow space. The processor can access both matrices, and their data should be identical. However, if the processor modifies one of them, it should be reflected in the other one –as they are both the same, but laid out differently. The solution here is to allow the processor to cache one of them, and treat any access to the two matrices similar to cache line accesses in multiprocessors with CC, invalidating the normal space cache lines when the shadow matrix is modified. CC is implemented here using an invalidation-based MSI bit-vector directory protocol with released consistency as the base protocol with further extensions specific to the application.

The problem with this technique and the Impulse MC is that it requires manual application (or compiler) and OS (to mediate the address mapping process) modifications. This makes it hard to run legacy code without re-writing it to make use of the MC new features. Moreover, it is useful only for a certain subset of applications that exhibit poor cache behavior.

2.3 **DRAM Power Management**

Some techniques focus on DRAM power management to meet either of the following goals: improve DRAM energy efficiency, or throttling of memory transactions to meet a certain power budget. The latter goal helps reduce system cost by avoiding the provisioning for worst case conditions and supports power shifting [17], a technique directed to maximize the performance of a certain workload by allocating proper budgets to different system components.

Since DRAM banks support going into power-down mode, a proper usage of this functionality can help meet both goals. It can help with energy efficiency by powering-down banks that are not accessed, and activating them when requests arrive. The problem, however, is to know when to transition into/out of the power-down mode as blind switching might impact performance significantly by increasing the latency of memory accesses as explained in the sequence of bank-state transitions in section 1.1.2. The power-down mode can be used as a throttling technique, since memory commands will be forced to wait in the MC. However, deciding a proper *throttling delay*- the period of time during which the memory commands will be blocked- to meet a certain power budget is important and not straight forward.

Much of the work on DRAM power management focuses on mobile embedded systems, like laptops and mobile phones, where a moderate loss in performance is acceptable in return for longer battery life [18]. Delaluz et al. [19] uses an idle-duration time-predictor for the power-down duration of the memory devices. Results were good on cacheless systems using Rambus DRAM. The work was then extended by Fan et al. [20] to use multi-level caches. The problem with such approaches is that it relies on a threshold value and it is both application and system dependent.

Throttling of DRAMs was first proposed by Felter et al. [17], while Diniz et al. [21] presents a set of throttling techniques with less performance degradation. Ibrahim et al. [25] proposes a comprehensive set of techniques that target a rank's power-down and throttling timings through a simple power-down policy and a modified Adaptive History-Based (AHB)

scheduler. Results show an increase in energy efficiency that ranges from 11.6-40% on different Stream, NAS and SPEC2006fp benchmarks.

The rank power-down policy relies on a status bit and a counter at the hardware-level per rank. The status bit is cleared when the rank is in non-power-down mode, and set otherwise. Whenever the rank is accessed, the counter is reset to a certain value and decremented every cycle. When the counter reaches zero, with the status bit being cleared, and no command in the command queue has the desired rank as the target memory device, then the rank is allowed to enter the power-down mode. The rank is then powered-up by checking the commands entering the command queue for commands targeting the rank itself.

CHAPTER 3: RESEARCH DESIGN AND METHODOLOGY

To study the feasibility of dynamic address mapping schemes, we ran several initial experiments on the DRAMSim2 simulator -discussed in Section 3.2- using sample application traces. Results showed that traces behave differently on different mapping schemes, and that MLP can be improved when using an application-specific mapping scheme. The idea of using dynamic address mapping schemes is described here with a brief introduction about the simulator we use and a detailed explanation about metrics and experimental setup.

3.1 **Motivation**

As seen in the related work, most of the techniques focusing on improving DRAM performance were either application/compiler-based or were optimizations to the scheduler running on the MC that reorders requests according to certain novel scheduling policies. Though some relied on the OS to define the page allocation and virtual address mapping [23], no direct control on data layout in DRAM memory was ever made.

O. Mutlu [22] tried to utilize bank-level parallelism in memory requests from different threads through a simple memory request batch scheduling technique. However, it relies on the inherent parallelism of the application –i.e. assuming back-to-back requests are distributed across different banks. What if data is laid out in a way that caused strided access to the same bank? This could happen when large data (in powers-of-two are accessed concurrently). For example, assume the problem of adding two arrays, A[i], and B[i] and storing the result in C[i]. If they are allocated contiguously in memory at an integer multiple or row-size address boundaries, they

might be allocated to different rows in the same bank. This problem is referred to as Bank Address Aliasing (BAA), or simply Bank Conflict.

This motivated us to study data layout in memory at a lower level by controlling the address mapping scheme –which has always been assumed to be fixed [26]. But how would the same application perform when used on the same DRAM but with a different address mapping scheme? What is the best address mapping scheme that could provide the highest bandwidth utilization and energy efficiency? Could the default address mapping scheme used in industry be the best compromise for all applications?

All of the above were questions that led us to study the feasibility of varying the address mapping scheme as a technique to alter data layout in the DRAM memory for different applications in a way that will minimize bank conflicts and improve both performance and energy efficiency. We ran several experiments to verify the performance variability of applications using different device address mapping scheme, and we propose a simple design to enhance the performance of a multi-threaded/core system using per-thread/core address scheme mapping. The simulator we use is introduced in the next section followed by the experimental setup. Results of the experiments are reported in Chapter 4.

3.2 DRAMSim2 Simulator

For this study to be possible, we needed a cycle-accurate simulator that accurately models performance and power of DRAM memory. The modeling of a DRAM system is not a straight - forward task as it depends on many variables, like the memory-system architecture and configuration, MC access protocol, and DRAM device timings. Therefore, most processor

simulators (like SimpleScalar, SESC) model main memory access as a fixed-latency operation. This has the side-effect of ignoring the effect of memory and bus contention and optimization techniques on the processor execution time.

The Memory-Systems research group at the University of Maryland developed DRAMSim2 [27], a cycle-accurate simulator that models DRAM devices, MC and buses accurately. The simulator is flexible enough to configure the system and DRAM device configuration independently. The *system* configuration mainly defines the MC access protocol with basic system variables, like bus width, row buffer page-policy, power-saving mode, scheduling policy and address mapping scheme. On the other hand, DRAM device configuration defines the device device-specific parameters, like capacity -through number of banks, rows, columns, and device width-, device clock speed, timing parameters and electrical-current at different modes of operation. The device configuration parameters are industry/technology specific, and the group made sure to provide real values that match the latest commodity DDR3 SDRAMs by Micron [29, 30].

DRAMSim2 can work in stand-alone mode, in which it simulates memory system traces (reads and writes with timestamps). Though such traces do not provide accurate modeling of a CPU system driving the main memory, it is convenient enough to validate memory-system design changes. Moreover, traces are easy to produce from any CPU simulator by logging the memory access stage in a format readable by DRAMSim2. DRAMSim2 can also be built as a dynamic-shared library to be connected to other CPU simulators. Fig. 3-1 shows a top-level memory-system functional flow diagram, and how it can be hooked to a CPU simulator.

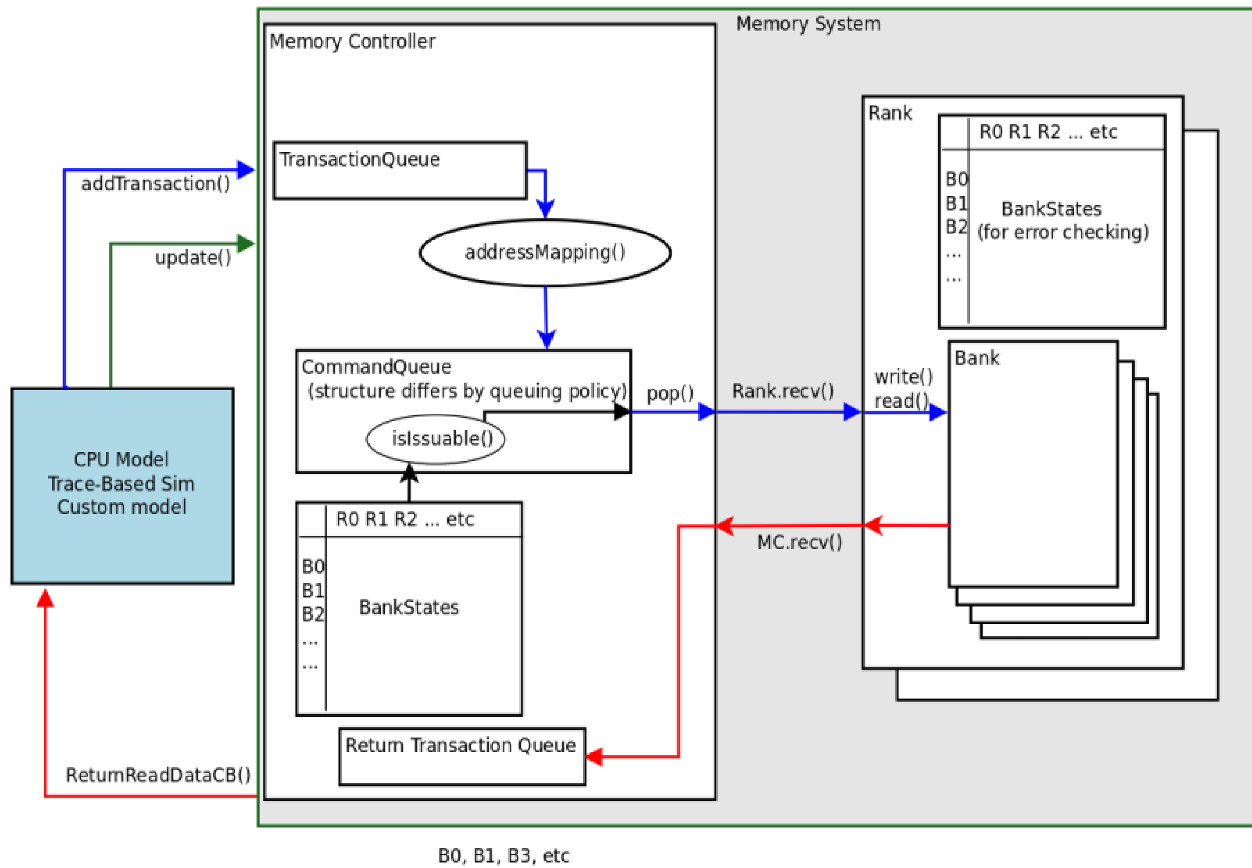

Figure 3-1: DRAMSim2 Functional Flow Diagram

After running DRAMSim2 on a memory trace, it reports different results at each N-cycle epoch (defined in the system.ini file), such as per-bank access time and bandwidth, aggregate bus bandwidth, number of transactions , and system power-breakdown.

We have modified the simulator to show peak-bus bandwidth utilization, per-bank power consumption, and the energy/operation. These metrics are defined in the next section. Moreover, we added full multi-threaded application support to the simulator that allows it to identify memory traces from different threads and facilitates studying per-thread optimization techniques. Fig. 3-2 shows a sample snapshot of the simulator output. A PERL parser script is developed to

parse relevant data from different epochs of the simulator output so that they can be used in the analysis and evaluation of proposed technique.

```
Total Return Transactions: 2215 (141760 bytes) aggregate average bandwidth
8.802GB/s ==> 82.515% of the peak bandwidth (10.667GB/s)

  -Rank   0 :
      -Reads  : 1996 (127744 bytes)
      -Writes : 219 (14016 bytes)

      -Bandwidth / Latency  (Bank 0): 1.105 GB/s  4055.833 ns |  Power Stats
(watts): BG=0.126, BURST=0.221, RFRSH=0.003, ActPre=0.105, TotAvg=0.455

      -Bandwidth / Latency  (Bank 1): 1.097 GB/s  4120.113 ns |  Power Stats
(watts): BG=0.128, BURST=0.230, RFRSH=0.003, ActPre=0.107, TotAvg=0.468

      -Bandwidth / Latency  (Bank 2): 1.105 GB/s  4064.201 ns |  Power Stats
(watts): BG=0.127, BURST=0.220, RFRSH=0.003, ActPre=0.107, TotAvg=0.457

      -Bandwidth / Latency  (Bank 3): 1.101 GB/s  4095.115 ns |  Power Stats
(watts): BG=0.127, BURST=0.231, RFRSH=0.003, ActPre=0.107, TotAvg=0.469

      -Bandwidth / Latency  (Bank 4): 1.105 GB/s  4063.586 ns |  Power Stats
(watts): BG=0.127, BURST=0.220, RFRSH=0.003, ActPre=0.107, TotAvg=0.457

      -Bandwidth / Latency  (Bank 5): 1.101 GB/s  4069.324 ns |  Power Stats
(watts): BG=0.127, BURST=0.232, RFRSH=0.003, ActPre=0.105, TotAvg=0.467

      -Bandwidth / Latency  (Bank 6): 1.101 GB/s  4059.807 ns |  Power Stats
(watts): BG=0.127, BURST=0.220, RFRSH=0.003, ActPre=0.105, TotAvg=0.456

      -Bandwidth / Latency  (Bank 7): 1.089 GB/s  4082.673 ns |  Power Stats
(watts): BG=0.128, BURST=0.229, RFRSH=0.003, ActPre=0.105, TotAvg=0.465

== Power Data for Rank        0
Average Power (watts)     : 3.757
 -Background (watts)     : 1.080
 -Act/Pre    (watts)     : 0.846
 -Burst      (watts)     : 1.804
 -Refresh    (watts)     : 0.028
 -Energy/Op    (nJ)     : 25.445
```

Figure 3-2: Sample DRAMSim2 Epoch Output

3.3 **Experimental Setup**

We performed several experiments to examine the effectiveness of dynamic address mapping schemes in enhancing application performance and energy efficiency. However, to quantize performance and energy, we have to use proper metrics to evaluate them. Bandwidth (peak bandwidth utilization), and energy/operation are the main metrics we use (section 3.3.1).

Since the power of variable mapping schemes relies on MLP (mainly bank-level parallelism), the problem is to find memory traces that exhibit the highest level MLP matching a certain scheme, and then run it on several other DRAM mapping schemes. To do so, we built a *synthetic trace generator* to produce interleaved-bank memory accesses according to the device configuration and the desired mapping scheme. Several scheme-matching traces are then mixed in a proper way to simulate multithreaded applications through the *synthetic trace generator*.

The above steps are necessary to quantify the promise of dynamic address mapping schemes, by measuring the loss incurred in performance and energy efficiency when limiting ourselves to a single address mapping scheme. However, real applications are the judge as they might show variable levels of MLP at different run epochs, or exhibit better performance and power results when using the default mapping scheme only. Several benchmarks should be examined like BLAS, multimedia applications (MP3, H.264), and some from SPEC2006 by generating their memory traces and checking how dynamic address mapping schemes can enhance their performance and energy efficiency. This will be an essential part of the future study explained in Chapter 5.

3.3.1 Metrics

There are many units of measure that can be used to evaluate DRAMs' performance and energy consumption, like number of operations (reads/writes), bandwidth, and total power consumption. However, since different DRAM devices vary in clocking frequency and technology used (DDR2, DDR3, FB-DDRAM, Rambus…etc.), we need to make sure to use normalized metrics to be able to make fair comparisons.

For example, DRAM peak bandwidth is calculated by:

$$\boldsymbol{BW_{Peak} = Data\ transferes \times BusWidth \times Device\ Frequency} \quad (3.1)$$

Equation 3.1 represents an optimistic assumption that the DRAM is keeping the bus busy at every cycle, not to mention that the parameters are technology dependent, and relying on the peak bandwidth value will not give an insight about the efficiency of the optimization techniques used. $BW_{\boldsymbol{eff}}$ is calculated during run-time as the aggregate banks' bandwidth Therefore, **peak bandwidth utilization** (BW_{Peak_Util}) is used as a better metric to evaluate pefromance. It is defined by:

$$\boldsymbol{BW_{Peak_Util} = BW_{eff}/BW_{Peak}} \quad (3.2)$$

BW_{eff} is evaluated during runtime, while BW_{Peak} is calculated using equation 3.1. $\boldsymbol{BW_{Peak_Util}}$ is a good measure of performance as any loss in MLP will be manifested in smaller utilization values, irrespective of the technology used, and hence can be used to evaluate the efficiency of the tested technique in enhancing performance.

Relying on total power consumption of DRAMs is tricky. For example, assume an 8-bank DRAM unit. Two extreme cases are possible for memory accesses:

1. Single Bank access: All memory requests will be satisfied from a single bank.
2. Interleaved bank access: Memory requests are distributed over all banks.

For a fixed number of cycles, the total power consumed will *always* be higher for case 2 as more devices will be active. However, this is natural as more work is done in case 2 than 1 (higher number of memory requests are satisfied). Therefore, total power consumption alone is insufficient to give an insight about energy efficiency.

For a fixed number of cycles, power consumption reduces to energy (as power is energy/time), and what matters is how much energy is consumed to satisfy *x* number of operations. This leads to the usage of Energy per operation metric:

$$\boldsymbol{Energy/Op} = \frac{\boldsymbol{P}}{\boldsymbol{N_{Ops} \times f}} \qquad \textbf{(3.3)}$$

Where P represents the power consumed, f is the device frequency, and N_{Ops} is the total number of operations (reads and writes) accomplished. All experiments below rely on BW_{Peak_Util} and $Energy/op$ as the metrics to measure performance and energy efficiency.

3.3.2 Scheme-Matching Synthetic Trace Generator

To study the effectiveness of bank-parallelism, it is important to understand the maximum utilization of such a technique. In other words, we do not expect a memory trace with perfectly-interleaved memory accesses that matches some address mapping scheme to provide 100% bandwidth utilization. This is because DRAM operations are more involved than that. For

example, refresh cycles are required periodically, during which the data bus cannot be utilized to satisfy requests. This impacts BW utilization, and it is important to understand the *realistic* maximum values that can be obtained for bandwidth utilization and energy/op using some *synthetic* trace that manifests the interleaved access. Moreover, applications show different levels of data locality/allocation and miss-rates, and hence using one application alone may not be conclusive about the effectiveness of a using dynamic mapping schemes.

This leads us to the conclusion that having multiple synthetic scheme-matching traces is a necessity for this research so that each can be studied independently on different schemes to understand the extreme cases of achievable gains (when running on the matching scheme) and potential losses (when running on other schemes)..

A PERL script is developed that enables us to generate such traces. The script (called `generate_trace.pl`) takes the following arguments:

- Scheme: The scheme for which the generated trace will provide a perfectly interleaved bank accesses.
- Page policy: Whether to provide requests that are optimized for open page (high locality) or closed page (low locality). In other words, it controls whether requests will cause same-row hit or different row hits in the same bank.
- Read/write ratio: The ratio of read-to-write requests in the whole trace. This is important to study the impact of read/write-heavy traces on performance and energy.

- Device configuration: The DRAM memory unit configuration used. It is provided as a colon delimited list of the number of: Channel:Rank:Bank:Row:Column
- Number of requests.
- Output file name.

3.3.2.1 Synthetic Trace Mixer

A single scheme-matching memory trace is useful for a single-threaded study. Since our study here focuses on multi-threaded applications, a memory trace made up of a mix of two or more scheme-matching traces will be useful here. This new trace can be considered as either a single threaded application that has different access modes (a relaxed version of a single matching-scheme trace, and closer to real applications), or a multi-threaded application.

Studying such trace mixes helps in comparing the performance and energy efficiency of using a single scheme configuration (the classical DRAM configuration) vs. dynamic scheme configurations on traces /threads of variable data layout requirements. Therefore, another PERL script was generated, called `mix_traces.pl` that generates such a trace according the following input parameters:

- Traces' list: List of Trace1 and Ttrace2 generated by `generate_trace.pl`.
- Mix ratio: The ratio at which both traces will be mixed together.
- Number of requests
- Output file name

3.3.3 Simulation Configuration

The DRAMSim2 simulator is set up through all experiments to use a matching commodity DRAM device configuration. A DRAM memory module of 4 GB is used with the timing and electrical parameters that match the latest widely-used Micron DDR3 SDRAM unit [29]. Table 3 and 4 list the device configuration used for the experiments in Chapter 4 with the timing and electrical parameter values that are obtained from the datasheet of the manufacturer. For example, the clocking speed of the device is chosen to be $(1/t_{CK})$ 666.7 MHz, offering data transfer rate of 1333 MT/s.

Table 3: Device Configurations and Electrical Parameters

Device Configuration	
Channels	1
Ranks	1
Banks	8
Rows	32768
Columns	2048
Device Width	8
Page Policy	Open Page
Total Capacity	4 GB
Electrical Parameters	
Parameter	**Value (mA)**
V_{dd}	1.5 V
I_{DD0}	130
I_{DD1}	155
I_{DD2P}	10
I_{DD2Q}	70
I_{DD2N}	70
I_{DD3Pf}	60
I_{DD3Ps}	60
I_{DD3N}	90
I_{DD4W}	300
I_{DD4R}	255
I_{DD5}	305
I_{DD6}	9
I_{DD6L}	12
I_{DD7}	460

Table 4: Timing parameters for the used device in simulations

Parameter	Value (ns)	Description
t_{REF}	7800	Refresh Period
t_{CK}	1.5	Device Clock Period
BL	8	Burst Length
t_{RAS}	24	Row-Access-Strobe: Time between row access command and data restoration. No pre-charging is possible until t_{RAS} passes after last bank activation
t_{RCD}	10	Row-to-Column delay: Time between row access and data being ready at the sense amplifiers
t_{RRD}	4	Row-Activation-to-Row-Activation: Min. time between two row activation commands to the same DRAM. Limits peak current profile
t_{RC}	34	Row Cycle: Time between accesses to diff. rows in a bank = t_{RAS} + t_{RP}
t_{RP}	10	Row Precharge: Time to precharge DRAM array for another access
t_{CCD}	4	Column-to-Column Delay: Min. column command timing. 2 beats (1 cycle) for DDR, 4 beats (2 cycles) for DDR2
t_{RTP}	5	Read-to-Precharge: Time to precharge array for another row access
t_{WTR}	5	Read-to-Write: Min. time between end of data read burst and start of column-read command.
t_{WR}	10	Write-Recovery: Min. time between end of data read burst and precharge command
t_{RTRS}	1	Rank-to-Rank-Switching time
t_{RFC}	107	Refresh-Cycle-Time: Time interval between refresh and activation command
t_{FAW}	20	Four (row) bank Activation Window: rolling time-frame in which a max. of 4 bank activation can be engaged. Limit current profile for DDR2/3 with > 4 banks
t_{CKE}	4	Defines next power up of an idle device
t_{CMD}	1	Command-Time: Time it takes the command to transport from the controller to the DRAM

3.3.4 Dynamic Address Mapping Scheme Design

As discussed in Chapter 1, memory controllers are always configured to use one default address mapping scheme. This mapping scheme is not fixed, in the sense that it can never be changed once the MC is manufactured. The mapping scheme to be used can be configured through specific configuration bits in the MC, as in the Intel 82955X Memory Controller [32], to support configuration flexibility when different device configurations are used. However, once it is set, it cannot be changed at run-time.

A simple MC design modification is proposed in this research that allows using multiple mapping schemes during run-time. To understand it, consider the example of using two mapping schemes, SchemeA and SchemeB. Modifying the MC to split the address space in main memory into equal portions (half in this example), in which each uses a different mapping scheme to allocate data. This can be done easily in hardware through a simple lookup table that defines the address boundaries and address mapping scheme for each portion of main memory.

However, once a memory request arrives to the MC, it needs to know which address mapping scheme to use for address translation. We suggest a simple static method to achieve per-thread address translation as shown in Fig 3-3. Following a similar method followed in [22], additional bits are added to each memory request, identifying the source thread-id. Each thread should be assigned to a mapping scheme that provides higher performance and energy efficiency (defined through application profiling). Therefore, whenever a memory request arrives to the MC, the requesting thread is identified, and hence the proper mapping scheme is used. This suggestion is invisible to the programmer, and relies on hardware modifications only, but might

be restrictive for applications that require larger memory sizes than available in the allocated memory portion. To relax this restriction, the memory-hungry thread can be allowed to access the other address mapping scheme memory portion, at the expense of losing performance as non-matching address mapping scheme will be used. For that to be feasible, the OS needs to be aware of the memory partitioning so that it can allocate pages of memory for the applications in the memory portions that use a better mapping scheme for the application

3.3.5 Set of Experiments

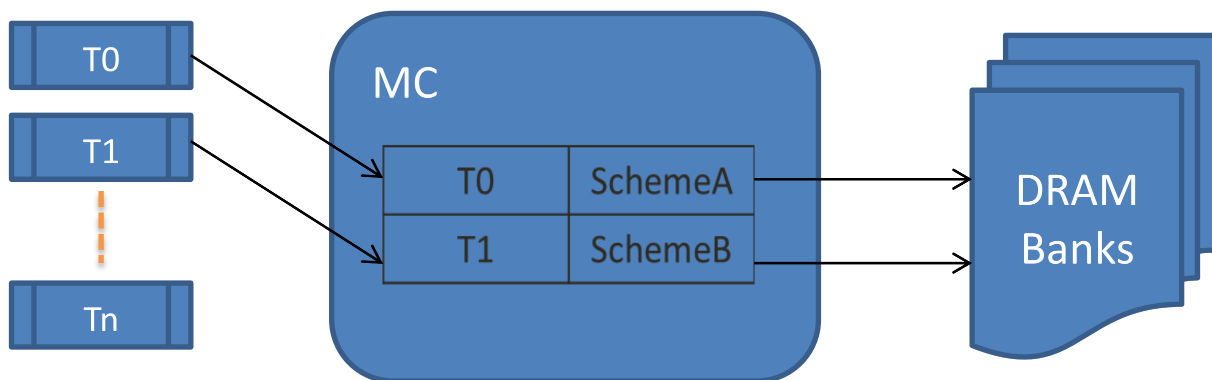

Figure 3-3: Per-thread address mapping

To test the impact of mapping schemes on performance and energy/op, we configure the simulator -explained in Section 3.3.3-, and run several experiments using 4 different mapping schemes. We limit ourselves to 4 schemes as they cover the wide range of the 120 possible address mapping schemes by studying intermediate cases. One of the schemes (SchemeB) is the default scheme widely used as the base scheme in current commodity Memory Controllers [31] . Each mapping scheme is described using the unit mapping position –as explained in section 1.1.2.2, and provided in Table 5.

Table 5: Address mapping schemes studied

Mapping Scheme	Address Resolution
SchemeA	Channel : Rank : Bank : Row : Column
SchemeB	Channel : Row : Bank : Rank : Column
SchemeC	Channel : Bank : Rank : Row : Column
SchemeD	Channel : Row : Rank : Bank : Column

Using the above mapping schemes, we conduct the following set of experiments:

- *Single synthetic trace on a single mapping scheme*:

  A synthetic scheme-matching trace is generated, and then tested over each mapping scheme independently. The purpose of the experiment is to examine the performance and energy efficiency on the matching-scheme compared to the other schemes, and measure the impact in best- the and worst-case scenarios. Moreover, it provides a measure of the sensitivity of the scheme-matching trace to the different schemes.

- *Single mapping scheme vs. dynamic mapping scheme*:

  We conduct a set of experiments using *synthetic trace mixes* Basically, a set of mixed synthetic scheme-matching traces –with different ratios of reads to writes- are tested on each mapping scheme individually, and then on a dynamic mapping scheme configuration.. It measures the performance improvement and the gain in energy efficiency that can be achieved using dynamic address mapping, and hence allowing us to make a fair evaluation of dynamic address mapping schemes.

CHAPTER 4: RESULTS AND ANALYSIS

In each experiment described in Chapter 4, we conduct a 1,000,000-cycle simulation, in which we group and log data at epochs of 100,000 cycles. Using the PERL script explained in section 3.2, we parse the logs and analyze collected data thoroughly. This chapter is organized in sections corresponding to the experiments conducted, presenting the data collected followed by a discussion of the results obtained.

4.1 Single-Scheme Device with Scheme-Matching Synthetic Trace

We generate a single SchemeB-matching trace with different read/write ratios. We use a scheme-matching trace to obtain higher bandwidth and energy efficiency -by utilizing high bank-level parallelism-, while the various read/write ratios help us understand their effect on performance and energy consumption. This is because different device timing constraints are required when executing various memory operations. For example, a write-to-read delay (t_{WTR}) is required after executing a write-after-read memory access, which will affect performance and energy efficiency. Fig. 4-1 and 4-2 summarize the results of this experiment.

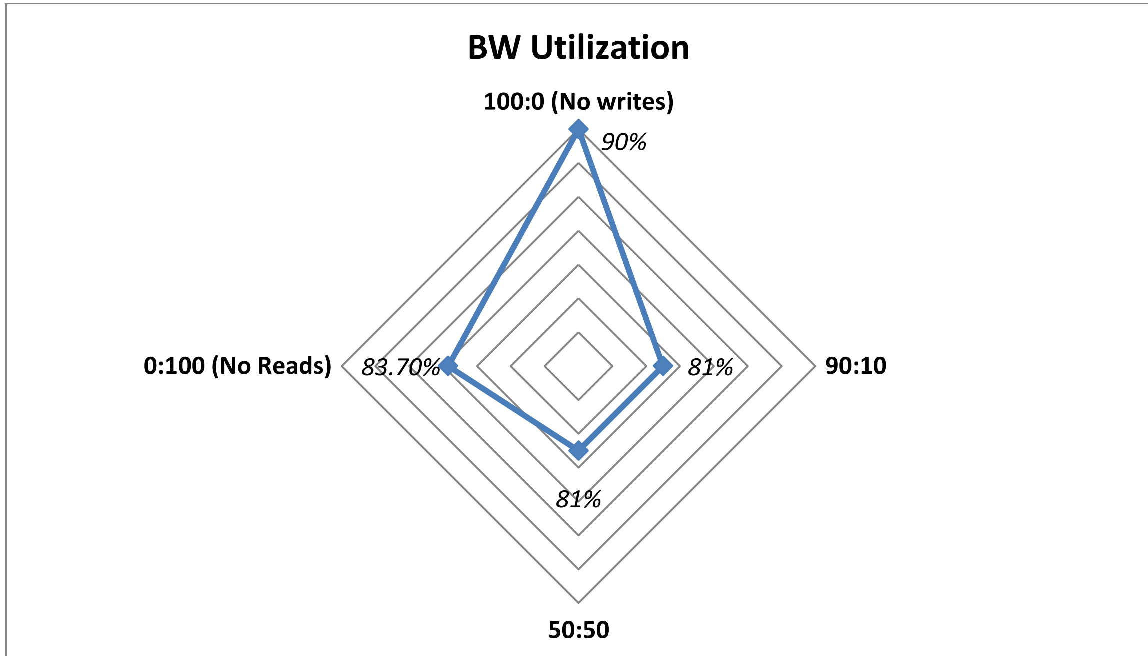

Figure 4-1: Bandwidth utilization of SchemeB-matching trace with different read/write ratios

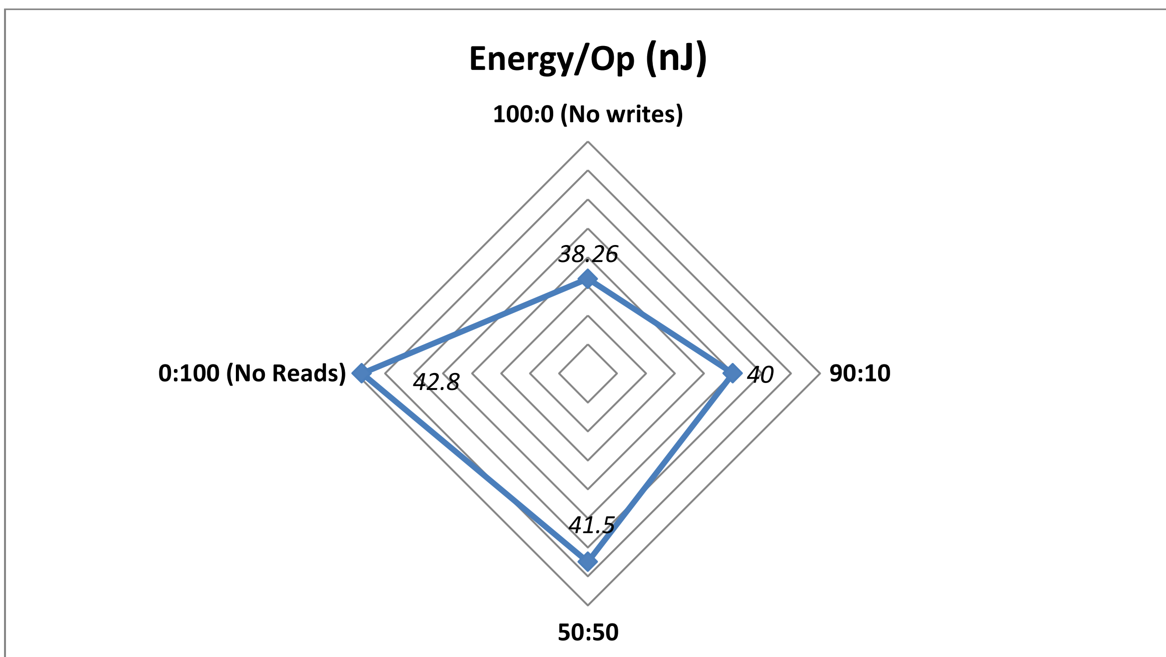

Figure 4-2: Energy/Op of SchemeB-matching trace with different read/write ratios

Results show that we achieve the highest performace (91% BW utilization) when the trace includes no writes at all. Multimedia traces are a good example of read-only applications. Increasing the write ratio decreases performance slightly, but maintains a stable 81% BW utilization. Though read/write ratios are application dependent, a sample study on various application memory traces showed aread operation percentage of 90-95% ,. *Therefore, in all subsequent synthetic trace runs, 90:10 read:write ratio will be adopted*.

Studying the energy/op results shows another interesting prespective of read/write ratio impact on energy efficiency. Smaller values in Fig. 4-2 means higher energy efficiency. It is noticeable from the graph that the energy/op is proportional to the number of writes. Therefore, we can make the conclusion that applications that exhibit high levels of bank-access parallelism will provide good bandwidth utilization, regardless of the read/write ratio, but with lower energy cost for read-dominant traces.

Table 6 and 7 present the bandwidth utilization of running four different scheme-matching traces on four different single device mapping schemes. Fig. 4-3 and 4-4 present a diagram of the values reported in these tables.

Table 6: Bandwidth Utilization

		Device Scheme			
		SchemeA	**SchemeB**	**SchemeC**	**SchemeD**
Trace Scheme	**SchemeA**	81.14%	35.50%	81.14%	35.50%
	SchemeB	35.50%	81.14%	35.50%	81.14%
	SchemeC	81.14%	35.50%	81.14%	35.50%
	SchemeD	35.50%	81.14%	35.50%	81.14%

Table 7: Energy/Op

		Device Scheme			
		SchemeA	**SchemeB**	**SchemeC**	**SchemeD**
Trace Scheme	**SchemeA**	25.757	34.952	25.757	34.952
	SchemeB	34.952	25.757	34.952	25.757
	SchemeC	25.757	34.952	25.757	34.952
	SchemeD	34.952	25.757	34.952	25.757

Results conform to the intuition that scheme-matching traces manifest the highest levels of performance and energy efficiency when running on their corresponding scheme device (the highlighted diagonal entries of the table). Moreover, some device schemes result in massive performance degradation (around 50%) and reduce energy efficiency when running non-matching traces on them, such as SchemeA-matching trace on the SchemeB device or SchemeB-matching trace on the SchemeA device. *This indeed proves that variable device mapping schemes have a direct impact on performance and energy efficiency of the running application*.

An interesting observation can be made from the results is that other device schemes provide almost equivalent performance and energy efficiency for other scheme-specific traces.

For example, SchemeA trace on the SchemeC device, or SchemeB trace on the SchemeD device show comparable values to running SchemeA trace on the SchemeA device and SchemeB trace on the SchemeB device, respectively. *Therefore, we can make the conclusion that different mapping schemes could be grouped together*, and hence it is sufficient to reduce further studies on the four schemes into two schemes only, SchemeA and SchemeB.

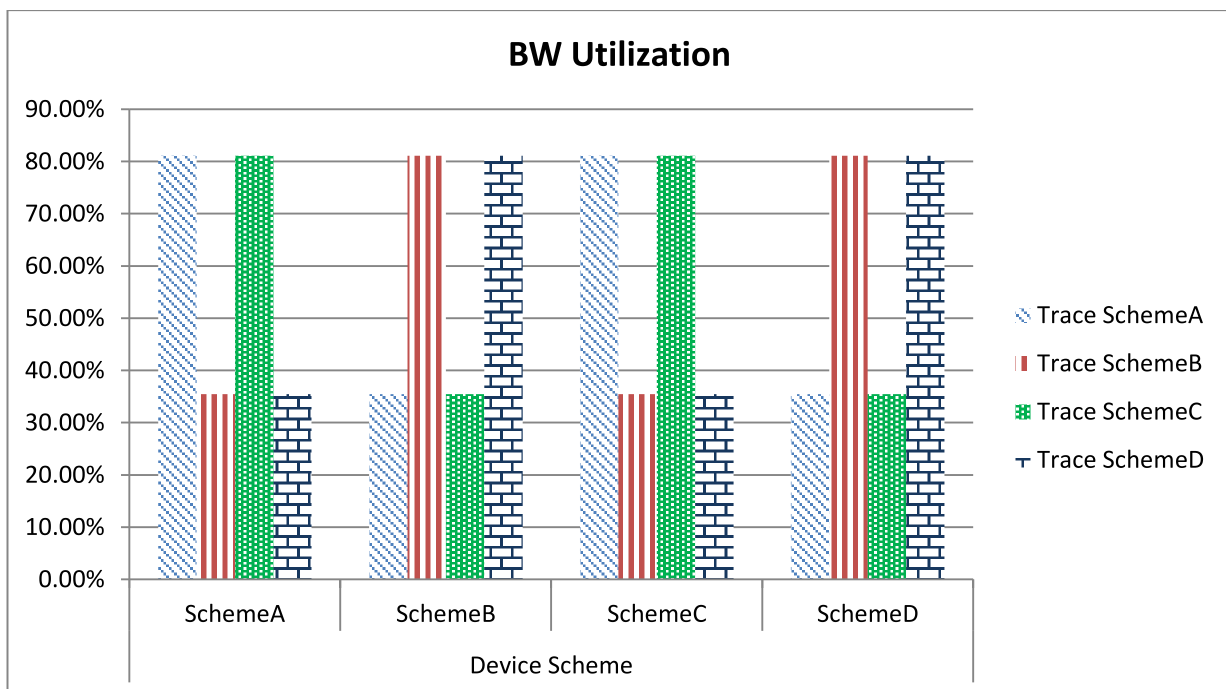

Figure 4-3: BW utilization for different scheme-matching traces on different single-scheme device configuration (*higher is better*)

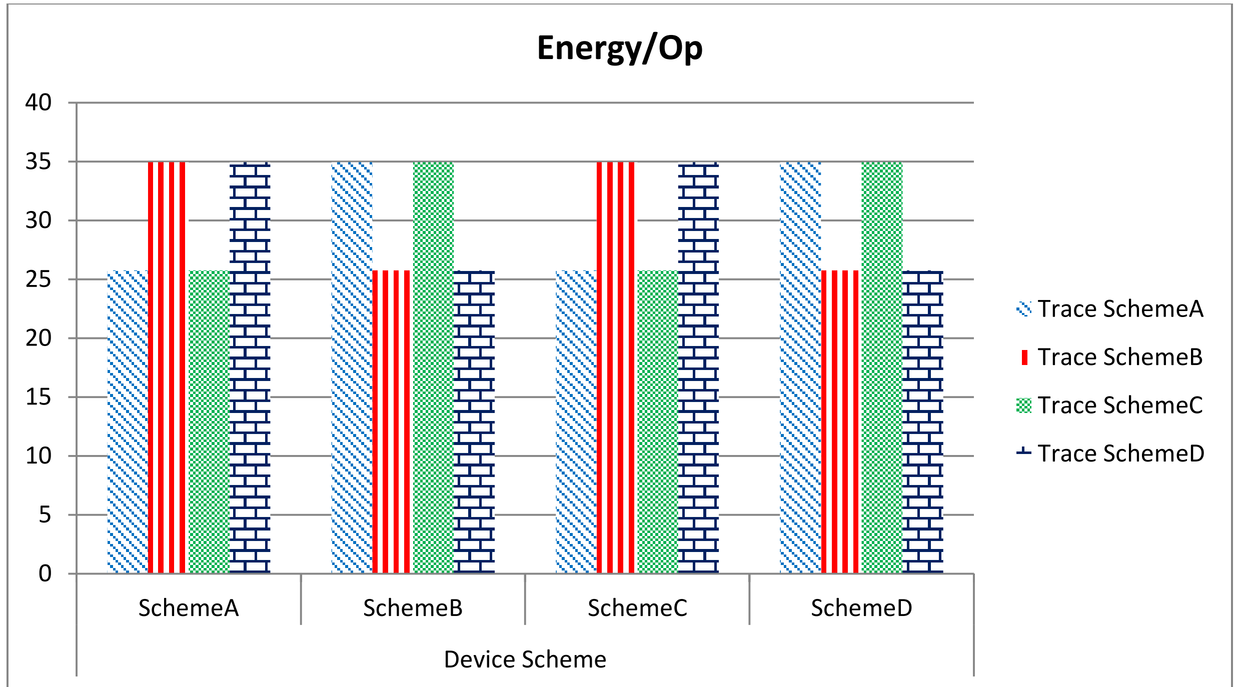

Figure 4-4: Energy/Op for different scheme-matching traces on different single-scheme device configuration (*lower is better*)

4.2 Single-Scheme Device with Multi-Threaded Synthetic Traces

Using the 90:10 read:write ratios to generate SchemeA and SchemeB matching traces, they are then mixed together to form two traces that differ in the mixing ratio: 90:10, and 50:50, in which SchemeB is the base scheme. These two traces were then tested on a single scheme device, one configured with SchemeA and the other with SchemeB for a total of four simulations. To ease data representation, they are denoted as follows:

- SchemeA_device_on_schemeB_A_trace_mix_90_10
- SchemeA_device_on_schemeB_A_trace_mix_50_50
- SchemeB_device_on_schemeB_A_trace_mix_90_10
- SchemeB_device_on_schemeB_A_trace_mix_50_50

Results are reported in Fig. 4-5 and 4-6 at different epochs, as described at the beginning of this chapter. Again, it is consistent with the conclusion made about single-trace runs, in which the same traces manifest different performance and energy efficiency values when running on different device scheme configurations. This can be noticed from the huge performance gap (around 2x BW utilization is lost) noticed in running schemeB_A_mix_90_10 trace on SchemeB vs. SchemeA device. Moreover, the performance of schemeB_A_mix_50_50 on different schemes lies in the same envelope (60% BW utilization), regardless of the device scheme used, and between the highest (82% BW utilization) and lowest (38% BW utilization) values achieved. This hints the linear performance gain when moving from the least-matching to the best matching device scheme.

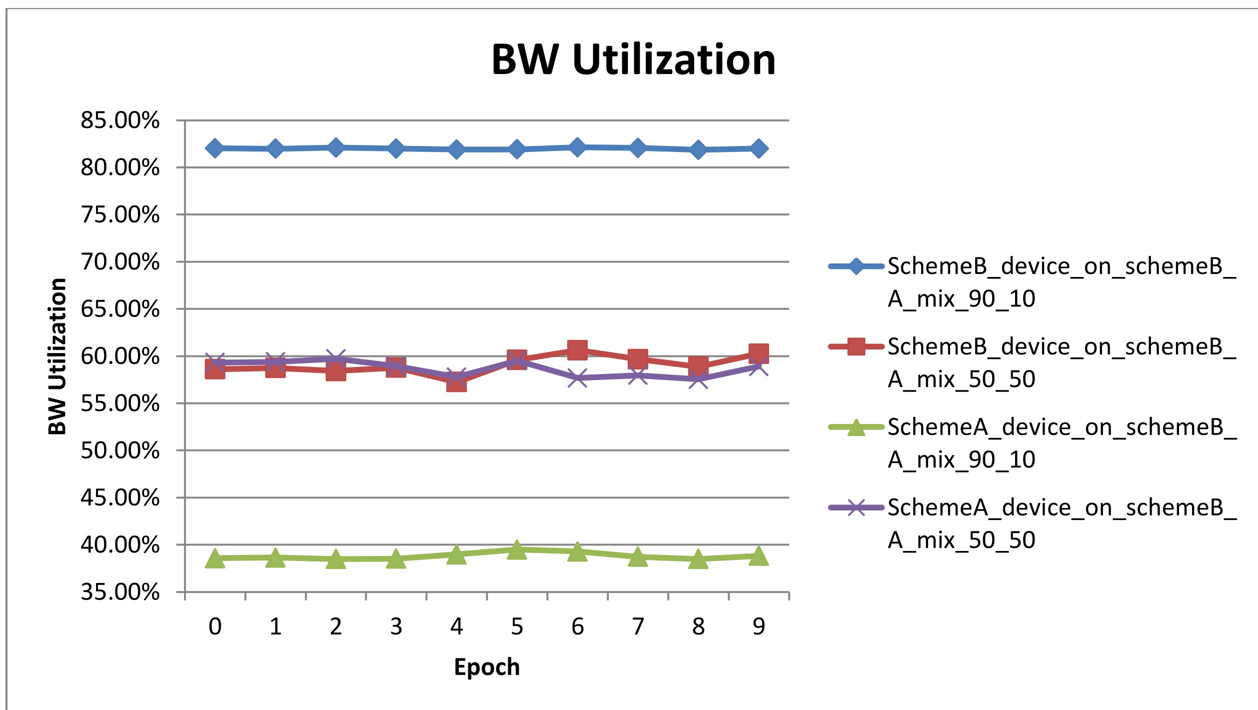

Figure 4-5: BW utilization of running mixed synthetic traces on a single device configuration

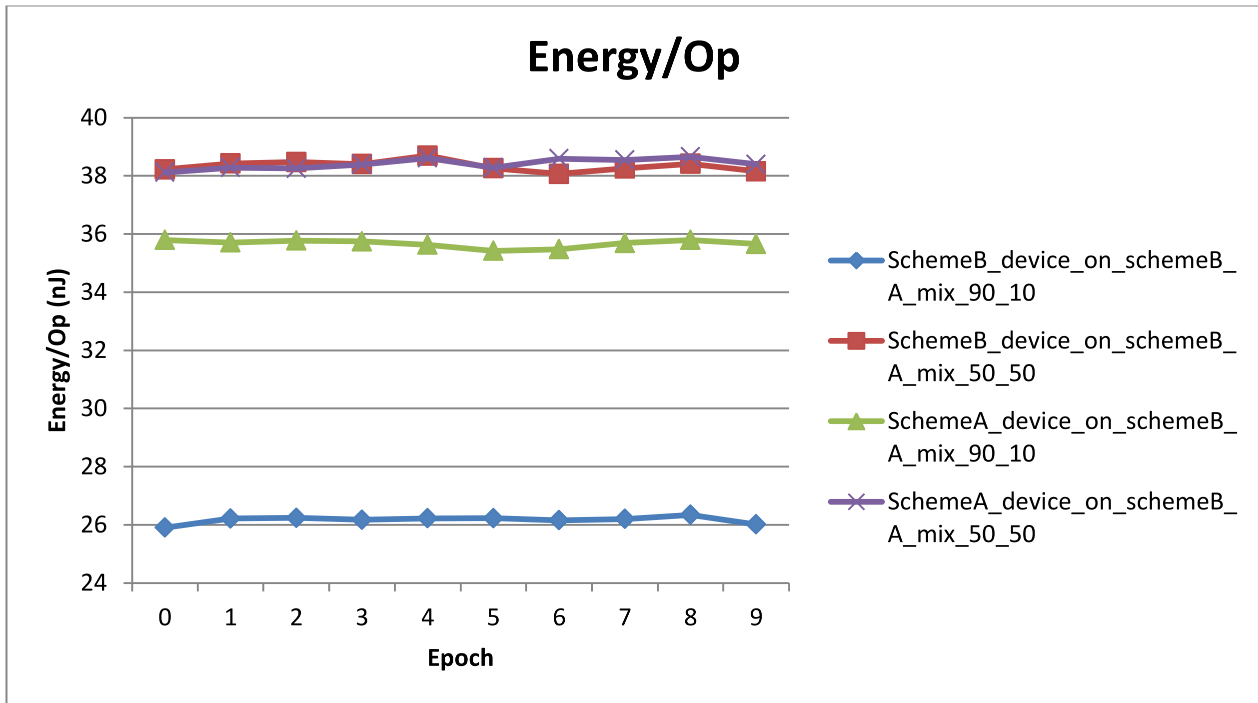

Figure 4-6: Energy/Op of running mixed synthetic traces on a single scheme device configuration

To understand the results obtained above, we took a deeper look inside the DRAM device, at the level of DRAM bank accesses. Fig. 4-6 shows the distribution of memory requests across the different banks. It clearly shows how bank0 (B0) is hot-spotted, in which several requests are satisfied from it, rather than other banks. *The result of this strided collision, presented as a loss of bank-level parallelism, results in lower performance and higher energy cost*. Using a device scheme matching the trace scheme results in more even distribution of memory requests across the different banks and hence prevents hot spotting (bank conflicts) resulting in higher bandwidth and energy efficiency. This can be noticed by the 4GB_schemeB_on_schemeB in Fig. 4-7.

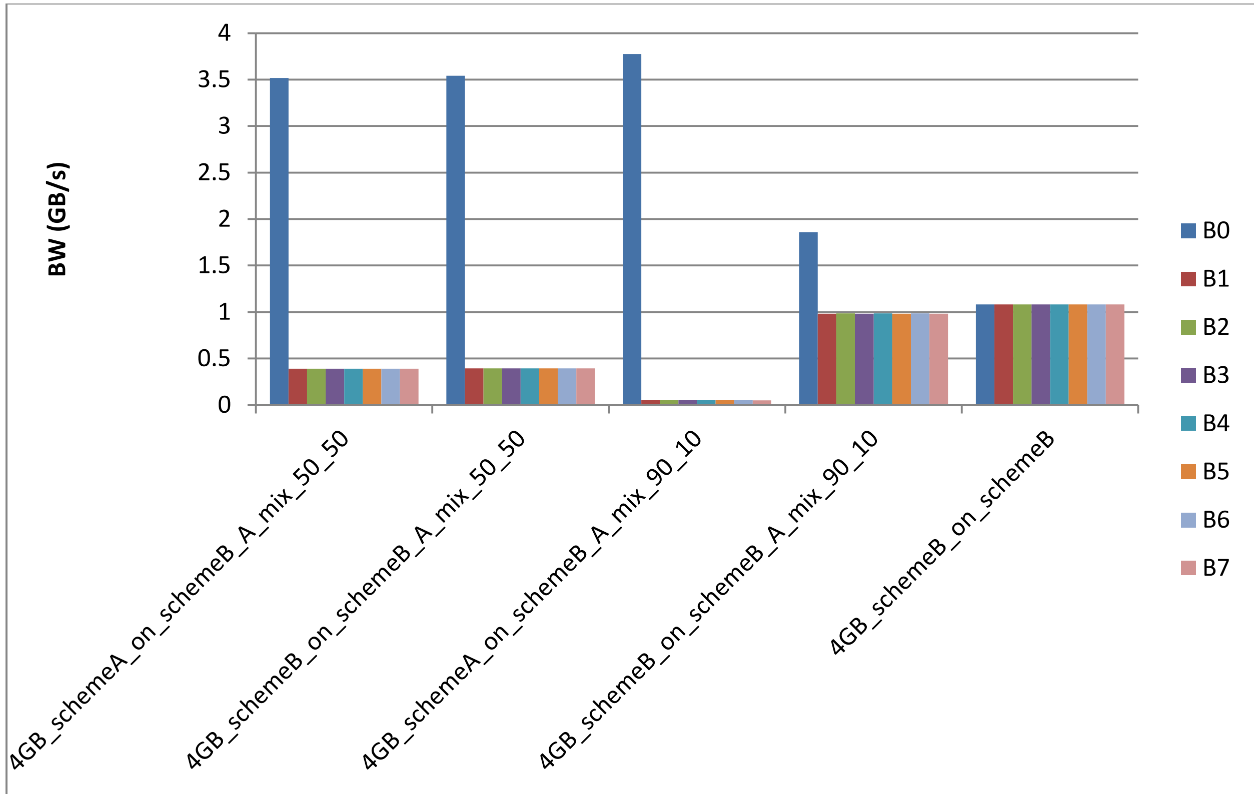

Figure 4-7: Per-Bank bandwidth from different runs on a single scheme device

4.3 Default Scheme vs. Dynamic Multiple Mapping Schemes

Results above show how the default scheme (SchemeB) can affect performance drastically (up to 2x reduced peak BW) when a non-matching scheme is used. The negative effect on performance is proportional with the ratio of memory requests received in a multi-threaded application from a thread showing schemeA tendency. Therefore, restricting the MC to use one default scheme is not an optimal solution for such workloads.

In this experiment, we run the same trace mixes in section 4.4, but implementing a per-thread dynamic address mapping scheme explained in section 3.3.4. A set of two experiments are conducted:

- 4GB_schemeB_t1_schemeA_t0_90_10: Identifying each request from the SchemeB-matching trace as thread1 (t1) and schemeA-matching trace as thread0 (t0), in a trace mix of 90:10 ratio.
- 4GB_schemeB_t1_schemeA_t0_50_50: Same as above, but with 50:50 mix ratio.

Using such mix ratios provides enough swing to understand the possible gains in performance and energy by using the dynamic address mapping schemes. Fig.4-8 and 4-9 shows the performance and energy results of the experiments compared to those in 4.2.

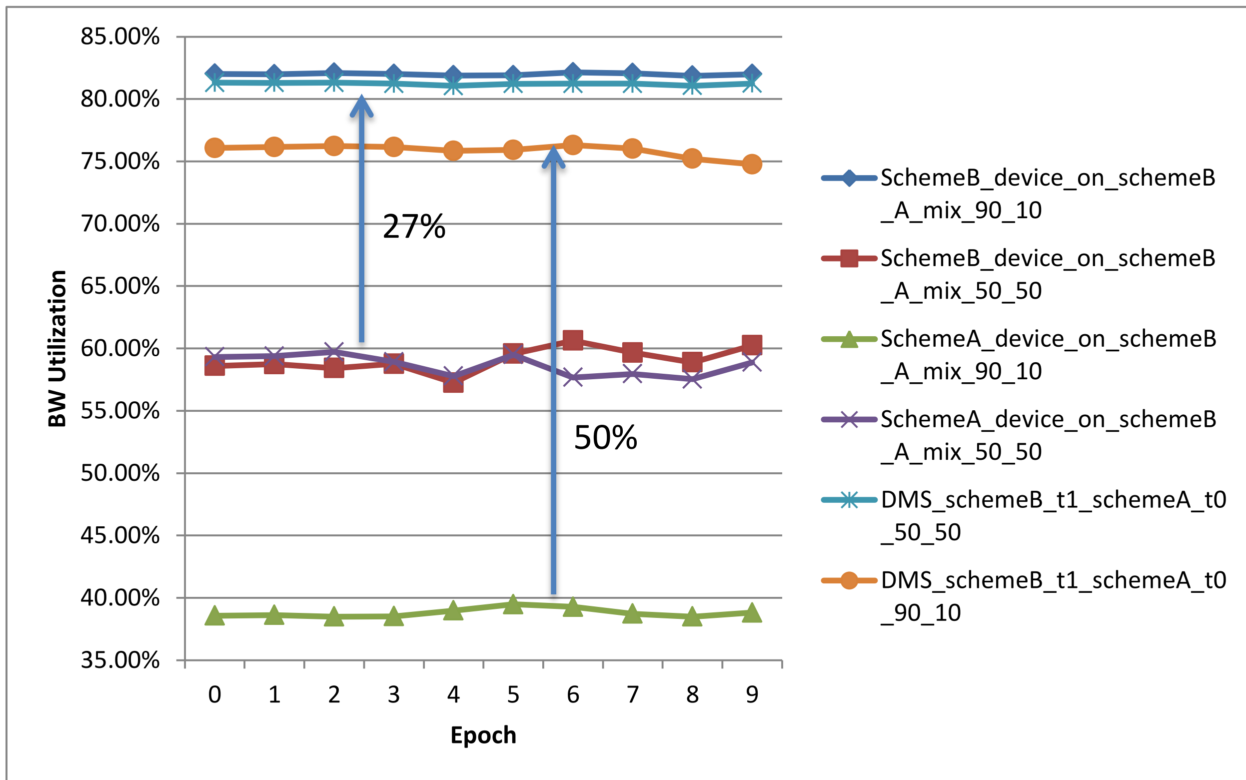

Figure 4-8: BW utilization of running mixed synthetic traces on a dynamic address mapping device configuration

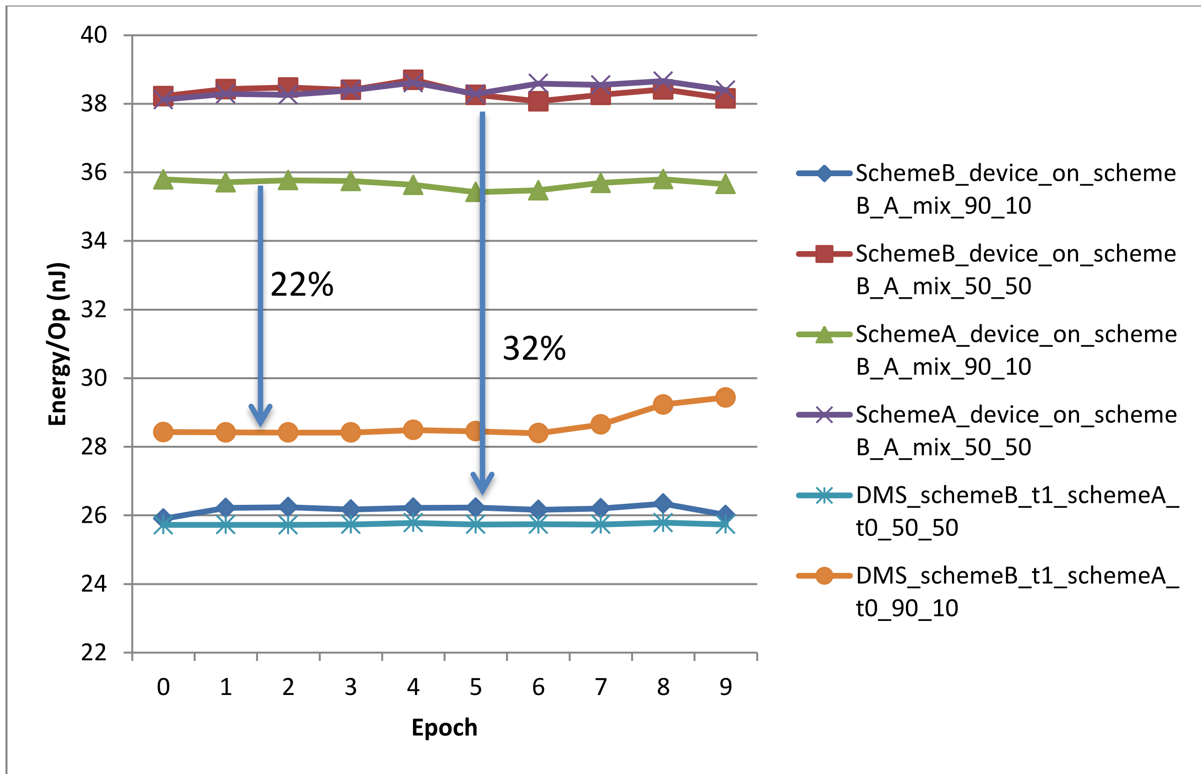

Figure 4-9: Energy/op of running mixed synthetic traces on a dynamic address mapping device configuration

Using a dynamic address mapping scheme boosted the equal workload performance by 27% with 32% higher energy efficiency compared to the single SchemeB device configuration. Up to 2x performance gain was achieved along with 22% increased energy efficiency when running the trace mix with a non-matching scheme dominant workload. This experiment shows the importance and superiority of our technique over the single fixed default mapping scheme.

Fig. 4-10 shows the per-bank bandwidth when using the dynamic mapping scheme. The 4 GB physical address space is divided equally between the two schemes by assigning four banks for each thread. The dynamic address mapping scheme maintains the thread bank-level parallelism by assigning the matching scheme to the corresponding application.

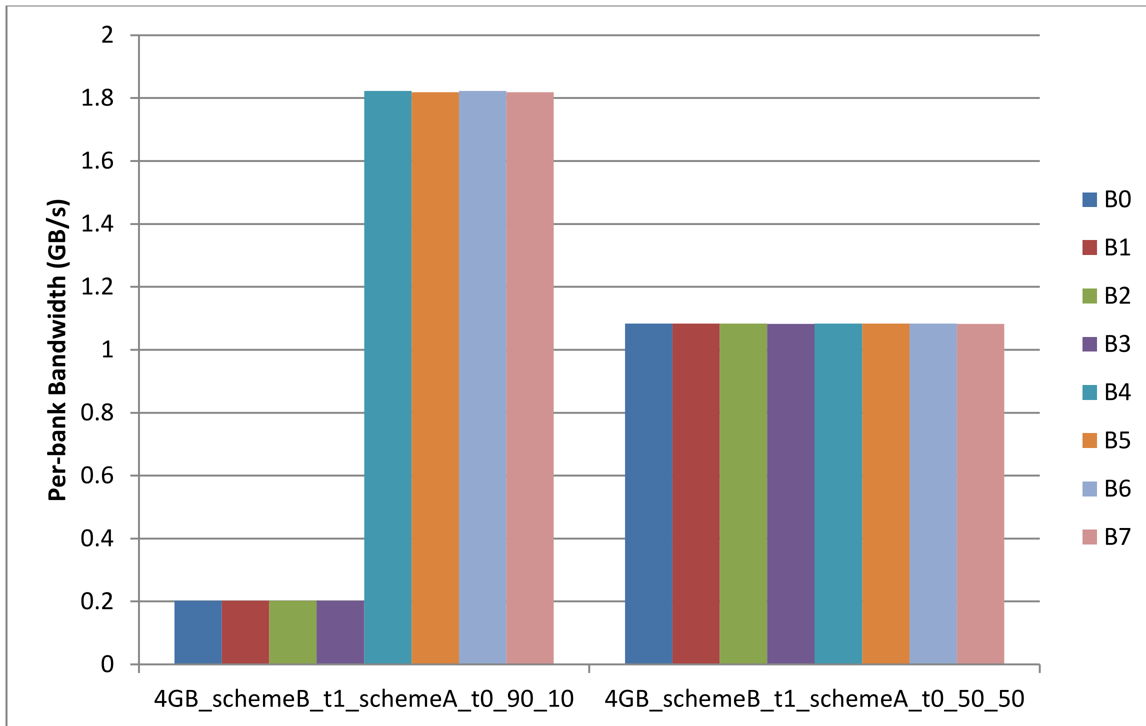

Figure 4-10: Per-Bank bandwidth from different runs on a dynamic mapping scheme device

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1 Summary

DRAM represents the backing-store of data to be processed by the CPU. It offers high capacity with moderate latency of data accesses. It is made up of several storage devices that work in unison on the data bus to supply data. The capacity of the DRAM is determined by the number of storage devices it has, which depends on the device and bus width. Storage devices are grouped into ranks, and each storage device is internally banked. Each bank consists of several memory arrays composed of rows and columns that activate a single-bit memory cell made up of a transistor-capacitor pair.

The DRAM is connected though a bus to the memory controller and communicates with the processor indirectly through it. The memory controller handles all processor requests- received in the form of cache misses- and satisfies them from the DRAM by a series of address translation and control signaling. The memory controller allocates the data in the DRAM using the physical address of the missed cache line by what is called the *address mapping scheme*. It translates the address into the channel, rank, bank, row and column to be accessed. This mapping scheme can be in different permutations of the DRAM device hierarchy (up to 120 address translation schemes are possible), and only one fixed scheme is used in the MC as it offers a simple design trade-off across many applications, while using other schemes blindly could affect performance and power consumption of other applications.

We investigate the above claim and study the impact of using a single mapping scheme on different sets of synthetic matching/non-matching traces running on various device scheme

configurations. For that to be possible, a synthetic-trace generator is developed that allows flexible generation of any single-threaded trace matching a specific device-scheme. Synthetic traces are used as a proof-of-concept that a single scheme is not sufficient to provide the highest levels of MLP, and to understand the maximum gains/losses that can be attained when running traces on non-matching device scheme. A trace mixer is then generated to allow studying multi-threaded applications by mixing different single-threaded traces at various proportions

We use DRAMSim2, a cycle accurate simulator of the DRAM unit, MC and busses, to run several experiments on four address mapping schemes. To measure performance, we use peak bandwidth utilization, while energy/op is used to measure energy efficiency. Results showed that the default scheme is not sufficient to even provide moderate performance on traces that are not aligned with it. In such cases, bank-conflicts arise resulting in a maximum loss of 2x in peak bandwidth utilization and 28% in energy efficiency.

To preserve such high peak bandwidth utilization and energy efficiency, we developed a new dynamic mapping scheme that allows multiple address translation schemes to be used, in which each thread is statically assigned to a matching scheme. In cases of fixed power budgets, as in mobile devices, a non-matching scheme can be assigned instead to limit bank-level parallelism into a single bank, allowing other banks to be placed in power-down mode, hence saving power –at the cost of performance.

5.2 **Future Work**

.Our study is the first to show that a single fixed DRAM address mapping scheme does not provide the highest levels of performance and energy efficiency on all applications. Using

synthetic traces helped us to quantify the potential gains from using dynamic address mapping, but real benchmark applications that exhibit access patterns matching the various schemes studied are the ultimate goal. For example, an initial study examining MP3 and H.264 decoder applications showed that the default scheme does indeed perform better than any other single fixed scheme (Fig. 5-1). The next task in this research is to investigate poor-cache performance benchmarks, like BLAS, sparse-matrix codes, applications with heavily strided access patterns, or possibly even others from SPEC2006. Once a set of benchmarks are found to perform poorly on the default scheme, and in which the proposed dynamic scheme offers better performance and energy efficiency, we can then stamp our technique as a valid approach to be adopted on specific problem sets.

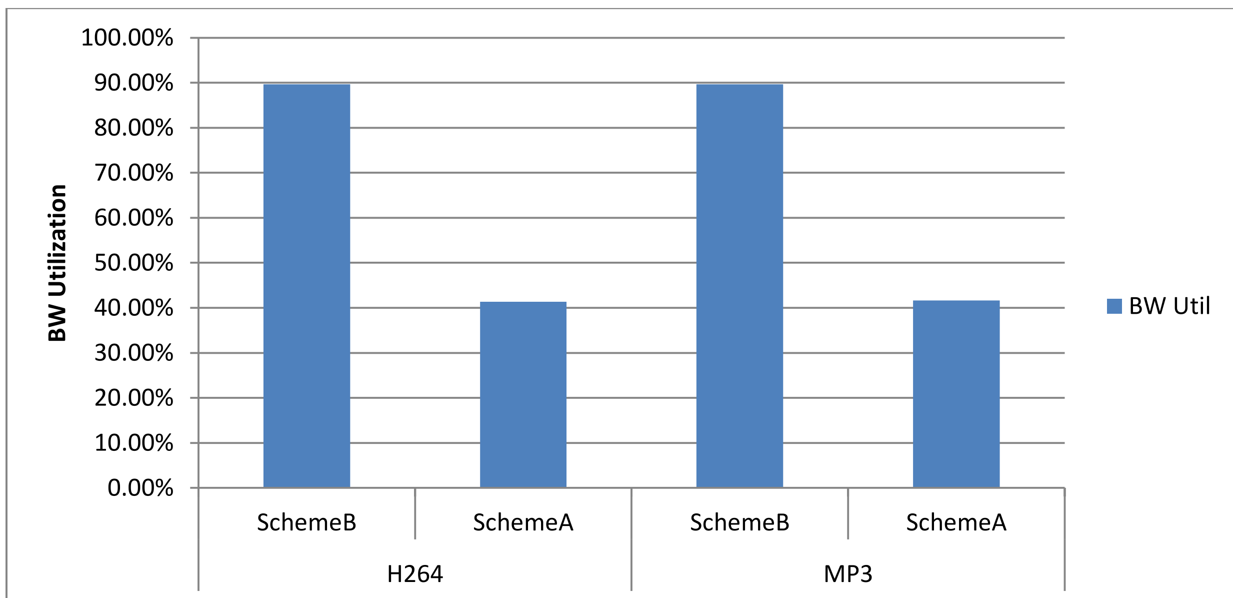

Figure 5-1: Utilization of some multimedia benchmarks

Currently, our dynamic address mapping scheme relies on application profiling prior to runtime to decide the matching scheme to be statically assigned during runtime. An adaptive

technique is favorable as application profiling data might not be readily available; hence an adaptive method will offer a more complex, yet more robust and dynamic solution. The initial idea is to implement a history-based scheduler that monitors the bank accesses –through counters- at certain epochs and decides whether to shift into another scheme or not. This process is involved, as data needs to be copied into the new locations accessible through the newly devised scheme. The copy process is pure overhead, and hence will negatively affect the optimization technique. Another method that relaxes the data copying method relies on directing new writes into the locations addressable by the new address mapping scheme. However, this could be limited as the ratio of writes could be less than reads, and hence not all reads could be satisfied from the newly writer locations, not to mention if the trace includes any writes at all –as in the multimedia decoder applications above.

Moreover, our technique is orthogonal to other optimization techniques, like the MC scheduler transaction reordering in [22], or power down methods [25]. It would be interesting if such techniques could be combined with ours to push the performance gains and energy efficiency further.

Finally, the DRAMSim2 is powerful, yet simple simulator to study memory traces. However, this obscures the direct effect of MC optimization techniques on the processor. The idea is to add this simulator to another cycle-accurate multi/single processor simulator, like SESC or SimpleScalar, and study the direct effect of such optimization technique on processor performance, such as IPC.

LIST OF REFERENCES

[1] P. Kogge, "The EXECUBE Approach to Massively Parallel Processing," in *International Conference in Parallel Processing*, Auguest 1994

[2] D. Patterson, T. Anderson, N. Cardwee, R. Fromm, K. Keeton, C. Kozyrakis, R. Tomas, and K. Yelick, “A case for Intelligent DRAM,” in *IEEE Micor,* pages 33-44, March/April 1997

[3] S. Przybylski, “Embedded DRAMs: Today and Toward System-Level Integration,” in *Tutorial with the Annual International Symposium on Computer Architecture*, May 1997.

[4] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang, “Evaluation of Existing Architectures in IRAM Systems,” in *First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember*, June 1997

[5] P. Kogge, “The EXECUBE Approach to Massively Parallel Processing,” In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994

[6] C.E. Kozyrakis et al, “Scalable Processors in the Billion-Transistor Era: IRAM.” *In IEEE Computer*, pages 75-78, September 1997

[7] S. Kaxiras, R. Sugumar, and J. Schwarzmeier, “Distributed Vector Architecture: Beyond a Single Vector IRAM.” In *First Workshop on Mixing Logic and DRAM: Chips that compute and Remember*, June 1997

[8] D. Patterson et al, “ISTORE: Intelligent Store.” *http://iram.cs.berkeley.edu/istore/index.html*, 1998

[9] J. Granacki et al, "Data Intensive Architecture: DIVA.", *http://www.isi.edu/asd/diva/*, 1998

[10] M. Oskin, F. Chong, and T. Sherwood, "Active Pages: A Computation Model for Intelligent Memory.", In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192-203, June 1998

[11] CUDA, a parallel computing platform and programming model invented by NVIDIA, *http://www.nvidia.com/object/cuda_home_new.html*

[12] Y. Kang, M. Huang, W. Huan, S.M. Yoo, Z. Ge, V. Lam, D. Keen, Z. Ce, V. Lain, P. Pattniak, and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," In *International Conference on Computer Design*, 1999

[13] R. Crisp, "Direct Rambus Technology: The New Main Memory Standard." In *IEEE Micro*, pages 18-28, November 1997

[14] J. A. Gasbarro, "The Rambus Memory System." In *International Workshop on Memory Technology, Design and Testing*, pages 94-96, 1995

[15] C. Lefurgy, K. Rajamani, F. L. Rawson III, W. Felter, M. Kistler, and T. W. Keller, "Energy Management for Commercial Servers." In *IEEE Computer*, 36(12):39–48, 2003

[16] W.A. Wulf and S.A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *Computer Architecture News*, vol. 23, no. 1, pp. 14-24, Mar. 1995

[17] W. Felter, K. Rajamani, T. Keller, and C. Rusu, “A Performance-Conserving Approach For Reducing Peak Power Consumption In Server Systems.” In *Proceedings of the 19th Annual International Conference on Supercomputing*, pages 293- 302 , 2005

[18] L. Benini, A. Macii, and M. Poncino, “Energy-Aware Design of Embedded Memories: A survey of Technologies, Architectures, and Optimization Techniques.” *ACM Transactions on Embedded Computing Systems*, 2(1):5–32, 2003

[19] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin, “DRAM Energy Management Using Software and Hardware Directed Power Mode Control.” In *Proceedings of the Seventh International Symposium on High- Performance Computer Architecture*, pages 159–170, 2001

[20] X. Fan, C. Ellis, and A. Lebeck, “Memory Controller Policies for DRAM Power Management.” In *Proceedings of the 2001 International Symposium on Low-Power Electronics and Design*, pages 129–134, 2001

[21] B. Diniz, D. Guedes, J. Wagner Meira, and R. Bianchini, “Limiting the Power Consumption of Main Memory.” In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 290–301, 2007

[22] O. Mutlu and T. Moscibroda, “Parallelism-Aware Batch Scheduling: Enhancing Both Performance and Fairness of Shared DRAM Systems.” In *ISCA-35*, 2008

[23] J. Carter, W. Hsieh, L. Stoller, M. Swansony, E. Brunvand, A. Davis, C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, T. Tateyama, and L. Zhang., “The Impulse Memory

Controller," in *IEEE Trans. Computers, special issue on advances in high-performance memory systems*, vol. 50, no. 11, pp. 1117-1132, Nov. 2001

[24] D. Kim, M. Chaudhuri, M. Heinrich, and E. Speight, "Architectural Support for Uniprocessor and Multiprocessor Active Memory Systems." In *IEEE Transactions on Computers*, pages 288-307, March 2004

[25] I. Hur, and C. Lin, "A Comprehensive Approach to DRAM Power Management." In *IEEE 14th International Symposium on High Performance Computer Architecture*, pages 305-316, February 2008

[26] B. Jacob, S. Ng, and D. Wang, *Memory Systems: Cache, DRAM, Disk*. Massachusetts: Elsevier, 2008

[27] "DRAMSim2 Memory Simulator, " *http://www.ece.umd.edu/dramsim/*

[28] "Micron, Inc." *http://www.micron.com/about/*

[29] "Micron DDR3 SDRAM," *http://www.micron.com/products/dram/ddr3_sdram.html*

[30] "PC Magazine Encyclopedia," http://www.pcmag.com/encyclopedia_term/0,2542,t=MTs&i=47408,00.asp#fbid=xbbDsDFJv36

[31] W. Lin, S. K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," in the *7th International Symposium on High-Performance Computer Architecture*, January 2001

[32] "Intel® 82955X Memory Controller Datasheet", pages 84-95, April 2005