Heterogeneous Memory System Design

by

Sujay S. Phadke

Ph.D. Thesis Proposal

Doctoral Committee:
Prof. Satish Narayanasamy
Prof. Scott Mahlke
Prof. Dennis Sylvester
Prof. Pramod Sangi Reddy

Department of Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, Michigan
2011

TABLE OF CONTENTS

CHAPTERS

CHAPTER 1

Introduction

Performance and power are two of the most important design parameters for computer systems. Often, performance can only be obtained at the expense of power and vice-versa. High performance servers play a key role in modern datacenters, serving huge amount of internet traffic per second. With the ever expanding services of companies like Google, Microsoft, Amazon, etc., understanding the performance and power characteristics of these servers is critical. The individual nodes in a datacenter may be composed of off the shelf general purpose machines or energy efficient blade servers. The performance of every node is critical to quickly access the required information or perform a computation. A large amount of power is spent in cooling the datacenters, which is necessitated by the amount of heat given off by the computing nodes. This heat is directly related to the amount of power consumed by the individual nodes. Hence, if we can optimize the power of a single node, we can directly impact the overall efficiency of a datacenter. It has been estimated that upto 25% of the operating costs for datacenters stem from cooling costs. [1].

In the consumer segment, desktop and laptop computers are increasingly being designed by pushing their boundaries of performance. While these systems were largely single core in the past, two or four core systems are now common for personal computing. The latest processor from Intel, core i7, has between 4 to 6 cores per chip [2]. AMD's

latest Phenom II processors feature similar 4 or 6 core designs [3] to do enormous number crunching and support computing, graphics rendering and connection to I/Os. For consumer desktop computers, the focus is often high performance, with less importance to power consumption. In laptop computers however, this design strategy is often reversed or balanced with a view to have high mobility, good performance and long battery life. The multi-core design paradigm was the answer to the saturating processor clock frequency problem. It enables us to achieve high performance by splitting a task into multiple tasks that can be done in parallel, with each core running at lower frequencies than the highest performing single core design.

A significant amount of effort has gone into designing low power systems while providing high performance. These efforts include techniques at various levels of abstraction like device, circuits, micro-architecture and OS/System level. Conventional techniques are focused on designing high performance, low power processors. Current advances in process and packaging technologies enable designers to integrate multiple processors on the same die. In the current generation of Intel and AMD processors, (like core i series), the memory controllers are also integrated on-chip. The power consumed by different components in the system directly affects the energy efficiency. Out of the total system power, the main memory (DRAM) consumes a significant amount of power, sometimes as high as 40% [4, 5]. Naturally, we need to develop new technologies to optimize the performance and power of the memory system, which contributes significantly to the overall system's performance and power profile.

1.1 Research Goals

1.2 Research Contributions

This thesis proposal makes following contributions:

- It introduces the idea of a heterogeneous memory system to serve the different needs of applications
- Using the proposed heterogeneous memory architecture, a static profiling scheme is described which results in an considerable performance benefits and power savings compared to existing schemes.
- A dynamic scheme exploiting the slack in memory requests in described to improve the efficiency of the memory controllers in the system.
- Finally this proposal presents the challenges in a CPU+GPU processor core and proposes the research directions for the future.

CHAPTER 2

Background

The main memory (DRAM) plays a key role in determining overall performance of the system. In this chapter, we will discuss the structure of the DRAM and DRAM operation. We will also look at the various tradeoffs in DRAM architecture.

2.1 Basics of DRAM Operation

The memory system block diagram is shown in Fig. 2.1. In modern computer systems, the DRAM consists of a number of devices bundled together. At the highest level, the memory consists of a number of modules called dual in-line memory modules or *DIMMs*. [6] Each of these DIMMs contain a number of individual chips, which store the user data. The DIMM modules are connected to the processor core through the

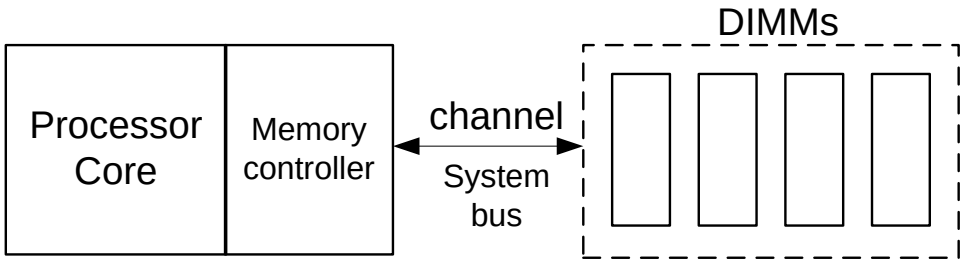

Figure 2.1: Processor Core to Memory Interface

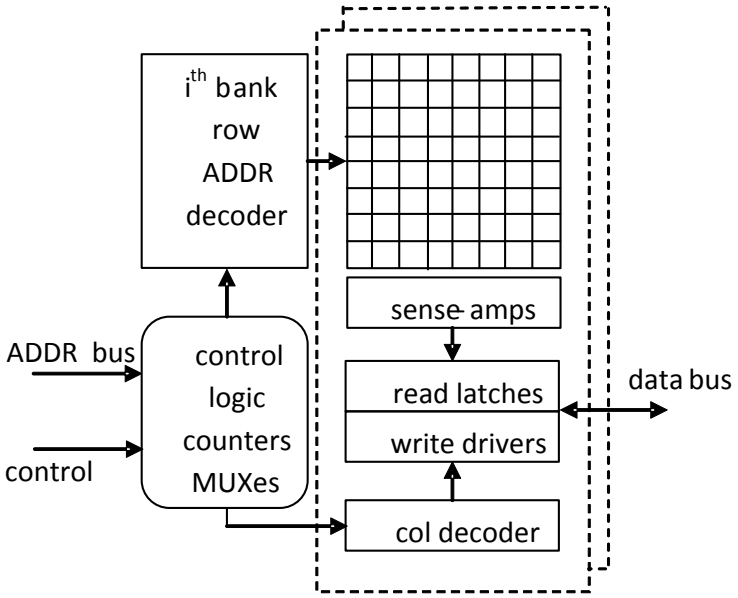

Figure 2.2: Architecture of the DRAM array within a single chip

system bus (also called a channel). Within the processor core, a module called the *memory controller* handles the transfer of data between the bus and the processing cores. Modern processors have multiple channels connecting the DIMMs to the on-chip memory controllers.

When the memory requests from the core(s) miss the **L2** (or the last-level) cache, the request is sent to the memory controller (MC). The MC is responsible for queuing the requests, priority and arbitration, communicating with the DRAM and getting the required data. It generates the required control signals necessary for the DRAM operation and reads/writes data over the system bus.

The bus is divided into an address bus, a data bus and a control bus. The bus is shared between all chips on a DIMM. The *rank* of the DIMM specifies the number of chips (or groups of chips) that can be accessed independently by the MC. DIMMs typically have a single or dual rank. The DRAM hierarchy requires that the incoming address be translated by selecting the channel, DIMM, rank, chip and then the cell within the chip.

Each DRAM chip consists of an array architecture as shown in Fig. 2.2. The chip is logically divided into a number of *banks*. Within each bank, data is arranged in an array like fashion, with rows and columns.

A row is also referred to as a *page*. The incoming address contains the relevant row and column from which data is to be accessed. The control signals generated by the MC interface with the control logic inside the chip and select the appropriate chunk of data. This data is internally stored within *prefetch buffers* to hide the latency of access of multiple data words (burst access). When a memory request is received, a particular row (or page) is first activated (or opened). This is specified by the row access strobe (RAS) signal and reads the entire row into the row-buffer (latch). In the next cycle, the column address is specified through the column access strobe (CAS) signal and this selects a particular column from the open row. Once the data is available, it is transferred over the bus to the memory controller. Since a DRAM read operation is destructive, the data read must be written back after an access is complete. Each DRAM cell consists of a transistor and capacitor pair (1T) design. The dynamic nature of this type of memory means that the charge (data) stored on the capacitor slowly leaks out, which must be periodically replenished by refreshing the cell. This operation is called *precharge* is handled internally within the chip.

2.1.1 Memory Latency

DRAM latency is the time required to fetch data from the memory array and send it to the requesting CPU core. It is typically measured in system clock cycles.

Low memory latency is important for transfer for small amounts of critical data and rapid synchronization. Low memory latency is essential for applications like video editing or voice and video based networking applications. Various circuit level techniques have been used to reduce DRAM latency. Reduced Latency DRAM (RLDRAM) [7] addresses the pre-charge and wait problem by breaking DRAM into

more banks, which gives a higher probability that the bank being accessed will be different from the one just accessed. RLDRAM also offers single cycle (SRAM) addressing. Instead of requiring 1 cycle for row activation and 1 cycle for column addressing, the row and column are addressed in the same cycle. RLDRAM II technology is built around minimizing access latency and reduced row cycle times ($tRC = tRAS + tRP$) are are ideally suited for latency sensitive network applications. It is also an ideal candidate for replacement to SRAM in L3 cache architectures because of its relatively low cost.

Fast cycle RAM (FCRAM) from Fujitsu offers a decreased latency due to integrated row address strobe and column address strobe operations in the command set. The pipelined operation for access commands and a hidden pre-charge reduce the random access cycle time. The memory array is divided into smaller arrays for better parallel data access. FCRAM is ideal for use as a buffer memory in high speed network applications. Enhanced SDRAM (ESDRAM) uses a small amount of on chip cache with regular SDRAM to speed up access to commonly used data. This allows for lower access latency. A secondary effect of this is that a larger bus can be used between the on-chip SRAM cache and DRAM, which provides a higher effective bandwidth for cache misses. ESDRAM also provides SRAM row buffers to eliminate the row precharge time (tRP) for back to back row accesses.

2.1.2 Memory Bandwidth

Bandwidth refers to the rate at which data can be transferred to or from memory, which can be averaged over a large sample of processor cycles. Applications demands can range from being extremely bandwidth sensitive to completely tolerant of low bandwidth. DRAM is organized into a number of modules, which have multiple ranks in them. Each rank is composed of multiple memory chips which are divided internally into a number of banks. This hierarchical system provides the much need

concurrency for applications. With bank-level parallelism, it is possible to issue simultaneous read/write commands to different banks in a pipelined fashion and hide the corresponding long latency.

The available bandwidth is bounded by several constraints and choices: power budget, total memory size, I/O bus width, total number of pins, cost. DDR family of memories are designed to strike a balance in these parameters for the various types of applications than can be executed in the system (the common case scenario). The worst-case usage is back-to-back reads/writes from/to the same bank but different rows. The row must be precharged at the end of each access, and no accesses are being made to other banks. Various strategies can be employed at the MC to optimize row usage.

Hence, in order to increase the per pin bandwidth, we can adjust the burst length(BL) size or the frequency of the bus. tCK and tRC are coupled tightly with the circuit design. The overall bandwidth (at the DIMM level) is further determined by:

$$BW = (\#channels) \cdot (bus\ width) \cdot (\#bits/cycle) \cdot (f)$$

The number of channels determines the parallelism for data access. A higher number of channels increases the bandwidth at the expense of latency and cost. Multiple channels need to be supported by a number of internal banks in the DRAM for parallel access. This interleaving of pages across different banks hides concurrent access latency. A wider system bus increases the number of bits that can be transferred per cycle and the bus frequency increases the rate at which these bits are sent. In the DDR family, 2 bits of data can be transferred in every cycle per line (dual data rate). The frequency (f) of operation of the bus is determined by circuit level factors. This equation refers to the theoretical peak bandwidth.

2.1.3 Memory Power Consumption

Calculation of power in DDR systems is non-trivial since its highly dependent on operating conditions, amount of data being written/read, power saving techniques employed, etc. The total power can be split into its various components. [8]

1. Background Power: If any bank is open, a small amount of background power is dissipated in keeping it open.

$$P_{pdn} = I_{dd2p}.V_{dd}$$
$$P_{stby} = I_{dd2n}.V_{dd}$$

2. Activate Power: In order to READ or WRITE data, a bank and row must first be selected using an ACT command. For every ACT command, there is a corresponding PRE command. The ACT command opens a row, and the PRE closes the row.

$$P_{ACT} = \left[I_{dd0} - \frac{I_{dd3n} t_{RAS} + I_{dd2n} \left(t_{RC} - t_{RAS} \right)}{t_{RC}} \right]$$

3. Read/Write Power: The power required to read/write data is computed from the amount of current required to read/write into the DRAM array.

$$P_W = \left(I_{dd4W} - I_{dd3N} \right) V_{dd}$$

$$\mathrm{P}_r = \left(I_{dd4R} - I_{dd3N} \right) V_{dd}$$

4. Refresh Power: Data in DRAMs need to be continuously refreshed to retain data integrity. DDR3 memory cells store data information in small capacitors

that lose their charge over time and must be recharged.

$$P_{ref} = (I_{dd5} - I_{dd3N})\, V_{dd}$$

Out of the total system power, the main memory consumes a significant amount of power, sometimes as high as 40% [4, 5].

Modern DRAMs have multiple low power states and employ on-chip power saving techniques. Memory requests are first processed by the memory controller which can be governed by different policies (peak performance, low power, etc.) for achieving different goals. The memory controller buffers requests and sends the appropriate control signals to the DRAM based on its scheduling policy. Based on workload characteristics and mapping of pages in memory, memory requests and physical addresses could be clustered together or distributed sparsely. Memory power optimization comes at the cost of latency though. For example, a finer split in number of ranks or a finer grained transitions (from active to low power state) for different banks will result in an increase in the latency due to higher wake-up times. The low power DRAM (LPDRAM) proposed can be designed by the technique described in Minirank [9]. For our study, we use the Micron DRAM power calculator [10] to estimate the power of different memory modules based on aggregate statistics obtained from simulation.

2.1.4 DRAM Timings Description

Within the memory array, the access delay depends upon the following dominant factors: [6]

- **CAS Latency (tCL):** Number of clocks that elapses between the memory controller telling the memory module to access a particular column in the current row, and the data from that column being read from the module's output pins.

- **RAS to CAS Delay (tRCD):** Controls the number of clocks inserted between a row activate command and a read or write command to that row. Some chipsets (like Intel 965 and P35) allow us to change `RAS` to `CAS` read delay and `RAS` to `CAS` write delay separately.

- **RAS Precharge (tRP):** Controls the number of clocks that are inserted between a row precharge command and an activate command to the same rank.

- **Activate to Precharge delay (tRAS):** Number of clocks taken between a bank active command and issuing the precharge command. Usually, $tRAS = tCL + tRCD + 2$.

Two more quantities are commonly used by manufacturers to characterize the delay [11]. The row-access related latencies can be combined to denote the row cycle time (tRC). tRC is the minimum amount of time required to transfer data from the internal array into the buffers, write data back into the cells and complete the bitline pre-charge (ie. a full cycle). It can be thought of as a random access time to access different rows within the same DRAM bank.

Row Cycle Time (tRC): $tRC = tRAS + tRP$

The worst case latency in DRAM is specified by manufacturers as the row access time (tRAC). The value mentioned in datasheets is the maximum amount of time required, after `RAS` goes low, to select a row and a column in the array, sense the bit values and load them onto the I/O buffers.

From an architecture standpoint, these delays can be mapped onto the following latencies:

- Address decoding latency: Time to drive signals `RAS` and `CAS`. Any pessimistic design adds directly to the delay. Some designs do not multiplex the addresses in order to eliminate this constraint, requiring extra package pins, and hence increased area.

- Word line activation latency: RC (resistive-capacitive) delay from the number of gates connected to the word lines. The resulting tRC easily accounts for the bulk of the total access latency. One approach to this problem is to divide the word line into smaller sections and add buffers at the cost of area.

- Bit line sensing latency: Time required to detect if stored bit is a 0 or 1. This delay can be reduced by using advanced high speed circuit topologies, high voltage, separate read and write sense amplifiers (hides refresh time), cell-to-bit-line capacitance ratio. To reduce the delay, this ratio should be as high as possible. In practice, this means increasing cell capacitance or putting fewer cells on a bit line, both of which come with their own penalties of increasing area.

- Output driving latency: Caused by the RC of driving a long wire across the die. Again, circuit techniques may be used to reduce delay, as well as careful layout to reduce the length of the wire. Also, new packaging methods such as lead-on-chip can be used to reduce wire length.

2.2 Design Tradeoffs in DRAM Architecture

DIMMs, Channels and Memory Controller: The main memory consists of a number of modules called *DIMMs* (dual in-line memory module) [6]. The DIMM modules are connected to the system bus (or a channel) which in turn connects them to a memory controller. A memory controller queues up memory requests from processor cores, arbitrates, sends control signals to DRAM, gets the data and sends it to the requested processor core. Modern processors have multiple integrated memory controllers servicing multiple channels.

Ranks and Chips: A DIMM contains multiple ranks. A rank is composed of multiple chips. Each ×N chip has pins to support read/write of `N` bits in a cycle. For

a 64-bit data bus and ×8 chip, each rank needs 8 DRAM chips. A cache line is usually striped across chips so that its parts can be accessed in parallel.

Banks: Each chip is divided into *banks*. Contents of a single logical bank span across multiple chips and they can be accessed in parallel. Higher number of banks can service more memory requests in parallel, thereby providing higher memory bandwidth. However, higher number of banks also require more complex control logic and area [6] leading to higher bank access latency, and therefore could adversely impact the performance of a latency sensitive application that generates fewer parallel memory requests.

Arrays and Row Buffers: Within each bank, data is arranged in an array like fashion, with rows and columns. The DRAM row buffer and latches the data in the entire row when any part of it is accessed. A DRAM row is typically much larger than a single memory request. If a memory read accesses an already open row, it would experience smaller memory array read latency. Row buffer hit rates are higher for application with a higher spatial locality. In general, bandwidth bounded applications with regular memory access patterns tend to have a higher spatial locality, and therefore benefit from larger rows. Latency bounded applications with less regular memory access behavior, however, tend to exhibit poor spatial locality and therefore do not benefit from a large row buffer. In fact, for a latency sensitive application, a smaller row buffer would be beneficial as it would take lesser time to drive a smaller row. For a given DRAM array size, smaller rows may require longer column bit lines, but in most memory devices, benefits of reducing the row access time (RAS) generally outweighs relatively small increase in column access time (CAS). Thus, for bandwidth sensitive applications a larger row is beneficial, whereas for latency sensitive applications a smaller row is beneficial.

Prefetch buffer: The prefetch buffer in DDR SDRAMs is part of the chip architecture, and acts as the interface between the array and the DRAM interface.

It can fetch multiple data words in a burst without the need for additional column address signals. A prefetch buffer with a depth x allows the interface I/O clock to operate x times faster than the memory array clock. DDR3 can support a buffer of maximum depth eight. But it can also support a smaller buffer depth that reduces latency but sacrifices memory bandwidth.

Page Size: The page size in a memory system is closely related to the row buffer size. In many modern DRAM system, the page size is equal to the row buffer size. Some of the reasons for large page size are: less memory required for the OS page table, efficient transfer to or from secondary storage and fast access times within the page (burst accesses). The reasons for a smaller page size are: less wasted space and less overhead during OS context switching.

Address Mapping and Interleaving: In all DRAM architectures, the best performance is obtained by maximizing the number of row-buffer hits while minimizing the number of bank conflicts. The addressing scheme dictates this to some extent. The addressing scheme refers to the physical mapping or interleaving of pages in memory. A page-level interleaving scheme aims to maximize the spatial locality in applications, by mapping consecutive pages into the same dram row. A cache-line interleaving aims to split up a page into individual cache-lines and distribute them across multiple banks to exploit bank-level parallelism. However, this comes at the expense of increased power, since multiple banks need to be kept active at the same time to service requests to multiple cache lines.

There are three main attributes that determine the efficiency an off-chip memory: latency (L), bandwidth (B), and power (P). Memory chip designers often need to make trade-offs between these parameters. For instance, Reduced Latency DRAM (RLDRAM) [7] can operate at a latency of about `25ns` but offers only about 4 GB/s bandwidth. While DDR3–2133 chip [12] can offer nearly 18 GB/s, it operates at a much higher latency of about `45ns`. Any DRAM chip can be operated at a lower

voltage to reduce power, but that comes at the cost of increased latency and reduced bandwidth.

Currently, memory system designers try to balance across the three parameters for the main memory design. However, a general purpose multi-core system would be simultaneously running a diverse set of applications with different memory requirements. Some applications are bandwidth-bounded (e.g. graphics) and some others are latency-bounded (e.g. pointer-intensive applications). There are applications for which main memory system's performance does not matter (e.g. computationally intensive applications with high data locality). For such applications a power-efficient main memory would be a better option. The diversity of applications in a system is only likely to increase as the number of cores on a processor continues to scale.

In this thesis, we present for the first time a heterogeneous main memory for future multi-core systems. It consists of three memory modules. Each module is designed such that it provides the best efficiency for one of three parameters at the cost of sacrificing the other two.

2.3 Memory Controller

The memory controller (MC) is responsible for the data being read from and written to the DRAM. The MC queues up memory requests (reads and writes) from the processor. In order to read or write from the DRAM, the MC sends appropriate control signals. Let us consider a memory read request. The address of the request is put onto the memory bus and sent to the DRAM. The address consists of all information required for the DRAM to locate the required data. Once the request is serviced by the DRAM, the data is sent back along the memory bus to the memory controller's internal buffers. It then forwards the data upwards in the memory hierarchy (say to the L2 cache).

The DRAM is a shared system resource. Multiple threads which run concurrently on the processors share the DRAM and send requests to the memory controller. Often, the memory controller gets multiple requests from different threads, to access data in the same memory module. In order to serve these requests, system designers put in place various arbitration policies. There are various policies, which aim to either maximize throughput, or reduce power, or improve fairness amongst the different threads. Due to the shared nature of DRAM, the requests may access different regions within the DRAM. However, the DRAM can only keep active a certain section at a given time. For example, if a particular row is activated within a bank inside the DRAM, it can only service requests hitting that bank and row. If a request to another row is made, the currently active row needs to be closed and the new row activated. This is a row-buffer conflict. In multi-core and multi-threaded systems, there are many occurrences of row-buffer or bank conflicts. Higher performance demands the use of multiple cores which results in increased contention for shared resources. As a result, the memory requests from different threads may cause increasing number of conflicts, referred to as inter-thread interference.

Memory controllers can be designed to meet different design points. However, most of the current memory controllers optimize the sustained throughput. The most commonly used scheduling scheme for prioritizing multiple memory requests is the row-hit-first-first-come-first-serve (FR–FCFS). This policy states that within a group of requests, the memory controller will schedule those requests which hit the currently open row first. If there are multiple such hits, it will follow a FCFS scheme. Essentially, this is useful for threads which send multiple parallel requests hitting the same DRAM row. In this case, they would see a high throughput. However, another thread whose requests may have queued up at the memory controller much earlier will be stalled till all the current row-hit requests are served. Thus, the other thread suffers in performance and may also experience starvation. [13]. In order for

the memory controller to have a balanced approach and ensure fairness in the system, it needs knowledge of memory access patterns across different threads. It was shown by Mutlu, et al. that the inter-thread interference can cause a thread to suffer a denial of service [13]

In single threaded systems, a thread has access to the entire DRAM. In this case, single-thread performance can be greatly increased by exploiting bank level parallelism. In order to do this, the thread can issue multiple requests spread out across different banks. If the system is capable of servicing multiple outstanding requests in parallel, then the latency of multiple memory requests is hidden behind the first one. The ability to service multiple outstanding memory requests in parallel is called Memory Level Parallelism (MLP) [14]. However, in multi-threaded scenarios, a thread's MLP can easily get destroyed because of the inter-thread interference. A parallelism aware scheme can help in solving this problem to an extent. [13]

CHAPTER 3

Motivation: Heterogeneity in Applications

Applications vary widely in terms of their main memory demands. For a processor with one core (configuration shown in Table 5.1), we study three benchmarks (from the SPEC 2006 benchmark suite in Table 3.1) with diverse memory demands. Fig. 3.1 and Fig. 3.2 present a case study to illustrate this observation. `astar` is sensitive to latency, `milc` is sensitive to bandwidth, and `perlbench` has high locality and only rarely accesses memory. We consider three different types of memory modules optimized for either latency (M_L), bandwidth (M_B) or power (M_P), at the cost of sacrificing some efficiency of the other two parameters. The detailed configurations for these three devices are discussed in Chapter 4 (Table 4.1). We evaluate their performance and power with respect to a commonly used DDR3-1600 memory, which we consider as our baseline.

For our simulation, we fast-forward to an appropriate simpoint, warmed up the caches for 1 million instructions and performed detailed out-of-order simulation of 100 million instructions. The memory timing and power models were integrated into the simulation framework (with data derived from our design on memory modules later on in this work). The performance results are all normalized to a system with DDR3 modules.

As shown in Figure 3.1, a latency optimized memory improves `astar`'s performance

Benchmark	Short Description
astar	Path-finding library for 2D maps that includes the A* algorithm
bwaves	Computes 3D transonic transient laminar viscous flow
bzip2	Based on bzip2 version 1.0.3, modified to do most work in memory rather than I/O
dealII	A C++ program library targeted at adaptive finite elements and error estimation
gamess	Program that implements a wide range of quantum chemical computations
GemsFDTD	Solves Maxwell equations in 3D using the finite-difference time-domain method
gobmk	Artificial intelligence simulator that plays the game of Go
gromacs	A molecular dynamics program; simulates the protein lysozyme in a solution
h264ref	Reference implementation of H.264/AVC, encodes a video stream
hmmer	Protein sequence analysis that uses profile hidden Markov models
lbm	Implements the Lattice-Boltzmann method to simulate incompressible fluids in 3D
leslie3d	Computational fluid dynamics, uses Large-Eddy simulations
libquantum	Quantum computer that runs Shor's polynomial-time factorization algorithm
mcf	Vehicle scheduler, which uses a network simplex algorithm
milc	Gauge field generating program for lattice gauge theory programs
omnetpp	OMNet++ discrete event simulator that models a large ethernet campus network
perlbench	Minimized version of perl 5.8.7; uses the core interpreter and third party modules
povray	An image rendering program; test case is a 1280x1024 anti-aliased image
sjeng	Artificial intelligence simulator that plays the game of chess
soplex	Solves a linear program using a simplex algorithm and sparse linear algebra
sphinx3	A well known speech recognition system from Carnegie Mellon University
tonto	An open source quantum chemistry package; based on Fortran 95
xalancbmk	Modified version of Xalan-C++ that transforms XML documents
zeusmp	A computational fluid dynamics code; simulates astrophysical phenomena

Table 3.1: Description of SPEC CPU2006 Benchmarks used for simulations

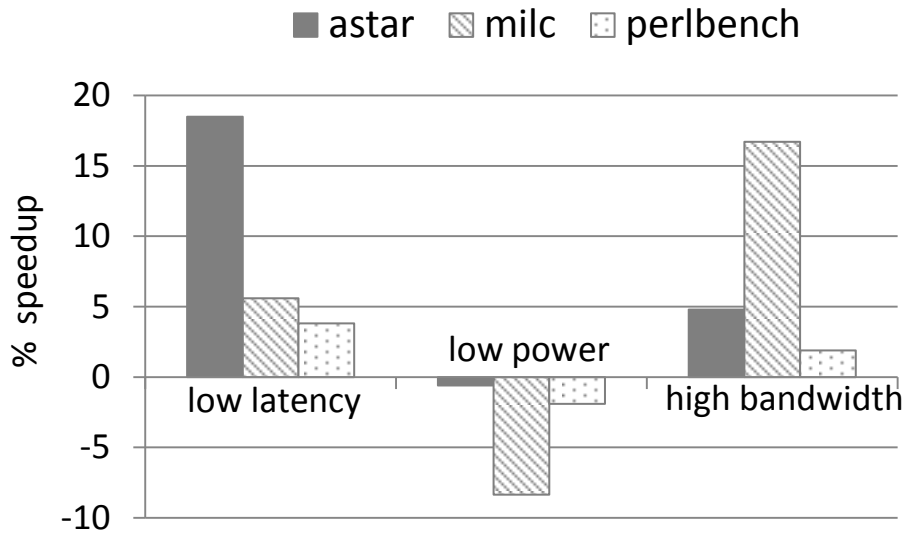

Figure 3.1: Performance of memory devices optimized either for latency, bandwidth or power, when compared to DDR3.

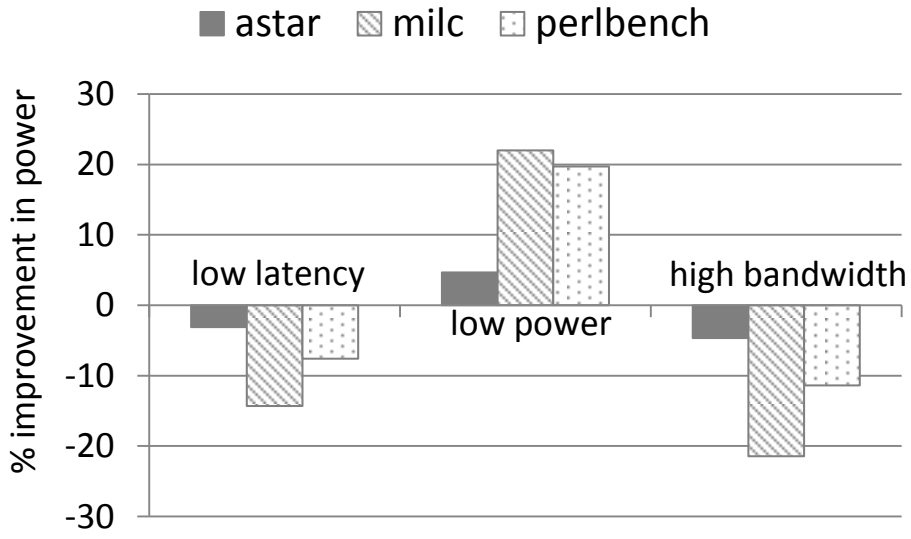

Figure 3.2: Power of memory devices optimized either for latency, bandwidth or power, when compared to DDR3.

by 18.5% for only about 1% power cost. But it is ill-suited for the other two applications. For `milc`, it improves performance only by about 5% for a significant power cost of 14%. The reason is that `milc` is bandwidth bounded and less sensitive to memory latency. Low power memory significantly reduces power by 20% for `perlbench` while incurring only 1% performance cost, because `perlbench` rarely accesses memory. However, low power memory is not suitable for `milc` as it suffers 8% performance penalty. Bandwidth bounded `milc`'s performance could be improved by nearly 17% using bandwidth optimized device. Though it comes at a power cost, the net energy-delay product is still better than baseline. But performance of the other two applications does not improve significantly while using bandwidth optimized memory.

Thus, if a multi-core system executes a mixture of the above three types of programs, then it would be beneficial to employ a memory device that contains three different memory modules where each one is heavily optimized for one of the three parameters.

CHAPTER 4

Design of a Heterogeneous Memory System

In this section, we present the design of a heterogeneous memory system to achieve performance and power improvements over the baseline homogeneous memory system.

4.1 Memory Design Choices

The spectrum of design characteristics (latency, bandwidth and power) for several known implementations is shown in Fig. 4.1 and their detailed characteristics are presented in Table 4.1. Sources for this data for some designs are provided in the table. The rest were obtained from the Micron data sheets [12]. Since different manufacturers quote different timings for the same DRAM (due to manufacturing methods, etc.), we provide an average value for the observed access latency. Power depends on activity which we derived using the Micron's power calculator [10]. Characteristics of three of our designs (M_L, M_P, M_B) were estimated using our simulation models which are discussed in Section 5.3.1.

As shown in Fig. 4.1 memory system architects balance the trade-off between three parameters. For example, DDR3–1600 has a high bandwidth, but comes at the cost of high latency and power. RLDRAM optimizes latency and is used in high speed network packet processing. However, the peak bandwidth of RLDRAM is much

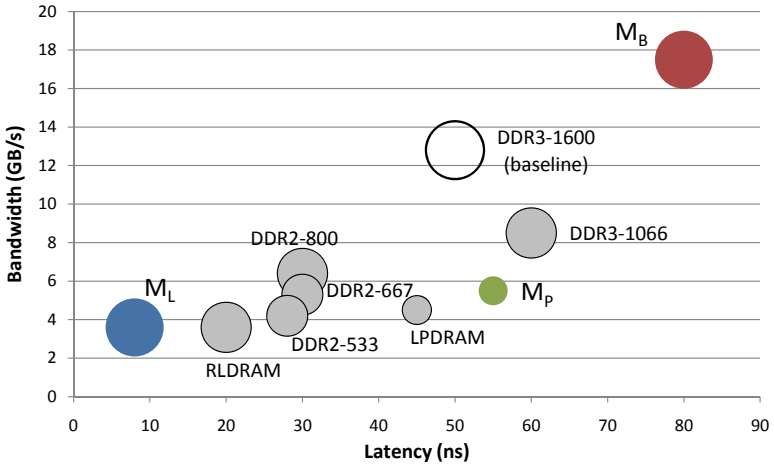

Figure 4.1: Memory design space showing latency and bandwidth for different memory types. The size of the bubble corresponds to the average power.

	# banks per module	Rowsize (bytes/ row)	Addr. mode	Pre-fetch buffer	Voltage (V)	Latency (ns)	B/W (GB/s)	Power (est.)
M_L	4	32	SRAM	2n	1.8	8	3.6	V. High
M_B	16	512	DRAM	8n	1.65	80	17.5	V. High
M_P	2	64	DRAM	1n	1.5-1.3	55	5.5	Low
DDR3-1600	8	256	DRAM	8n	1.5-1.3	50	12.8	High
DDR3-1066	8	256	DRAM	8n	1.5	60	8.5	High
DDR2-800	4	128	DRAM	4n	1.8	30	6.4	Med
DDR2-667	4	128	DRAM	4n	1.8	30	5.3	Med
DDR2-533	4	128	DRAM	4n	1.8	28	4.2	Med
ESDRAM 166 [15]	4	512	DRAM	2n	3.3	11	1.6	Med
LPDRAM	4	256	DRAM	2n	1.5	45	4.5	Low
RLDRAM 800	8	64	SRAM		1.5	20	3.6	High
FCRAM 200	4	128	DRAM		1.8	25	3.46	Low
PCM [16]	4	256	-		1.2	R: 50-100; W: 1000	0.1	V. Low

Table 4.1: Heterogeneous Memory Modules Design Space

lower than the DDR3 family. LPDRAM [17] is optimized for power to support mobile applications at the cost of higher latency (compared to DDR2) and lower bandwidth (compared to DDR3). In short, there is no single homogeneous memory which can provide best possible latency, power and bandwidth.

4.2 Heterogeneous Architecture Overview

As described in the previous section, there are many different design points which we can use to build a heterogeneous memory. In this section, we discuss one simple heterogeneous main memory design. Our design is composed of three memory modules where each of them is either optimized for latency, bandwidth or power. We start with the DDR3-1600 design and optimize its structures for any one of the three parameters. For simplicity, our architecture assumes a single channel connecting three memory modules to an on-chip memory controller. Also, it uses the industry standard DDR3 interface as the interface of choice for all three types of memories. The interface can support varied latency modules and offers high bandwidth [18].

Fig. 4.2 illustrates the difference between a homogeneous and a heterogeneous memory architecture. In our heterogeneous memory, based on the demand of an application, its pages would be stored in the appropriate memory module (Fig. 4.2). For bandwidth sensitive applications, in addition to improved device efficiency, it is also important to exploit bank-level parallelism to issue simultaneous read/write commands to different banks in a pipelined fashion and achieve higher bandwidth. Therefore, we allocate pages of a bandwidth sensitive application across all the banks in all the memory modules. A profiling algorithm to classify applications and a heterogeneous memory aware page allocation policy are described in Section 5.2. The rest of this section describes the three different memory modules used in our heterogeneous memory.

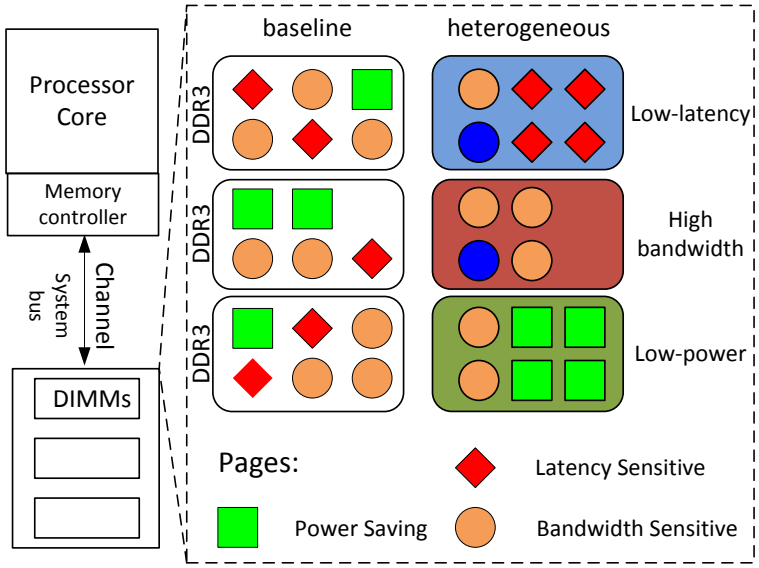

Figure 4.2: Page allocation in baseline homogeneous memory versus heterogeneous memory

4.2.1 Latency Optimized Memory Module (M_L)

DRAM latency is the time required to fetch data from the memory array and send it to the requesting CPU core. As described in Section 2.2, smaller row buffers could reduce the worst case memory latency. While smaller rows could degrade the performance of bandwidth bounded applications with higher spatial locality, it could improve the performance for many latency-sensitive applications with poor spatial locality. For our latency optimized memory module, we study DRAM module with relatively smaller rows (32 bytes/row).

Smaller DRAM array rows: When a row in the DRAM is activated, the data is latched in the row buffer. Reducing the row size (to 32 bytes/row) helps reducing the delay in activating the row, and thereby reduce the net latency. The cost we pay is the reduced row buffer hit rate. This memory is suited for latency sensitive applications that have low spatial locality with low memory-level parallelism, where fast access to a small number of consecutive words is more important.

Small prefetch buffer: Smaller prefetch buffer depth could reduce I/O buffer latency due to lesser logic overhead and circuit area. The tradeoff is the reduced memory bandwidth. We use a prefetch buffer of depth `2n`.

Addressing Mode: When compared to DRAM addressing mode, an SRAM addressing mode (implemented in RLDRAM [7]) reduces memory latency. In traditional DRAM devices, the address is supplied in two consecutive clock cycles (command and bank address in the first clock cycle, remaining address in the second clock cycle). This provides the advantage of reducing the number of pins required on the controller side by half. In SRAM addressing, the entire address is provided in one clock cycle which reduces latency. However, power consumption could be higher and pin bandwidth could be lower.

Operating voltage: For this module, we also use maximum voltage(1.8V) to reduce associated circuit delays, and not transition to low-power states often.

4.2.2 Bandwidth Optimized Memory Module (M_B)

Bandwidth refers to the rate at which data can be transferred to or from memory over a period of time. As shown in Fig. 4.2, we allocate pages of a bandwidth sensitive application across the banks in all the modules to harness the benefit of bank-level parallelism. With bank-level parallelism, it is possible to issue simultaneous read/write commands to different banks in a pipelined fashion and achieve higher bandwidth. To further improve bandwidth, we employ a bandwidth-optimized module with relatively a large number of banks (16 banks in a module) and a large row buffer size (512 bytes/row).

Proposition 1: Large burst size

The burst size is directly proportional to the per pin bandwidth. We consider DRAM multiplexed addressing scheme to maintain effective pin bandwidth and pipeline

the command sequences for multiple accesses. A large row buffer size helps in caching more data from the memory array and reduce time required to access consecutive data words.

Proposition 2: Large number of banks/chip

DDR3 supports upto 8 banks/chip. Increasing the number of banks would increase in greater bank-level parallelism per chip. This would in turn result in greater overall bandwidth, assuming that memory accesses are spread out across the banks. We propose upto 16 banks/chip for our high bandwidth memories to support this bandwidth.

It uses DRAM multiplexed addressing scheme (instead of SRAM scheme) to improve effective pin bandwidth. We set the prefetch buffer depth to be `8n` which is the maximum supported by the DDR3 interface. All these design choices could lead to higher latency and power, but provide higher bandwidth.

4.2.3 Power Optimized Memory Module (M_P)

We employ two techniques used in Micron's LPDRAM to reduce refresh power [17]. One is called Partial Array Self Refresh (PASR) where a memory controller selects a portion of memory to refresh. Another is called Temperature Compensated Self Refresh (TCSR) where an on-chip temperature sensor adapts the refresh interval based on the device temperature. In addition, we consider four key design choices to arrive at a low power design. First, we use a small number of banks (two banks per module). Second, we use small row size (32 bytes per row). We choose the smallest possible prefetch buffer size (`1n`) and restrict it to perform only one read/write per cycle. Finally, operation voltage is selected to be the minimum possible (1.5V). All these design choices reduce power but increases memory latency and reduces bandwidth.

The PASR and TCSR are techniques used commonly in mobile DRAM. However,

for a lower bit-density and latency tolerant DRAM, we could use the same techniques in our heterogeneous memory system. The Micron LPDDR is based on this concept. LPDDR uses an on-chip temperature sensor that controls the refresh interval based on the device temperature. TCSR only applies to the self refresh mode and not to the auto refresh mode. For power savings, the PASR feature enables the controllers to select the amount of memory that will be refreshed during the self refresh.

Quoting a power figure is difficult since it depends on access patterns and dynamic operating conditions. Therefore we provide only an estimate for the power by studying the worst case power figures from Micron's DRAM power calculator [10] and use it estimate power characteristic of different memories. In order to obtain the latency or bandwidth numbers for our custom memories, we use the DRAM calculator and scale or estimate the change that we would see, using our modifications.

For low power memories, since latency and bandwidth are not the main concern, we can the row size at a small size of 32-64 bytes/row. We propose to eliminate the pre-charge buffers and read/write only 1 word/cycle. Operation voltage is selected to be the minimum possible (1.5V) for DDR3. Memories M_P is the proposed low-power memory module using these specifications.

4.3 Putting it all together

The baseline system with DDR3 is the typical model used in today's systems. In order to build our system, we select a 3 memory combination (M_L, M_P, M_B) from the different memories proposed in Table 4.1. For latency sensitive applications (*now referred to as type L*), we want to map the pages into module M_L. For bandwidth sensitive applications (*type B*), we distribute accesses across all banks and all modules, to leverage bank-level parallelism. The reason for using all modules here is to ensure that we do no hurt performance by allocating type B pages into only one module. The

power-sensitive application pages (*type P*) are mapped onto module M_P. Our system is flexible enough to accommodate different types of DRAM by changing the various timing and power parameters in the configuration. We accurately model page-faults in the system. This scheme is described in detail in the following section.

CHAPTER 5

Static Profiling of Workloads and Mapping onto the Heterogeneous Memory System

In this chapter, we discuss a static profiling method we use for application classification and mapping onto the heterogeneous memory system. We propose a profiling algorithm to classify application into three categories (L, B, P) based on their memory demand. Two key characteristics of an application define its memory demand: Level-2 (L2) cache miss rate and its Memory Level Parallelism (MLP) [14]. MLP is the degree to which the L2 misses of an application can be serviced in parallel. If a memory operation results in a cache miss, modern processors have the ability to fetch and execute later memory operations [19]. If those later memory operations also incur cache misses, the latency of their cache misses can be hidden by the earlier cache misses.

Two applications with the same cache miss rate may not have the same MLP, as it is dependent on how clustered the cache misses are during an execution and whether one is dependent on the other or not. Typically scientific and graphics applications with data-level parallelism have higher MLP than integer applications with pointer-based data structures. Applications with low MLP are typically sensitive to main memory latency and therefore can benefit from a low latency memory. The

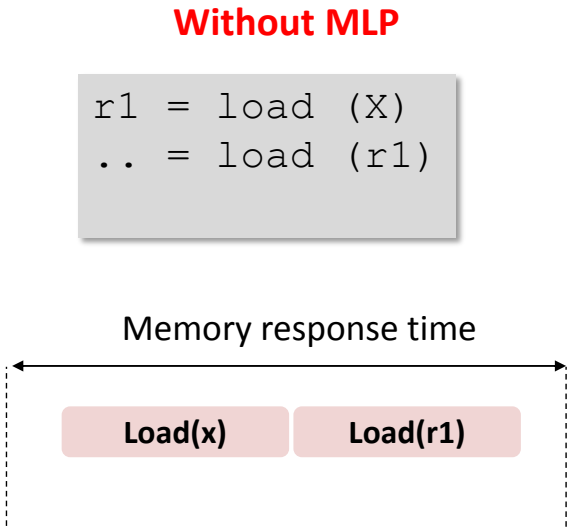

Figure 5.1: Memory accesses without MLP

upper bound on performance for applications with high MLP and high L2 miss-rate is determined by the main memory's bandwidth than the latency. Applications with high MLP and low L2 miss-rate can tolerate high memory latency and do not require high bandwidth. Clearly, for such applications, a low power memory system would be efficient. Based on these observations, we propose a profiling algorithm that uses L2 miss rate and MLP to classify applications according to their memory demands into the three categories.

We also propose an operating system page allocation policy that can allocate pages based on its application's memory demand but also ensures that the page fault rates do not increase significantly.

5.1 Background: Memory Level Parallelism

Applications show different characteristics with respect to memory accesses. The performance of applications is closely linked to the overall characteristics of the memory system. Some applications are sensitive to the bandwidth offered by the memory.

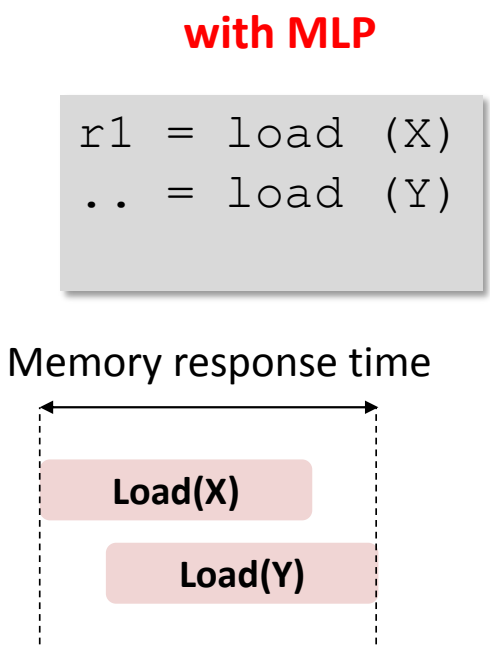

Figure 5.2: Memory accesses with MLP

Whereas others are sensitive to the absolute amount of time required to service a memory request, though the bandwidth may be of a lesser concern here.

Memory Level Parallelism (MLP) is the ability to service multiple outstanding L2 cache misses in parallel. It is a measure of how well we can hide the penalty of consecutive long latency loads.

In order to quantify the MLP of an application, we use the number of cycles an instruction is stalled for at the head of the re-order buffer (ROB), during an L2 miss as a metric. [19] The higher this time, the smaller the MLP for that application. The stress put on the memory system by an application can be captured by measuring the L2 MPKI, which refers to the number of L2 misses per kilo (1000) instructions. It is a measure of the bandwidth requirement of the applications.

$$MLP \propto \left(\frac{stall\ cycles\ at\ ROB\ head\ on\ miss}{\#\ L2\ misses} \right)^{-1}$$

$$L2\ MPKI = \frac{\#\ L2\ misses}{total\ \#\ instructions} * 1000$$

In current computer systems, the main memory (DRAM) is optimized for a particular design point, which typically serves the *common-case* of application accesses well. The spectrum of design points for common memory types is shown in Fig. 4.1. The design points represent the tradeoffs made in the architecture of the memory system. For example, DDR3–1600 has a high bandwidth, but comes at the cost of high latency of access. The RLDRAM [7] type of memory is optimized for high speed network packet accesses, with low latency of access. However, the peak bandwidth of RLDRAM is much lower than the DDR3 family. LPDRAMs are optimized for the mobile application space and consume significantly lower power than their DDR counterparts. The tradeoff made here is the high latency of accesses (compared to DDR2) and lower bandwidth (compared to DDR3). In short, there is no single memory which can serve as the best memory with the lowest latency, lowest power and highest bandwidth. This we call as the *homogeneous system*.

These facts motivate us to design a *heterogeneous memory system*, which can offer differentiated performance points to different applications, in order to maximize overall system performance. **The key idea is:** For a low MLP application, we would optimize it for latency since it would not benefit from a higher bandwidth. If an application has high MLP and a high number of L2 misses, it would benefit from higher bandwidth because of the clustered pattern of memory accesses. A low power configuration can be used to save power for latency-insensitive applications with a low number L2 misses.

5.2 Application Profiling and Operating System Support

In this section, we describe the classification methodology used to group applications in 3 different types based on their characteristics. We also propose an efficient page allocation policy using which an operating system can allocate pages of an application to an appropriate memory module without significantly increasing the page fault rate.

5.2.1 Classifying Applications

An application's type could be determined either at compile-time using profiling or at runtime. We employ offline profiling, but the mechanisms that we discuss can be adapted to engineer a dynamic profiler using processor's performance counters. In our study, we used execution-driven simulations to profile the memory access behavior of applications.

Algorithm 1 Benchmark Classification Algorithm

```
for all benchmarks do
  if (L2 MPKI ≥ thr_1b) AND (ROB stall time ≤ thr_mlp) then
    Type B: optimize for bandwidth
  else if (L2 MPKI ≥ thr_1l) AND (ROB stall time ≥ thr_mlp) then
    Type L: optimize for latency
  else
    Type P: optimize for power
  end if
end for
```

$thr_{1b} = 10$, $thr_{1l} = 0.5$, $thr_{mlp} = 5$

Our offline algorithm classifies applications based on its memory demand which is determined by profiling its L2 miss-rate and Memory Level Parallelism (MLP). MLP is a measure of how well we can hide the latency penalty of consecutive long latency loads. To measure the MLP of an application, we profile the average number of cycles a memory read instruction is stalled at the head of the re-order buffer (ROB) due to

a L2 miss [19]. Higher this stall time, lower the MLP for that application.

Algorithm 1 describes our algorithm. If an application has an L2 MPKI (Misses Per Kilo Instructions) higher than a threshold and high MLP, we classify that application as bandwidth sensitive. If an application is not bandwidth intensive, then we check if the application has L2 MPKI higher than another threshold *and* whether it has low memory-level parallelism (MLP). If an application is neither bandwidth nor latency sensitive, then it is classified as a candidate for power optimization.

5.2.2 Page Allocation and Page Fault Mechanism

The operating system (OS) selects physical pages to store an application's virtual page. We propose a slightly modified LRU page replacement policy to ensure that an application's page is stored in a memory module that meets its demand as much as possible. For `type L` and `type P` applications, the pages should be mapped to their respective modules. For `type B` workloads, its pages should be allocated across all the modules in a round-robin fashion to maximize module- and bank-level parallelism.

Our modified LRU (Least Recently Used) policy works as follows. On encountering a page fault, the OS selects m least recently used pages as potential candidates for replacement. From these m pages, it picks the best victim based on whether it is stored in a module that would benefit the new page's application's memory demand. If none of the m pages are stored in the required memory type, then the OS picks the closest match. If a page in the latency optimized module is not available among the m pages, we select a page in the bandwidth optimized module. If a page in power optimized module is not available, we try to find a page in the latency optimized module first. If both of these attempts fail, then the most LRU page is replaced, irrespective of its type. In our study, we configured m to be 10. This page replacement policy ensures that most pages are allocated on the desired memory module while not significantly increasing the page fault rate when compared to our baseline LRU

Execution core	1 GHz Alpha ISA out-of-order, Dispatch/Issue/Commit width 8, 80 entry ROB, 32 entry LSQ
On-chip caches	64KB split L1, 2-way, 2 cycle hit, 1 read/write port, Unified L2 (2-8 MB), 4-way, 15 cycle hit,1 read/write port, 64B line size, 12 MSHRs
Pre-fetcher	distance of 64, tagged, degree of 2
Memory-core interface	DDR3, 128-bit channel
Main Memory	Baseline: Three 1GB DDR3-1600, LRU page replacement Proposed system: M_L, M_B and M_P modules, Biased LRU (Section 5.2.2)

Table 5.1: System configuration used for simulations with static profiling policy.

5.3 Results of Static Application Profiling

This section evaluates the performance and power benefits of our heterogeneous memory architecture.

5.3.1 Experimental Setup

We use the M5 [20] execution-driven simulator for modeling the performance of the multi-core processor and the system bus. The detailed configuration is presented in Table 5.1. The simulator was augmented with DRAMSim [21] for detailed DRAM timing. It models the DRAM system in detail for each type of memory we studied, by keeping track of the internal states of each DRAM rank, bank and channel. We model memory, bus and memory controller delays and contention as well as ability to service multiple outstanding memory requests. We use an open-page policy in the SDRAMs. Power modeling was done by using activity factors from the simulation and feeding them into the Micron DRAM power calculator [10]. To model the power saving techniques used in M_P, we scaled the worst-case power figures in the DRAM calculator by the potential savings that can be obtained.

Type	Benchmarks
L (latency)	dealII, gromacs, soplex,omnetpp, xalanbmk, astar
P (power)	povray, leslie3d, perlbench,bzip2, zeusmp, libquantum, h264ref, sjeng, sphinx3, tonto, hmmer, gobmk, bwaves
B (bandwidth)	lbm, mcf, milc

Table 5.2: Benchmarks Classification based on Static Profiling

We used the SPEC CPU2006 V1.1 suite of benchmarks [22]. We evaluated 24 of the 31 benchmarks which includes all the benchmarks that we could run on our simulator. We used the available Simpoints [23] to select phases of the workloads. After fast-forwarding to the relevant simpoint and warming up for 1 million cycles, we simulated 100 million instructions in detailed out-of-order mode for each core. For a multi-core, multi-workload simulation, we simulated until every core has executed at least 100 million instructions. but capture the results of simulation of each core after its own 100 million instructions.

5.3.2 Benchmark Classification

Using the profiling algorithm described in Section 5.2.1, we classified benchmarks into 3 types as shown in Table 5.2. Fig. 5.3 shows the L2 Misses-Per-Kilo-Instructions (MPKI) and the average number of processor cycles that a memory operation spends at the head of the ROB. Three benchmarks (*lbm, mcf and milc*) are sensitive to bandwidth due to high L2 MPKI and high MLP (as seen from the low ROB stall time in Fig. 5.3). Six applications (e.g. `dealII`) with high ROB stall time and adequate L2 MPKI are classified as latency sensitive. The rest are classified as candidates for power optimization.

5.3.3 Workload Mixes and System Setup

We evaluate for 4, 8 and 16 core configurations, each with 12 different workload *types*. A workload *type* is determined by the number of Latency (L), Power (P) and

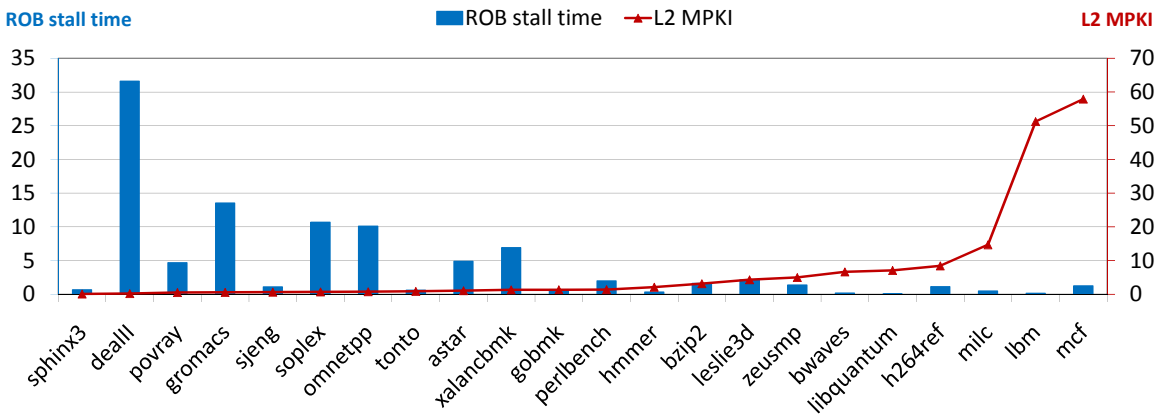

Figure 5.3: Bar graph: Number of machine cycles spent stalling at head of the ROB per L2 read miss. Line Graph: L2 MPKI

Bandwidth (B) applications in the workload. `1L4P4B` stands for a workload type with 1 latency sensitive benchmark, 4 power optimized benchmarks and 4 bandwidth bounded benchmarks (determined in Section 5.3.2). For each workload type, we consider 10 different workloads (mixture of different benchmarks selected based on their type), thus yielding a total of 120 workloads for each of the three processor configuration.

We use the *weighted speedup* to measure the performance of a multi-core system. In the metric described below, IPC^i_{shared} and IPC^i_{alone} denote the IPC of the i^{th} application when running in a shared multi-core environment with others, and running alone respectively. The weighted speedup metric measures the overall reduction in execution time, by normalizing each application's performance to its IPC value when run alone.

$$W.Speedup = \sum_n \left(\frac{IPC^i_{shared}}{IPC^i_{alone}} \right)$$

5.3.4 Benefits of Heterogeneous Memory

Fig. 5.4 and Fig. 5.5 shows the improvement in weighted speedup and power for a 4-core system using a heterogeneous memory (M_L, M_P, M_B) when compared to

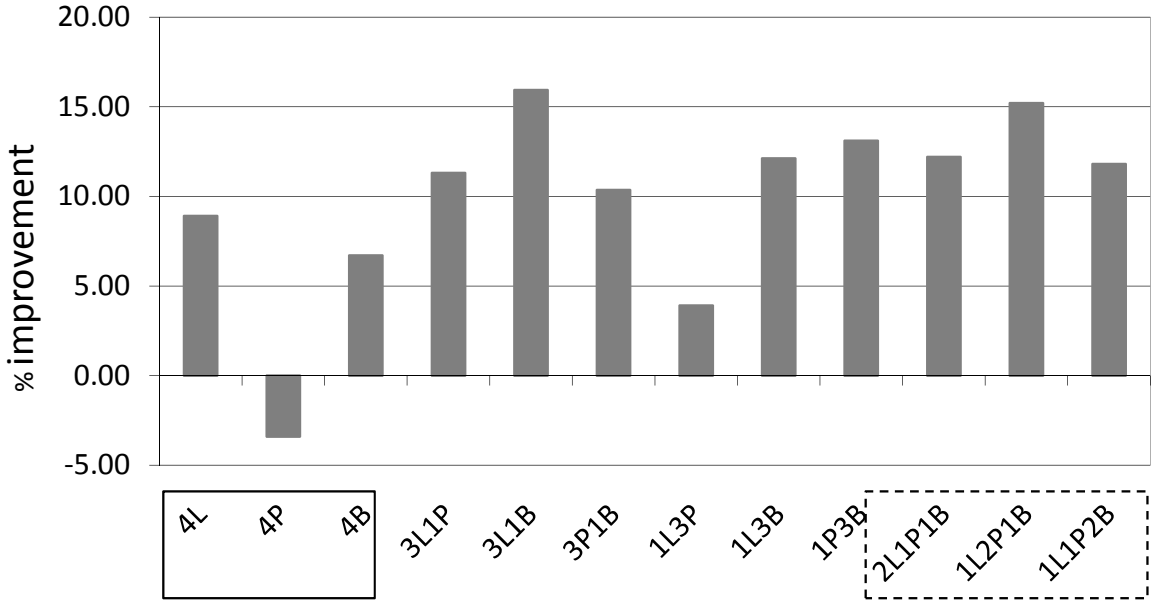

Figure 5.4: Performance of heterogeneous memory system over DDR3 homogeneous memory for a 4-core system.

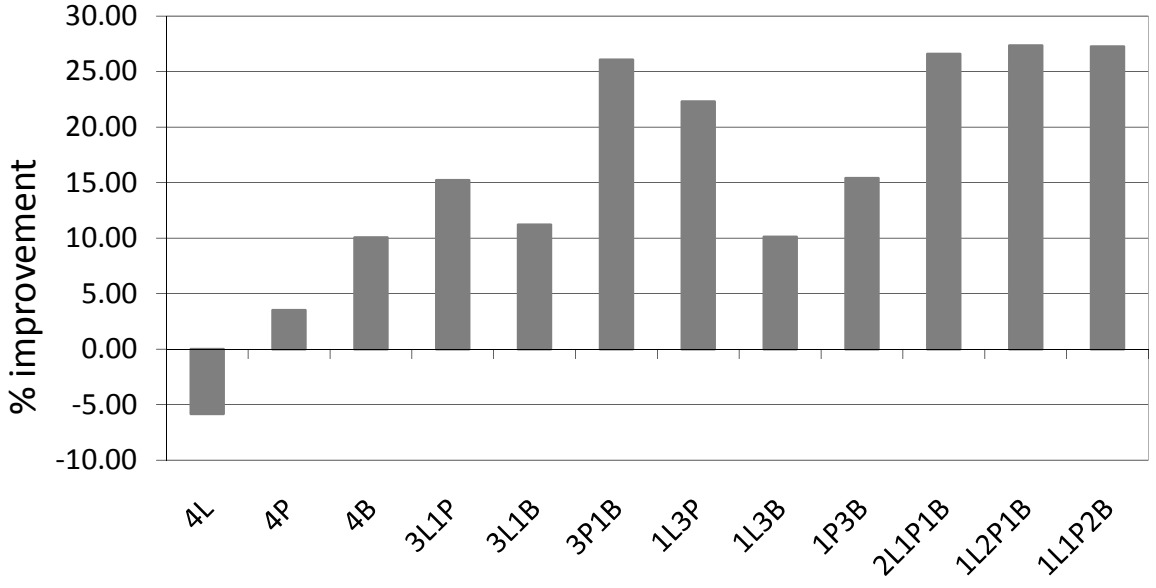

Figure 5.5: Power improvements heterogeneous memory system over DDR3 homogeneous memory for a 4-core system.

the DDR3-1600 homogeneous memory. We show results for 12 different workload types. For each workload type, we present the average improvement observed for the 10 workload mixtures.

We observe improved efficiency for most workload types. Maximum gains are observed for types with roughly equal distribution among application types (last three on the right-hand side) – 16% performance, 26% memory's power. Our worst case is when application mixture is heavily imbalanced (leftmost 3 workload types). Even so, the maximum performance penalty noticed is only 3% (`4P`). We believe such imbalance in workload mixture would be rare in most user scenarios. On average our system improves performance by 13.5%, memory's power by 20.01% averaged across 120 different workloads for a 4-core configuration.

Fig. 5.6 shows results for 4, 8 and 16 cores (average over 120 workloads for each configuration). We find that heterogeneous memory retains its advantages as the number of cores scale. We find on average performance improves by 14% (16-core) and memory's power reduces by 17% (16-core).

We measured the efficiency of our page allocation method (Section 5.2.2) for a 4-core system. For only about 1% increase in page fault rate, our policy was able to allocate pages on the required memory type for most pages. Only 2.35% memory accesses were to a page that was allocated on a memory type that was not optimal.

5.4 Conclusions

Most computer systems already use multi-core processors. Applications that could run simultaneously on multi-core systems have varying needs from the main memory system. The proposed heterogeneous memory system provides better latency and bandwidth based on an application's memory demand. Our cycle accurate simulations demonstrate significant performance and power benefits of the heterogeneous

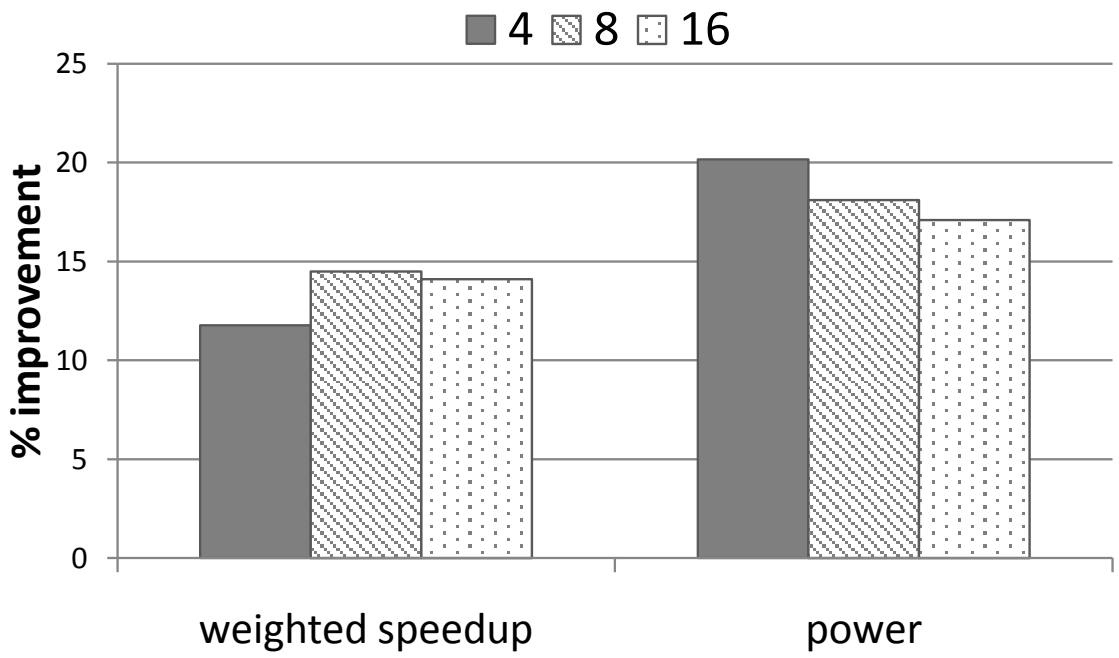

Figure 5.6: Improvements in weighted speedup (WS) and memory power for 4, 8 and 16-core system due to Heterogeneous Memory.

memory.

CHAPTER 6

Dynamic Policy: Application Slack-Aware Scheduling

In this section, we discuss the scheduling technique used in the memory controller (MC) to improve the performance of our heterogeneous memory system.

As observed in static profiling, the memory level parallelism (MLP) is not uniform across all memory accesses in a program. The distribution of L2 cache misses hitting main memory is dependent on the application. For application which have a regular pattern of memory access, we observe a clustered set of misses with high potential of overlap. For others, the L2 misses are spread out. The classification we used for static profiling is good at distinguishing between MLP of different applications in a broad sense. However, applications also show various phases of execution where the memory access pattern may vary greatly between phases. [24]. In complex multi-core systems, with large number of applications, the application performance is dictated by various parameters like MLP, burstiness of the accesses observed at the memory controller, spatial locality of data, etc. An important factor that controls the performance at the memory controller level is the scheduling policy used. Traditionally, memory controllers use a *First-ready-first-come-first-serve* (FR–FCFS) scheduling policy. [25, 26]. This policy prioritizes the memory requests in a way to maximize row-buffer hits in the

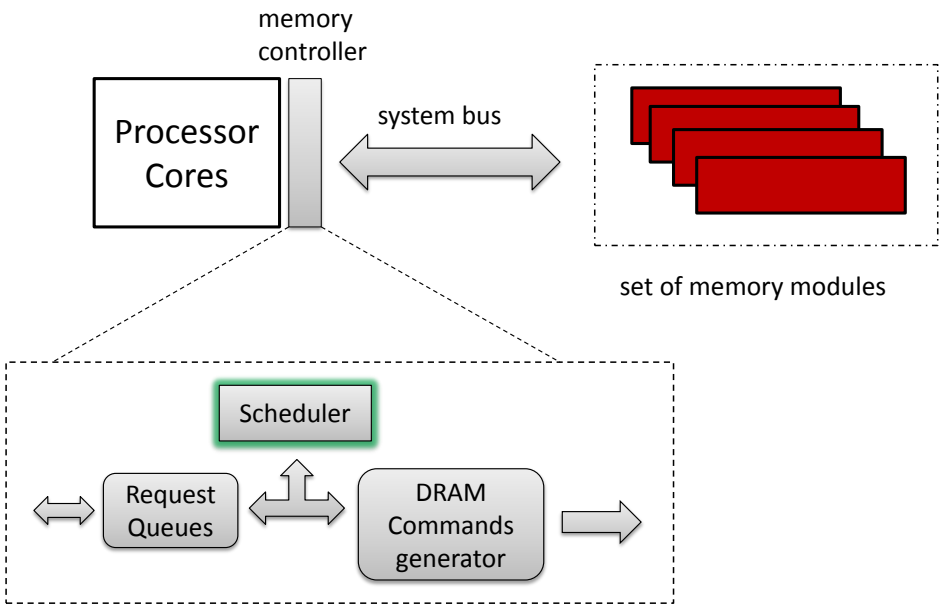

Figure 6.1: Memory controller block diagram

DRAM. While this policy is simple to implement in hardware and efficient for single threaded systems, it is not optimal for multi-threaded or multi-workload systems. This is because it has no information about the interference between different threads or workloads.

In this chapter, we use to concept of application level slack to improve the performance of the memory controller in our heterogeneous memory system design. In the context of memory accesses, slack is defined as the amount (number of cycles) by which a memory instruction can be delayed without affecting the execution time of the application. A simple scheduling policy gives higher priority to requests with low slack. This captures the inherent *criticality* of the memory request and aims to improve overall performance of the memory system.

6.1 Memory Controller Scheduling Policies

Long latency memory read operations can significantly impact the performance of processors. When a long latency load reaches the head of the re-order in the processor, it is stalled till the request is serviced and the data sent back to the processor. This ensures that instructions are committed in-order. Hence, the scheduling policy at the memory controller can determine the amount of stall cycles and overall performance of the system. This is more pronounced in a multi-core, multi-workload setup because satisfying the requirements of one application may significantly impact another.

The DRAM chip architecture is described in Section 2. The multiple banks within the DRAM help in servicing multiple memory requests in parallel (*bank-level parallelism*). Each DRAM bank consists of columns and rows in a 2D array fashion. A DRAM row is also called a *page*. DRAM row size can range from 256B-2KB, mapped to a consecutive address range. Before data can be read from the DRAM, the row must be activated and the row is said to be *open*. The data from a specific row is latched into the row buffer. A bank contains a single row buffer. Once the data is latched, a particular column of data can be read out into the I/O buffer. A row holds several consecutive cachelines of data. Once the required data is read out, the row buffer could either be kept in the current state (open page policy) or the data could be written back into the DRAM row cells (closed page policy). In terms of latency, opening and closing a page is expensive, while transferring data out of the row buffer onto the memory data bus is comparatively cheap. There is also a minimum allowed time between opening and closing a page (the minimum activate-to-precharge latency).

The amount of time it takes to service a DRAM request depends on the address fields in the memory request and the status of the row buffer. Every time a bank needs to be switched, there is an associate switching time penalty. Similarly, switching to a different row within the same bank involves a delay. The following conditions affect the time required to read out of the DRAM array:

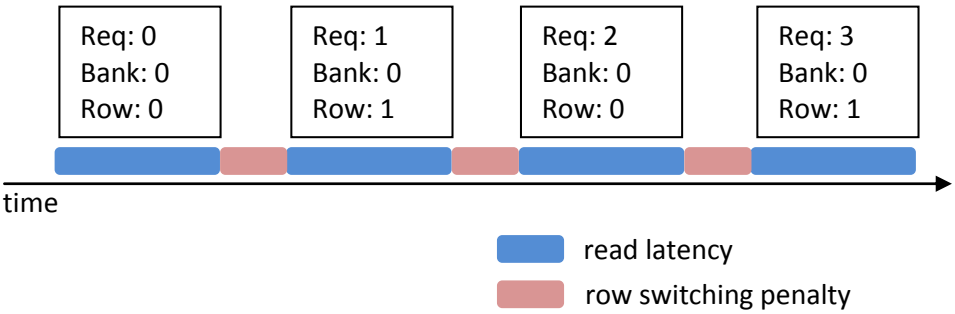

Figure 6.2: First Come First Serve (FCFS) Scheduling Policy

- Row-hit: If the incoming memory request reads from a currently open row, the delay associated in read out from the DRAM array cells and latching into the row buffer is eliminated. The only delay is for the associated column select, resulting in a read latency of t_{CL}.

- Row-miss: If the incoming memory request is to a different row than the one currently open, there is a delay associated with closing the current row and activating the second one. This required the DRAM controller to issue a row precharge command to the second row and then an activate command. This access incurs a read latency of $t_{RP} + t_{RCD} + t_{CL}$.

6.1.1 First Come First Serve (FCFS) Scheduling Policy: Baseline Policy

The simplest memory scheduling policy is the *first come first serve* (FCFS) policy. In FCFS, requests are serviced in the order they appear at the memory controller. This is simple to implement in hardware with very small circuit area required. For applications or a mix of applications where the memory access pattern is uniform and spread out over all the banks, this policy may suit well. That is because it will use the bank-level parallelism inherent in the application memory requests. However, the observed memory access patterns of applications are not always uniform and may

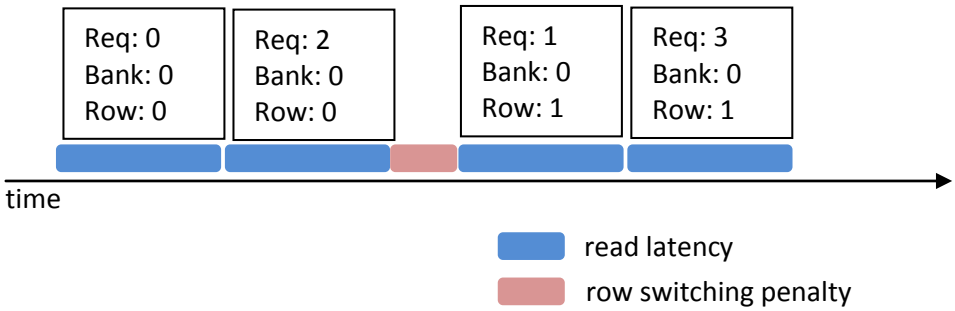

Figure 6.3: First-Ready-First-Come-First-Serve (FR–FCFS) Scheduling Policy

exhibit poor bank-level parallelism requiring only a subset of the banks. FCFS is not priority-aware or thread-aware. It has no knowledge about the completion time of the various memory requests. It can unfairly penalize short requests from one thread (which may be time-critical) while servicing a burst of long requests from another (which may be latency tolerant). Secondly, if there are too many many consecutive requests going to different rows, a penalty is incurred every time a row is switched. This is shown in Fig. 6.2. In this example, the 4 memory requests alternate between using rows 0 and 1 and incur a switching penalty between requests.

6.1.2 Row-Aware (R): First Ready - First Come First Serve (FR–FCFS)

The FCFS scheduling policy described above is non-optimal in many situations. It could result in too many bank or row level switchings to be performed. It could also penalize short but critical requests from one thread/application because of previous long but non-critical memory requests from another. These situations commonly arise when request streams from different threads/applications running in a multi-core system become interleaved. The FR-FCFS scheduling policy re-orders memory requests to improve row hits based on row locality of requests. It checks if a DRAM row is currently open (ready) to serve requests. If so, it first services requests which

hit that open row, ahead of others. As seen in Fig. 6.3, this scheduling policy would schedule **req 2** before **req 1** in the previous example. This reduces the numbers of row-switching delays to only 1 in this case compared to 3 for FCFS. This improves the overall row hit-rate and memory service throughput.

However there are certain drawbacks with FR-FCFS. In a multi-core, multi-workload system, FR-FCFS may prioritize request streams with a high row hit-rate over one with a lower hit-rate. This results in unfairly prioritizing memory intensive workloads over non-intensive ones, even though the non-intensive memory requests may be critical to the completion time of that workload [13, 25]. As a result, even though FR-FCFS achieves high DRAM data throughput, it might starve requests or threads for long time periods, causing unfairness and relatively low overall system throughput.

FR–FCFS is the most commonly used scheduling mechanism in contemporary memory controllers.

6.1.3 Application Aware (A): Prioritize low-latency packets

In an application aware scheduling policy, we prioritize application packets based on the application type (low latency, power-sensitive, high bandwidth). The basic idea is that low-latency packets are more critical than low power which are more critical than high bandwidth. For high-bandwidth packets, the overall throughput determines the performance more than a single packet's latency of access. Hence the prioritization rule is: low-latency, power-sensitive, high-bandwidth, in that order. In case there are multiple packets of the same type, we follow a FCFS policy within them.

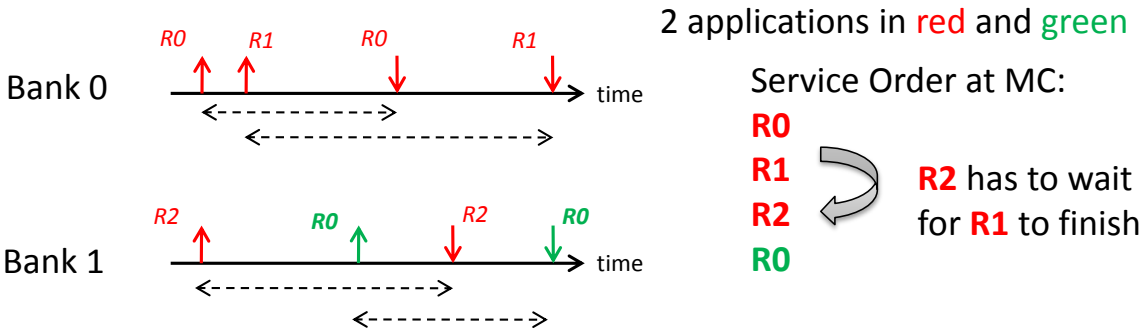

Figure 6.4: Slack-Unaware Scheduling

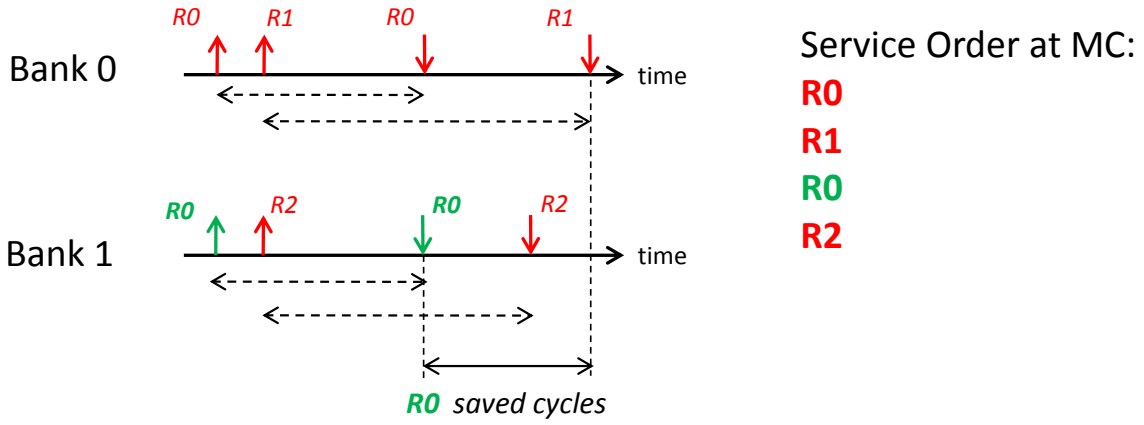

Figure 6.5: Slack-Aware Scheduling helps reduce wait time

6.2 Motivation: Slack in DRAM Memory Requests

A slack aware scheduling algorithm was introduced in [27]. Their architecture, Aérgia, introduces and exploits a key concept: the *slack* associated with the latency of a packet in a network-on-chip (NoC). The technique is applied in a critical shared resource environment, shared by many different classes of applications with different requirements. Exploiting packet slack to optimize scheduling improves overall throughput and fairness in the system. The slack can also be calculated dynamically, unlike previous techniques described in [28, 29] which perform static priority assignment.

The *slack* of a memory request is defined as the amount (number of cycles) by which a memory instruction can be delayed without affecting the execution time of the application. [27] When multiple requests are pending at the memory controller, the controller must pick a request to schedule in the next cycle. Intuitively, we can think of slack as inverse of the *criticality* of a memory request. The higher the slack for a memory request, less critical the request. This means that the request could be delayed by a certain amount of time(cycles) without affecting the overall execution time of the remaining instructions of that application. This delay tolerance can be mapped onto a scheduling policy. If there is a memory request with low slack, it is more critical and should be scheduled over requests with higher slack. In this way, we can capture the changes in the phase of an application at the memory controller level.

Fig. 6.4 shows a motivating example. In this example, we have 2 applications, `A0` (in red) and `A1` (in green). `A0` has generated 3 requests. Out of these, `A0-R0` and `A0-R1` goto Bank 0 while `A0-R2` goes to Bank 1. `A1` has generated only 1 request (which may be latency critical), `A1-R0`. With FCFS or FR–FCFS scheduling, the order of requests being serviced would be as shown in the figure. Here, even though `A0-R2` has to wait for `A0-R0` and `A0-R1` to finish, it is serviced before `A1-R0`. For application `A0`, even if `A0-R2` would be delayed by the amount of time shown as $slack_{A0-R2} = t_{A0_R1} - t_{A0_R2}$, (where t refers to the arrival time), the overall execution time of application `A0` would

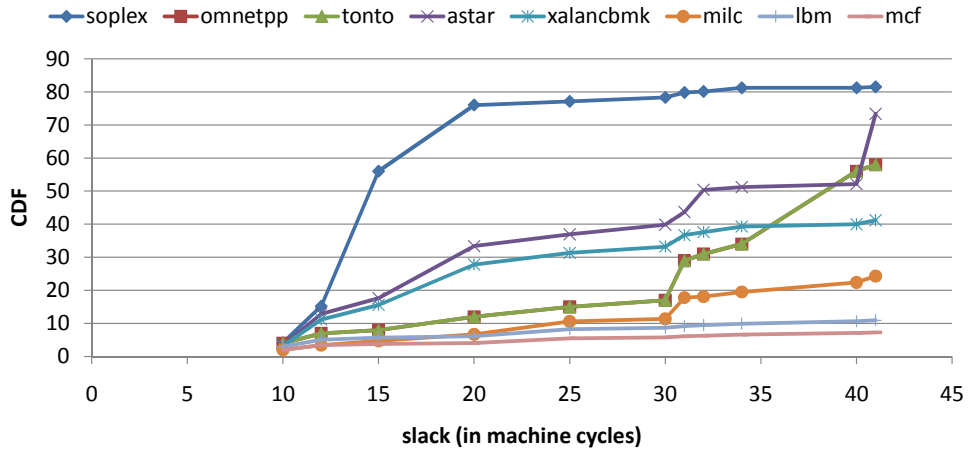

Figure 6.6: Available Slack Distribution for Applications in SPEC CPU2006

be unaffected. However, `A1-R0` has to unnecessarily suffer a higher latency of lat_{A1-R0}, because of being scheduled after `A0-R2`.

If we could estimate the slack for the requests, we could schedule the request `A1-R0` before `A0-R2`. This is shown in Fig. 6.5. Doing so reduces the latency seen by `A1` to lat'_{A1-R0}. The increased latency for `A0-R2` reduces its slack to $slack'_{A0-R2}$. However, this still keeps the execution time for `A0` within the earlier range (does not *significantly* increase execution time). The result is `A1` benefits from a reduced latency of access with this out-of-order scheduling.

6.2.1 Slack Diversity

A predecessor request is any pending memory request (from the same application) queued up at the memory controller. For a given memory request, the slack is given by:

$$slack_i = \max\left(E\left(T_{i-n}\right)\right) - E\left(T_i\right) \qquad n \in [0, p]$$

where p is the number of predecessor requests and T_i is the time to completion of the i^{th} request.The oracle slack calculation would involve perfectly calculating the slack of all preceding requests. However, this is computationally intensive and in many cases

not possible without doing a critical path analysis. It is possible to characterize the slack and determine the factors which affect slack. Using this, we can estimate the slack of a memory request and quantize it broadly into 2 levels: high slack and low slack. This methodology is explained in the next section.

6.2.2 Characterization of Slack

Calculation of the slack of a memory request at the memory controller requires estimating the completion time of predecessor requests. If there are many requests *preceding* the incoming one, it is highly likely that the incoming requests' latency could be hidden due to overlap with the predecessors. This makes the incoming request latency tolerant and has some amount of slack in it. On the other hand, if a request has few or no *predecessor requests*, then its latency cannot be effectively hidden and hence it would exhibit a very low amount of slack.

In order to predict the completion time of the predecessor requests, we examine various characteristics of memory requests. One indicator of high amount of slack is the number of predecessor requests. Higher the number of predecessors for a request, more likely for its latency to be hidden. The amount of slack estimated is the product of the number of predecessors and the average DRAM latency ($p.T_{avg}$).

At the DRAM level, we examine the state of the memory array. We look at the module, bank and row IDs of the predecessor requests. If the request corresponds to the bank and row which is currently open, the read latency will simply be the `CAS` latency. However, if we have to switch to another row, the current row must be closed and a new one opened. We call this portion of the estimate latency as the *state latency* (T_{state}).

1. **Same Bank, Row:** t_{CL}

2. **Same Bank, different Row:** $t_{RC} + t_{CL}$

3. **Different Bank, Row:** $t_{BL} + tRC + t_{CL}$

Using these metrics of memory access latencies, we can estimate the time to completion of a request

$$E\left(T_i\right) = p.T_{avg} + T_{state}$$

Fig. 6.6 shows the cumulative distribution of the available slack for certain SPEC CPU2006 benchmarks. In this figure, we plot the amount of observed slack on the X-axis. The Y-axis shows us the percentage of requests with atleast the amount of slack on the X-axis. We see a fairly wide distribution in the available slack across different applications. This diversity in the available slack is the key point to exploit the slack at the controller. If there are no high-slack requests, then there would be little benefit of scheduling the low-slack requests ahead.

6.2.3 Slack Aware Scheduling (S): Prioritize low-slack packets

Our slack based scheduling policy prioritizes packets at the MC based on the estimated slack. At any given time, multiple memory requests could be pending at the MC. In order to schedule the next request, MCs use a scheduling policy. The MC policy has to be simple enough so as not to incur too high a cost of hardware implementation. In our system, we describe a memory controller policy which aims to observe the dynamic behavior of workloads and prioritize memory requests at the MC to improve performance where possible.

Algorithm 2 Slack Based Scheduling (S): rules

0: **Threshold:** Schedule low slack requests over high slack ones
0: **FCFS:** Under same level of slack, follow a FCFS policy

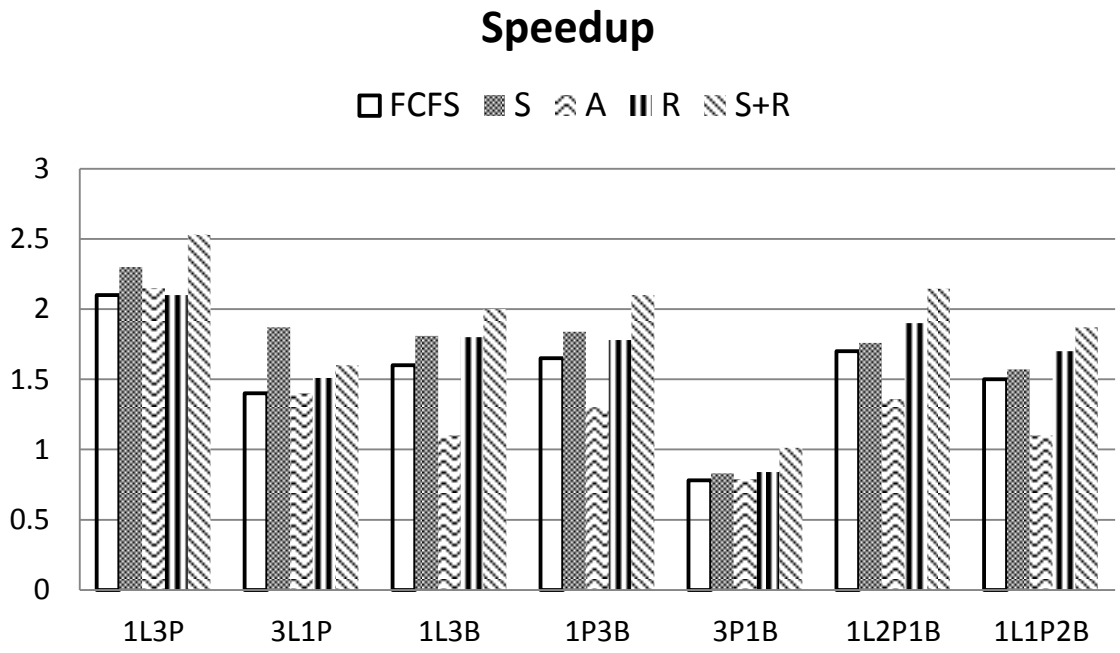

Figure 6.7: Speedup

6.3 Results of Slack-Aware Memory Request Scheduling

We evaluate these policies for a 4 core system. The different policies we consider and the slack (S), application (A), row-first(R). In addition, we create combinations betwenn them (example: S+R). For each workload type, we consider 10 different workloads.

Fig. 6.7 shows the improvement in weighted speedup for a 4-core system using a heterogeneous memory (M_L, M_P, M_B) when compared to the DDR3-1600 homogeneous memory. The application only aware policy (A) performs poorly in terms of speedup improvement over the stock FCFS. The only case is 1L3P, where it shows a slight improvement. The row-first (R) policy for the most part performs similar to FCFS. It however, shows better speedup in workloads with a higher number of high-bandwidth workloads. This is because with more B types workloads, the R

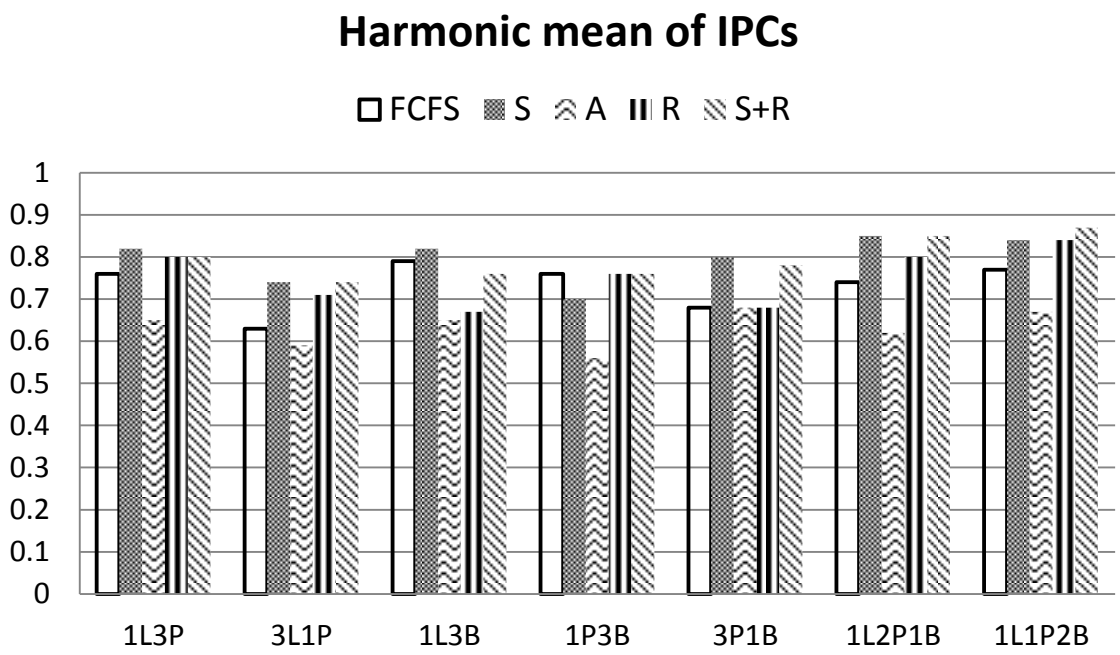

Figure 6.8: Harmonic mean

policy maximizes row-hot and hence row-level parallelism. The slack-aware (S) policy out-performs most policies in a mix of L,P, B workloads by upto 12%. In cases of higher number of latency-sensitive workloads, it performs the best (3L1P. However, if there are no L types workloads (3P1B), the speedup is not improved much. Combining S and R (S+R) achieves the best performance overall in our system.

Fig. 6.8 shows the harmonic mean of normalized IPCs. The closer to 1, the better is the fairness in the system. From this figure, we observe that the S+R policy achieves greatest overall fairness compared to the other policies. However, when we have a a workload skewed towards B-type applications (1L3B), the S policy outperforms S+R in terms of fairness.

We also study the maximum slowdown experienced in the system, shown in Fig. 6.9. Overall, the various policies slow down the system with a maximum of 4.8%. The S+R policy performs best overall in minimizing the maximum slowdown. This is another indicator fairness in the system.

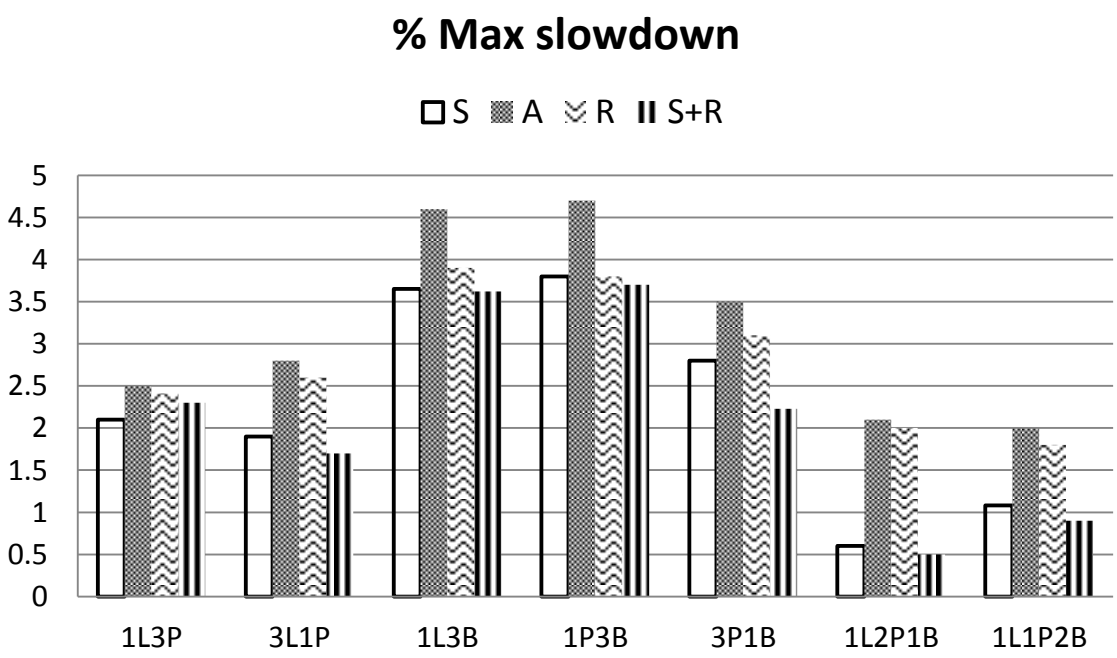

Figure 6.9: % max slowdown

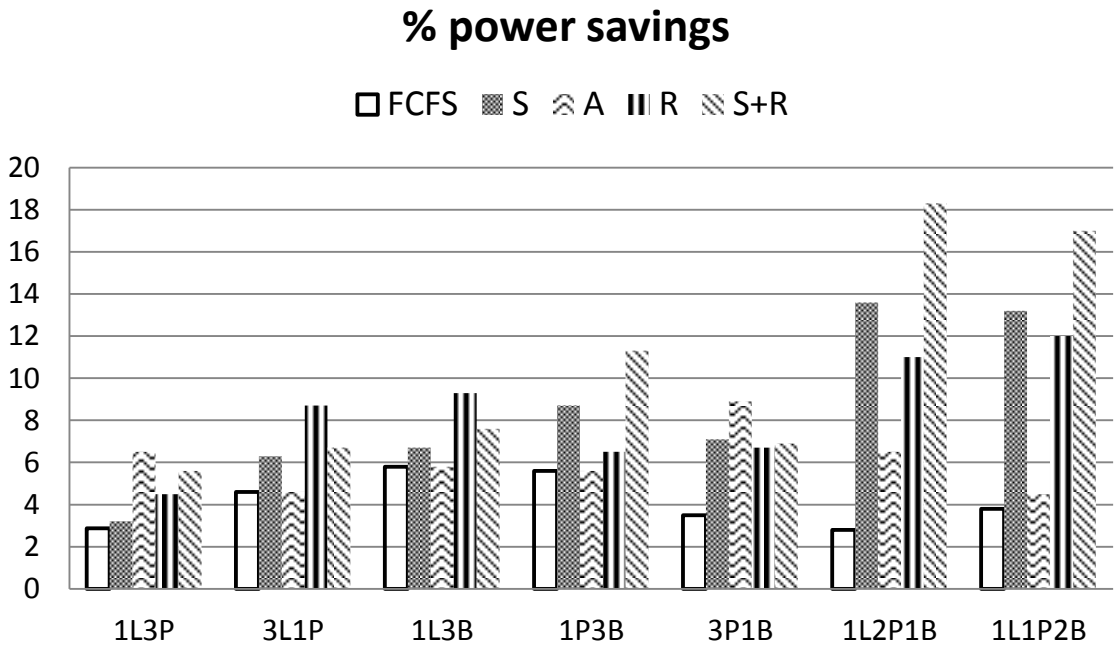

Figure 6.10: Power

Fig. 6.10 shows the power benefits of the various scheduling schemes. The row-first (R) policy is adept at minimizing the power consumption by maximizing the number of row hits. This works well in case of 3L1P, 1L3B, 3P1B. The R policy becomes less effective with a good mix of applications. In case of 1L2P1B or 1L1P2B, the slack only (S) or the S+R show better power savings. The application only (A) policy reasonably well in case of a high number of power-sensitive applications (3P1B or 1L3P) but poorly in others.

CHAPTER 7

Future Directions

7.1 Intra-application heterogeneity

Applications also show various phases of execution where the memory access pattern may vary greatly between phases. [24]. During a window of time, a program may show high bandwidth requirements from the main memory. In another window, it may have much fewer memory accesses. The phases an application exhibits are not necessarily random but can exhibit repeating patterns. They are closely linked to the data structures of the code and the flow of information within the application.

The MLP–based classification we proposed works statically, at the application level. This averages out the effect over some 100 million instructions. However, given the different phases of applications, an application could be thought of as behaving as either a high bandwidth (**B**), latency-sensitive (**L**) or a power-sensitive (**P**) one in the different windows of execution. If we could extract this information and pass it onto our system, we could further classify different pages of the application at the intra-application level. The key idea is to identify and exploit heterogeneity of memory accesses within the pages of the application.

At the global level, we could have classified an application as bandwidth intensive (**B**). However, it is possible that certain pages are cold, in the sense that they are not

accessed frequently. Keeping these pages in the high-bandwidth module only wastes power, without any significant improvement in performance. Similarly, an application which is observed to be latency sensitive globally, may have data stored in pages which are not accessed often. Keeping these pages in the low-latency module wastes power and also takes up space for truly latency sensitive pages of other applications.

Given a classification of an application at the global level (say **L** or **B** type), we want to measure the trend for its pages. If pages are cold, we want to instead keep them in the low-power module. The other pages would be mapped as before, based on the application's type.

A accurate estimation the maximum memory requirement of an application and access patterns at compile-time is difficult. Data in applications is typically stored into three logical memory segments: global, stack and heap data. The global segment has a fixed size and is possible to determine at compile time. The stack and heap segments grow at run-time and are comparatively harder to estimate. However, we propose that data in the global and stack segments could be classified based on the original, global application classification (**L**, **P** or **B** type).

For the data in the heap, we use a different approach. The heap segment starts at the end of the data segment and grows towards larger addresses. The heap segment memory management is handled by *malloc(), realloc(), and free()* functions (or equivalently by the C++ operators *new* and *delete*). These functions may make the **brk** and **sbrk** system calls (or use *mmap*) to adjust space requirements of the application. The *callsite* of a function is the line in the code which calls a function. A function called from multiple callsites will generally exhibit similar behavior with respect to memory access patterns. Hence, if the pages accessed by a function from one site are infrequently accesses, they could be placed in the low-power module. This could also apply to all other pages from other callsite accessing the same function. Based on this information, dynamically allocated pages' behavior can be guessed. This

would help in determining a location for these pages (low-power modules or original global classification).

We hope that this hybrid classification scheme will be important in systems with large number of cores and limited, shared memory. This is an area where we see an opportunity for our heterogeneous memory architecture. If we could classify application needs at this lower granularity, our system could provide performance and power efficiency.

7.2 CPU + GPU: Towards Combined Future Cores

The use of graphics processing units (*GPUs*) for general purpose computing is an active area of research. The design space in discrete GPUs is dominated by NVIDIA, AMD (formerly ATI). Intel is the major player in the integrated GPU (on-chip) design space. Discrete graphics cores are optimized for graphics operations and consist of large number of very simple cores. GPUs can perform large amount of vector processing which is required for high-end graphics. This however comes at the expense of high power consumption. Secondly, only a fixed set of algorithms can be efficiently mapped onto a GPU core. A architectural limitation of current GPUs is the absence of coherent shared caches. This translates into high amount of memory traffic between the GPU core and the graphics main memory (*GDDR*).

General-purpose GPU computing (*GPGPU*) is the use of a GPU as a general purpose computing core. The idea is to add a GPU core to a traditional computational CPU core. A heterogeneous model like NVIDIA's CUDA architecture [30] combines a GPU with a CPU. Intel's Larrabee was proposed as a hybrid CPU and GPU, consisting of multiple in-order CPU-style processors. AMD's Fusion [31] processor pairs a CPU and GPU on the same processor die with a on-chip north-bridge memory controller to communicate with external memory. In any hybrid CPU and GPU architecture, the

design of memory system would play a key role in overall performance. In Larrabee, Intel uses a hybrid bidirectional *ring bus*. This gives it the flexibility to run dynamic algorithms like ray tracing.

There are many interesting sub-problems that crop up here:

- **Memory traffic characteristics:**

  With a CPU and GPU on the same die, the memory traffic between this core and the memory system will show significant changes. In addition to different phases for applications running on the CPU, we now have phases of CPU and GPU memory traffic. CPU traffic tends to be more random compared to GPU traffic. The challenge lies in correctly classifying applications in this combines space and the modifications required to ensure the heterogeneous system still leads to an improved performance with power savings.

- **Error Tolerance Based On User Perception:**

  In computer graphics and animation, a range of motions for a high level of realism is obtained through motion picture data. In the processing of these motions, minor details are often unnoticed by the average viewer. Such 'errors' can be hence be tolerated. A technique for measuring the sensitivity to perceived errors and quantifying them is described in [32].

  Sheaffer et al. [33] describe the problem of transient faults in graphics. In their work, they characterize the kind of faults from which the system needs protection and present simple recovery schemes. We would like to apply these in the context of our proposed heterogeneous memory system. A cost effective way to maximize system performance for a CPU+GPU system in which non-perceived errors can be ignored is an important future direction. There are 2 directions which we need to explore:

  Page Criticality:

In GPU processing, not every page is equally critical. In GPU implementations, much of the data transfer to and from memory may occur only at the beginning and end of an algorithm. Global memory is not cached on GPUs. If we can determine that certain categories of pages are less critical than others, we could map them into the low-power module to conserve power and bandwidth for the other critical pages.

Physical Design Limits:

GDDRs have tighter timing contraints compared to DDRs. For example, GDDR can send control signals and receive data on the same clock cycle, whereas DDR cannot do that. GDDR has higher heat tolerance compared to DDR which necessitates better cooling systems. An interesting area for us to look at is if these limits have been thoroughly visited. We want to investigate if we could apply tighter timing controls on our memory modules to improve performance. The maximum frequency (overclocking), minimum cycle time and maximum temperature that it can be operated at need to be investigated and quantified.

- **Address Space:**

  With CPU and GPU sharing the same address space, we need to analyze if a equal sized heterogeneous system would work. The ratio of 1:1:1 for the 3 different types of modules was found to be suitable for CPU traffic alone, but this may not be optimal for the GPU case. We may have to allocate more for latency critical data required by GPU threads. The memory alignment also matters. For GPU traffic, the way in which arrays have been mapped onto physical memory is over wider boundaries than CPU traffic.

We believe that these directions will open up a new avenues of thinking about heterogeneous memory systems. By providing a differentiated level of service to

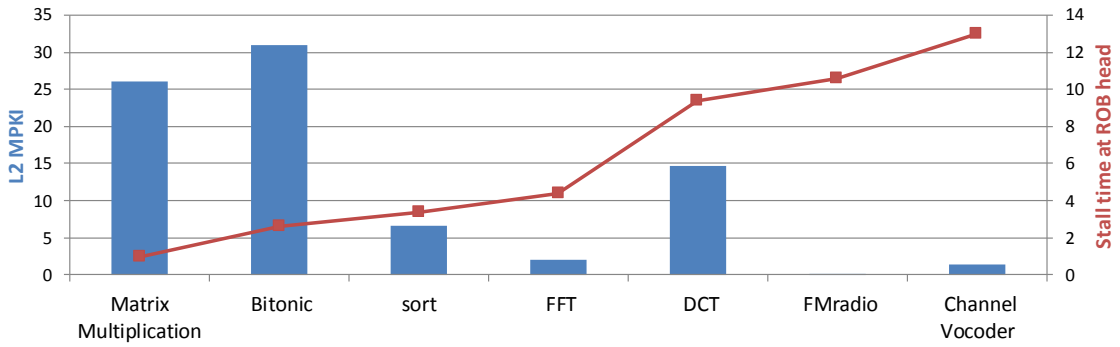

Figure 7.1: Rob stall time and L2 MPKI for StreamIt workloads

different applications in a variety of design spaces, heterogeneous memory systems can provide a good performance boost and power savings.

In the next section, we describe some preliminary results with GPU workloads.

7.2.1 Graphics Workloads

StreamIt [34] is a data flow language which helps in writing optimal streaming applications code. The StreamIt compiler does automatic optimization and partitioning of computation across multiple processing elements.

We use some of the benchmarks from the StreamIt [34] framework. The StreamIt framework defines transformations of data using greedy heuristics for maximizing parallelism. The memory access patterns of these benchmarks vary from the Spec2006 benchmarks in the distribution across memory banks and row hit ratios. The various applications in the suite involve the processing synchronous information flow.

7.2.2 Static Analysis: Preliminary results

The representative set of programs from the StreamIt benchmarks used are: Matrix Multiplication, Bitonic, sort, FFT, DCT, FMradio, Channel Vocoder. We plot the L2 MPKI and ROB stall time for these workloads in Fig. 7.1. As can be seen from

Bandwidth	Matrix Mult, bitonic, DCT
Power	sort, FFT, FMradio, channel

Table 7.1: Graphical Workloads Classification

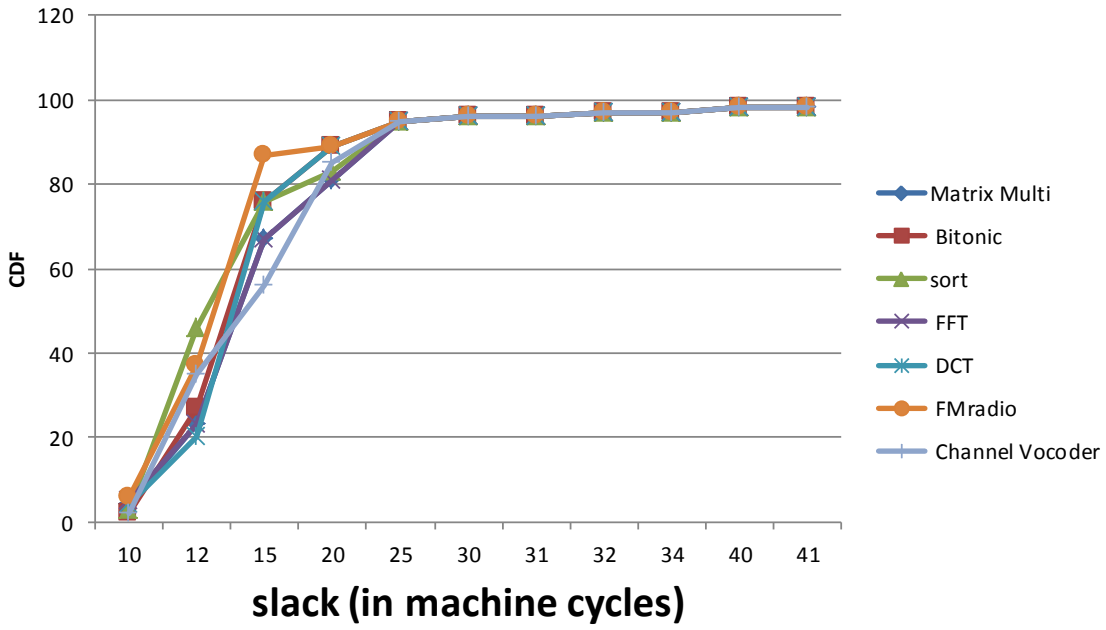

Figure 7.2: Slack Characterization for StreamIt workloads

Fig. 7.1, these workloads do not show a clear case of latency-sensitive workloads. We do see bandwidth intensive workloads and others which we can optimize for power. We can classify these applications as follows:

In order to optimize the accesses for these memory requests, we study the slack characteristics.

We characterize the slack for graphics workloads in the same way as described in Section 6.2.2. The available slack in the workloads is shown in Fig. 7.2. Most of the workloads (close to 80%) show a slack of less than 20 cycles.

We want to use these results and introduce a mix of CPU and GPU workloads and study their behavior in our heterogeneous memory system. Our algorithms and threshold conditions may require tuning to achieve the best results.

BIBLIOGRAPHY

[1] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34:52–58, April 2001. 1

[2] Intel® Core™ i7-900 Desktop Processor Extreme Edition Series and Intel® Core™ i7-900 Desktop Processor Series on 32-nm Process, 2010. http://download.intel.com/design/processor/datashts/323252.pdf. 1

[3] Family 10h AMD Phenom™ II Processor Product Data Sheet, 2010. http://support.amd.com/us/Processor_TechDocs/46878.pdf. 2

[4] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36:39–48, December 2003. 2, 10

[5] Taeho Kgil, David Roberts, and Trevor Mudge. Improving NAND flash based disk caches. In *Proceedings of the 35th International Symposium on Computer Architecture*, pages 327–338, 2008. 2, 10

[6] Bruce Jacob, Spencer Ng, and David Wang. Memory systems: Cache, DRAM, disk. In *Elsevier*, 2008. 4, 10, 12, 13

[7] Reduced latency dram (RLDRAM). http://www.micron.com/products/ProductDetails.html?product=products/dram/rldram/MT49H16M18FM-33. 6, 14, 26, 33

[8] Calculating memory system power for DDR3, Technical Note TN-41-01. http://download.micron.com/pdf/technotes/ddr3/TN41_01DDR3%20Power.pdf. 9

[9] Hongzhong Zheng, Jiang Lin, Zhao Zhang, Eugene Gorbatov, Howard David, and Zhichun Zhu. Mini-rank: Adaptive dram architecture for improving memory power efficiency. In *Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 210–221, 2008. 10

[10] The Micron System Power Calculator. http://www.micron.com/support/dram/power_calc.html. 10, 22, 28, 36

[11] Understanding DRAM performance specifications, IBM Applications Note. http://info.ee.surrey.ac.uk/Personal/R.Webb/l3a15/extras/dramperf.pdf. 11

[12] DDR3 SDRAM part catalog. http://www.micron.com/partscatalog.html?categoryPath=products/parametric/dram/ddr3_sdram. 14, 22

[13] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers. *Micro, IEEE*, 29(1):22 –32, 2009. 16, 17, 47

[14] Andy Glew. MLP yes! ILP no! In *Wild and Crazy Idea Session, ASPLOS*, 1998. 17, 30

[15] Hongzhong Zheng, Jiang Lin, Zhao Zhang, and Zhichun Zhu. Decoupled dimm: Building high-bandwidth memory system using low-speed dram devices. In *Int'l Symp. on Computer Arch.*, June 2009. 23

[16] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th International Symposium on Computer Architecture*, 2009. 23

[17] Low-power versus standard DDR SDRAM. http://download.micron.com/pdf/technotes/DDR/tn4615.pdf. 24, 27

[18] External Memory Interface Handbook Volume 3. http://www.altera.com/literature/hb/external-memory/emi_ddr3up_ug.pdf. 24

[19] Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Efficient runahead execution: Power-efficient memory latency tolerance. *IEEE Micro*, 26(1):10–20, 2006. 30, 32, 35

[20] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006. 36

[21] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: A memory-system simulator. *SIGARCH Computer Architecture News*, 33(4):100–107, 2005. 36

[22] SPEC CPU2006. http://www.spec.org/cpu2006. 37

[23] Arun Nair and Lizy John. Simulation points for spec cpu 2006. In *Proceedings of the 26th International Conference on Computer Design, ICCD*, pages 397–403, 2008. 37

[24] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23:84–93, November 2003. 42, 57

[25] Scott Rixner. Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 355–366. IEEE Computer Society, 2004. 42, 47

[26] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM. 42

[27] Reetuparna Das, Onur Mutlu, Thomas Moscibroda, and Chita R. Das. Aérgia: Exploiting packet latency slack in on-chip networks. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 106–116, New York, NY, USA, 2010. ACM. 49

[28] Jae H. Kim and Andrew A. Chien. Rotating combined queueing (RCQ): bandwidth and latency guarantees in low-cost, high-performance networks. In *Proceedings of the 23rd annual international symposium on Computer architecture*, ISCA '96, pages 226–236. ACM, 1996. 49

[29] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium proceedings on Communications architectures & protocols*, SIGCOMM '89, pages 1–12, 1989. 49

[30] NVIDIA® CUDA™ architecture, 2009. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf. 59

[31] AMD Fusion™ Family of APUs: Enabling a Superior, Immersive PC Experience. http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf. 59

[32] Paul S. A. Reitsma and Nancy S. Pollard. Perceptual metrics for character animation: sensitivity to errors in ballistic motion. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 537–542, 2003. 60

[33] Jeremy W. Sheaffer, David P. Luebke, and Kevin Skadron. The visual vulnerability spectrum: characterizing architectural vulnerability for graphics hardware. In *Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 9–16, 2006. 60

[34] StreamIt Language Specification, Version 2.1, 2006. http://groups.csail.mit.edu/cag/streamit/shtml/documentation.shtml. 62