The Dissertation Committee for Ibrahim Hur
certifies that this is the approved version of the following dissertation:

Enhancing Memory Controllers to Improve DRAM Power and Performance

Committee:

Calvin Lin, Supervisor

Kathryn S. McKinley

Margarida F. Jacome

Gustavo de Veciana

Dewayne E. Perry

Enhancing Memory Controllers to Improve DRAM Power and Performance

by

Ibrahim Hur, B.S.; M.Sc.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

To Ece

Acknowledgments

This work would not have been possible without the relentless support and encouragement of my advisor Dr. Calvin Lin. I would like to thank him for his wisdom, advice, patience, and invaluable guidance during my doctoral studies.

I would also like to thank members of my dissertation committee, Dr. Kathryn S. McKinley, Dr. Margarida F. Jacome, Dr. Gustavo de Veciana, and Dr. Dewayne E. Perry. I especially thank Dr. McKinley for taking time and effort to help me improve this dissertation.

Many thanks to David W. Matula, Harvey G. Cragon, Earl Swartzlander, and Turhan Tunali who inspired me to do research in computer architecture. Thanks to my friends Alper Buyuktosunoglu, Daniel A. Jimenez, Men-Chow Chiang, and Brian O'Krafka for their help in my research. Thanks to Alison N. Norman, Maria Jump, and all members of the Speedway group for their feedback on my practice talks. I also thank Murat M. Tanik, Mehmet M. Kayaalp, and Cengiz Erbas for their help during my first years in the graduate school.

I would like to thank the faculty and staff of The University of Texas at Austin. I especially thank Melanie Gulick and Gem Naivar for their help in every administrative issue. I also thank International Business Machines Corporation for giving me resources, financial support, and flexibility during my graduate studies.

I am very fortunate to have wonderful parents and a sister who have always believed in me. I thank my father Hamza, my mother Mufide, and my sister Safiye for their constant encouragement. I am grateful to Remziye Sener Deveci, Tekin Sayilar, and Neset Sayilar for their influence on me for doing academic research. I would also like to thank my grandparents for their belief in the importance of education.

Finally, many thanks go to my best friend Ece. I am truly grateful to her for her unconditional support over many years. Without her encouragement during every day of my graduate studies, I would not be able finish this dissertation.

IBRAHIM HUR

The University of Texas at Austin
December 2006

Enhancing Memory Controllers to Improve DRAM Power and Performance

Publication No. ____________________

Ibrahim Hur, Ph.D.
The University of Texas at Austin, 2006

Supervisor: Calvin Lin

Technological advances and new architectural techniques have enabled processor performance to double almost every two years. However, these performance improvements have not resulted in comparable speedups for all applications, because the memory system performance has not kept pace with processor performance in modern systems. In this dissertation, by concentrating on the interface between the processors and memory, the memory controller, we propose novel solutions to all three aspects of the memory problem, that is bandwidth, latency, and power.

To increase available bandwidth between the memory controller and DRAM, we introduce a new scheduling approach. To hide memory latency, we introduce a new hardware prefetching technique that is useful for applications with regular or irregular memory accesses. And finally, we show how memory controllers can be used to improve DRAM power consumption.

We evaluate our techniques in the context of the memory controller of a highly tuned modern processor, the IBM Power5+. Our evaluation for both technical and commercial benchmarks in single-threaded and simultaneous multi-threaded environments show that our techniques for bandwidth increase, latency hiding, and power reduction achieve significant improvements. For example, for single-threaded applications, when our scheduling approach and prefetching method are implemented together, they improve the performance of the SPEC2006fp, NAS, and a set of commercial benchmarks by 14.3%, 13.7%, and 11.2%, respectively.

In addition to providing substantial performance and power improvements, our techniques are superior to the previously proposed methods in terms of cost as well. For example, a version of our scheduling approach has been implemented in the Power5+, and it has increased the transistor count of the chip by only 0.02%.

This dissertation shows that without increasing the complexity of neither the processor nor the memory organization, all three aspects of memory systems can be significantly improved with low-cost enhancements to the memory controller.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past few decades, advances in silicon process technology have significantly reduced the size and switching times of transistors. As a result, both the number of transistors on a single die and clock rates of processors have increased rapidly, enabling processor performance to double almost every two years. However, these performance improvements have not resulted in comparable speedups for all applications. For instance, increasing processor performance by 50% of an IBM Power5+ system improves the performance of the SPEC2006 benchmarks by only 13.1%. Overall performance does not scale at comparable rates in all applications because the memory system performance has not kept pace with processor performance in modern systems.

There are two aspects of the memory system performance: *latency* and *bandwidth*. Today, latencies have already reached hundreds of processor cycles, because memory access delays do not decrease as fast as processor speeds increase. Moreover, memory latencies are expected to become even longer in the foreseeable future because memory developers are required to create a balance between the speed and capacity of memory chips, rather than focusing solely on speed. In order to tolerate growing latencies, modern systems increasingly use techniques, such as data prefetching and simultaneous multithreading, which often elevate memory bandwidth demands. In addition to latency tolerating techniques, technology trends, such as faster processor clock rates and chip multi-processors, increase bandwidth requirements in modern systems. Hence, memory bandwidth, once a concern for only streaming scientific codes, has become crucial for non-streaming applications as well.

While long latency and insufficient bandwidth limit the performance of modern systems, another performance criteria has recently emerged: *power*. Power is not an issue just for processors, but it is a first order concern for DRAM as well. For example, in systems with large memory capacities, DRAM's are reported to consume up to 45% of a system's total power [42]. Limited power budgets force designers to trade off performance for power. Therefore, power savings in DRAM will reduce overall power consumption and may improve system performance and energy usage.

1.1 Our Solution

Previous proposals for improving latency, bandwidth, or power aspects of memory systems have significantly increased the complexity of processors and/or main memory organizations. For example, prefetching approaches for hiding latency require large chip area to be effective for irregular memory accesses; bandwidth improving methods, such as multiple banks and multiple channels between processors and memory, create a challenge for the processors to schedule memory commands intelligently; and mechanisms for reducing DRAM power consumption require complex algorithms to reduce performance degradations.

Although processor and memory systems have been explored extensively, the interface between them, the *memory controller*, had received relatively less attention. The memory controller, either off-chip or integrated with the chip, controls the flow of data to and from the memory, buffers data if necessary, and performs optimizations to improve performance. As processors and memory systems become increasingly complex, it makes sense to explore ways that the memory controller can be made more sophisticated. Therefore, we concentrate on the interface between the processor and memory, and we propose a low cost memory controller design that improves all three aspects of memory systems:

- To hide latency, we propose a new prefetching approach that is useful for applications with regular or irregular memory accesses.
- To improve bandwidth, we introduce a memory command reordering technique that reduces contention in the memory system.
- To address DRAM power consumption, we augment our command reordering approach to include power optimizations, and we present a new model-based throttling technique.
- To put it all together, we present and evaluate a memory controller design that includes all of our enhancements for latency, bandwidth, and power.

1.2 Thesis Statement

All three aspects of memory systems, that is latency, bandwidth, and power consumption, can be significantly improved with small modifications to the memory controller.

1.3 Contributions

In this dissertation, we make the following contributions:

- To deal with increasing memory latencies, we introduce a probabilistic hardware prefetching technique that is particularly useful for applications with low spatial locality. This technique keeps track of the frequency of stream sizes in an application and uses that information to make prefetching decisions. We implement this low cost method as a memory-side prefetcher, and we show that it complements an existing processor-side prefetcher. To better assign resources to prefetch and regular commands, we also introduce an adaptive approach that modulates the relative priority of prefetch commands and regular commands by monitoring the status of the memory system.

- To satisfy growing memory bandwidth demands, we present a new memory scheduling approach. To reduce contention in the memory system, this scheduling technique chooses commands to issue to memory by considering physical characteristics of main memory and the history of memory commands. In addition, to reduce bottlenecks in the memory controller itself, this technique matches the sequence of memory commands to a predetermined command pattern. To make this method work for more than one command pattern, we introduce an adaptive method that dynamically selects from among multiple schedulers.

- To address the power issue, we provide an algorithm to manage powerdown capabilities of DRAM chips; we design a memory scheduler that optimizes for both performance and power; and we develop an approach to throttle memory traffic, with minimal performance degradation, so that DRAM power consumption will meet some specified budget.

- We evaluate our techniques in the context of the memory controller of a highly tuned modern processor, the IBM Power5+. Our evaluation covers both technical and commercial benchmarks in single-threaded and simultaneous multi-threaded environments. We show that our techniques for latency hiding, bandwidth increase, and power reduction, achieve substantial improvements. For example, our prefetching approach improves the performance of our technical and commercial benchmarks by an average of 10.2% and 8.4%, respectively. Similarly, on the same benchmarks, our scheduling method increases performance by 9.7% and 7.5%. When we combine our latency hiding and scheduling methods, we achieve 14.3% and 11.2% performance improvement.

1.4 Organization

This dissertation is organized as follows. The next chapter presents background and our experimental methodology. In the following three chapters, we present our new solutions and their empirical evaluation: in Chapter 3, the Adaptive History-Based Schedulers to improve available bandwidth; in Chapter 4, Adaptive Stream Detection for latency hiding; and in Chapter 5, DRAM Power Optimizations. In Chapter 6, we place our work in the context of prior work; and finally in Chapter 7, we conclude and discuss future work.

Chapter 2

Background and Methodology

We evaluate our bandwidth, latency, and power improvement techniques using simulation of a modern architecture, the IBM Power5+. In this chapter, we first present an overview of the Power5+ architecture. We then describe our simulation methodology. Finally, we discuss the details of the benchmarks that we use to evaluate our approaches.

2.1 A Modern Architecture: The IBM Power5+

The IBM Power5+ [10, 35] is the successor to the Power5 and is the latest member of the Power4 [69] line of processors. The Power5+ chip has about 300 million transistors and is designed to address both scientific and commercial workloads. Some improvements in the Power5 and Power5+ over the previous generation Power4 include a larger L2 cache, simultaneous multithreading, power-saving features, and an on-chip memory controller.

As shown in Figure 2.1, the Power5+ has two processors per chip, where each processor has split first-level data and instruction caches. Each chip has a unified second-level cache shared by the two processors, and it is possible to attach an optional L3 cache. Four Power5+ chips can be packaged together to form an 8-way SMP, and up to eight such SMP's can be combined to create 64-way SMP scalability.

The Power5+ [35] has an aggressive processor-side prefetching unit [69] that prefetches from memory to L2 and from L2 to L1. The prefetcher implements a sequential prefetching policy that waits to issue prefetches until it detects two consecutive cache misses. There are 12 entries in the stream detection unit, and eight streams can be prefetched concurrently. When the steady state is reached, each stream brings one additional line into the L1 cache, and one additional line into the L2 cache.

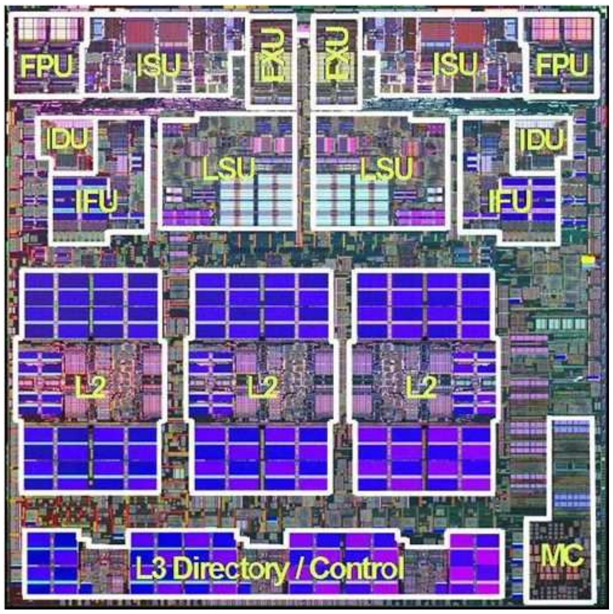

Figure 2.1: The IBM Power5+ chip.

The Power5+'s memory controller, as shown in Figure 2.1, is shared by two processors. The memory controller has two reorder queues: a Read Reorder Queue and a Write Reorder Queue. Each of these queues can hold 8 memory references, where each memory reference is an entire L2 cache line or a portion of an L3 cache line. An arbiter selects an appropriate command from these queues to place in the Central Arbiter Queue (CAQ), where they are sent to memory in FIFO order. The memory controller can keep track of the 12 previous commands that were passed from the CAQ to the DRAM.

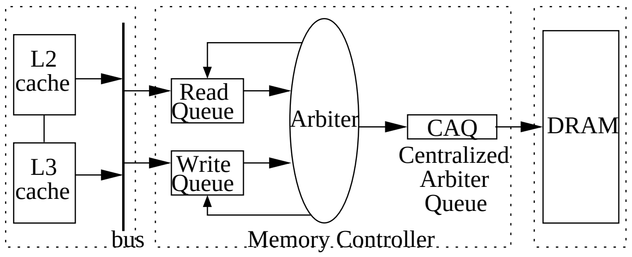

Figure 2.2: The Power5+ memory controller.

The Power5+ does not allow dependent memory operations to enter the memory controller at the same time, so the arbiter is allowed to reorder memory operations arbitrarily. Furthermore, the Power5+ gives priority to demand misses over prefetches, so from the memory controller's point of view, all commands in the reorder queues are equally important. Both of these features greatly simplify the task of the memory scheduler.

2.1.1 DRAM Organization and Power Consumption

The Power5+ systems that we consider use DDR2 SDRAM chips, which are essentially a 5D structure. Two *ports* connect the memory controller to the DRAM. The DRAM is organized as 4 *ranks*, where each rank is an organizational unit consisting of 4 *banks*. Each bank in turn is organized as a set of rows and columns. This structure imposes many performance constraints. For example, port conflicts, rank conflicts, and bank conflicts each incur their own delay, and the costs of these delays depends on whether the operations are Reads or Writes. In this system, bank con-

State	Average Power (normalized)
Read transfer (1 bank)	1.000
Read transfer (4 banks, staggered)	1.875
Activate-Precharge (1 bank)	0.594
Idle (precharge quiet)	0.281
Power-down (precharge)	0.038

Table 2.1: Power consumption for various states of the Micron 512MB DDR2.

flict delays are an order of magnitude greater than the delays introduced by rank or port conflicts.

With multiple ranks in a system, it is possible that at any given time some of the ranks are idle. While DRAM power consumption is lower when a rank is idle, the low-power mode can reduce power consumption by another order of magnitude. Table 2.1 shows the relative power consumption for some prominent modes for the ranks of a Micron 512MB DDR2-533MHz SDRAM chip. A rank can enter low-power mode, with a command from the memory controller, only if no bank of the rank is processing a memory command. There is an exit latency, which is 12 processor cycles for the memory chips that we simulate, for transitioning from the low-power mode to other modes. Additionally, other timing constraints place restrictions on how soon the low-power mode can be entered. Our simulation environment [59] accurately models all timing constraints, modes, and activities of the ranks and banks; it uses the corresponding power consumption information from the DRAM datasheets [22] to correctly model power and performance of the DRAM chips.

2.1.2 Architectural Parameters

In Table 2.2 we present the base parameters for the IBM Power5+ systems that we simulate in our studies. These parameters represent one of the most modern system configurations with the Power5+.

Parameter	Value
L1D, L1I	64KB, 2way, 128B
L2	1.9MB, 10way, 128B, shared
L3	36MB, 16way, 128B, shared, victim
Frequency	2.132 GHz
Memory Address Bus	8B
Memory Read Data Bus	16B
Memory Write Data Bus	8B
Read Reorder Queue	8
Write Reorder Queue	8
Centralized Arbiter Queue	3
DRAM Type	DDR2
DRAM Speed	533 MHz
Number of Ranks	4
Banks in a Rank	4
Active Commands in DRAM	12

Table 2.2: Base parameters for the IBM Power5+.

2.2 Simulation Methodology

The simulators that we use are for actual commercial products, namely the IBM Power4, Power5, and Power5+ systems. They are developed by the processor design and modeling teams of IBM. The simulators represent the modeled systems in extensive detail. Their development, validation, and verification took many years of manpower. For example, one of the simulators consists of about 1.5 million lines of VHDL code and is cycle accurate. With our set of simulators, we can simulate details of both the processor and memory system. We are also able to perform multithreaded simulations as well as multiple processor simulations.

The simulation environment that we use consists of three main parts: a simulator for the processor, a simulator for the level two and level three caches, and a simulator for the main memory. The simulators for caches and main memory use the event-driven CSIM [63] framework.

2.2.1 Processor, Nest, and Main Memory Simulators

Our processor simulator, ProSim, is a trace driven simulator for a single processor of the Power4, Power5, or Power5+ system. The processor includes execution units, control logic, pipeline structure, and the first level data and instruction caches. ProSim reads a single record from an instruction trace and processes it through the processor units. This simulator is designed with the purpose of evaluating various design options. Therefore, we are able to change many architectural parameters before simulation. Cache size, associativity, number of floating point units, and branch history table size are a few examples of these configurable paramaters.

ProSim delays the processing of an instruction if that instruction causes a miss in a first level cache. NestSim, the second part of the simulation environment, handles the processing of these missed instructions. As soon as a load or store instruction misses in a first level cache, a new thread is generated. This thread flows through the second and third level caches and returns the result to ProSim to wake the sleeping ProSim thread. NestSim simulates the details of the second and third level caches in detail, but it stops processing the level-1 cache miss when there is a need for a main memory access.

The third simulator, MemSim, is a DRAM simulator that jointly models power and performance of the main memory subsystem. It is also a highly configurable simulator, originally designed for modeling the main memory system of high-end servers, with support for different memory interleaving, page modes, and power management policies. We extend MemSim to act as a module in our simulation environment along with cycle-by-cycle tracking of activities in the memory system. In this mode, MemSim models all the memory system activity while synchronizing with the NestSim simulator at every processor cycle.

We integrate NestSim with the MemSim memory simulator by replacing NestSim's fixed-latency memory model with MemSim. The integrated simulator generates timing information for both processor and memory subsystems. In addition, MemSim provides detailed power and energy information for DRAM.

2.2.2 Verification of the Simulators

We verify our simulators against an RTL simulator (VSim). VSim consists of about 1.5 million lines of VHDL code that has been developed by the IBM designers for the Power4, Power5, and Power5+ systems. Even though VSim represents the actual system correctly, it cannot be used in our studies because it is extremely slow and difficult to modify. VSim has been intensively validated and verified for functionality and performance. Verification of VSim itself is beyond the scope of our study.

We have performed performance verification and simulator development concurrently. Whenever a discrepancy is detected between VSim and our simulators, we modify our simulators and perform the comparisons again. The development of VSim and our simulators is also concurrent. In other words, as the designers add new details, VSim changes, which further complicates our simulator development process.

We create a verification environment where we can run the same test cases with VSim and with our combined simulators. To test various sections of the hardware, there are several hundred basic test cases with one or a few instructions. We also have longer test cases to test memory bandwidth.

In general, the error between our simulators and the VSim is within 1%. The verification process involves not only the comparison of the absolute execution times, but it also compares the of timing of various events. For example, for an instruction that needs main memory access, it is important to match memory queue entry and exit times in addition to overall memory latency. For most test cases, we perform these comparisons manually.

2.2.3 Simulation Approaches

There are two modes of running simulations. In the first mode (trace-based), instructions are fed to the simulator from a trace file. Instructions are processed through all levels of the simulaton environment, i.e. ProSim, NestSim, and MemSim. In the second mode (stream-based), only NestSim and MemSim are used. We use this mode to study test cases with heavy main memory access requirements. A stream generator creates various number of data streams (Reads and/or Writes) and feeds those to NestSim. Multiprocessor simulations can use only this mode. For a set of microbenchmarks, we compared the results of trace-based and stream-based approaches, and we found that average performance difference between these approaches is 1.3%.

Our simulation environment allows us to perform uniprocessor or multiprocessor runs. We can simulate any test case with uniprocessor configurations, but multiprocessor simulations have limitations. If the configuration is for a uniprocessor, we can also specify the number threads to run. Each thread can use different trace files.

2.3 Benchmarks and Microbenchmarks

We evaluate our bandwidth, latency, and power improvement techniques using both technical and non-technical benchmarks. For technical benchmarks, we use the Stream [48], NAS [3], and recently released SPEC2006fp benchmarks [68]. For non-technical workloads, we use IBM internal benchmarks for commercial applications. We also create a set of microbenchmarks for detailed analysis of the memory controller.

The first set of benchmarks measures streaming behavior. The Stream benchmarks, which others have used to measure the sustainable memory bandwidth of

Kernel	Description
daxpy	x[i]=x[i]+a*y[i]
copy	x[i]=y[i]
scale	x[i]=a*x[i]
vsum	x[i]=y[i]+z[i]
triad	x[i]=y[i]+a*z[i]
fill	x[i]=a
sum	sum=sum+x[i]

Table 2.3: The extended set of Stream Benchmarks.

systems [12, 64, 8, 72], consist of four simple vector kernels: *copy*, *scale*, *vsum*, and *triad*. The Stream2 benchmarks, which consist of *fill*, *copy*, *daxpy*, and *sum*, were introduced to measure the effects of all levels of caches and to show the performance differences of reads and writes. In our study, we combine the Stream and the Stream2 to create the extended Stream benchmarks that consist of seven vector kernels. We list these kernels in Table 2.3 and, for simplicity, we refer to them collectively as the Stream benchmarks in the rest of this dissertation.

The second set of workloads, the NAS (Numerical Aerodynamic Simulation) benchmarks, is a group of eight programs developed by NASA (see Table 2.4). These programs are derived from computational fluid dynamics applications and are good representatives of scientific applications. The NAS benchmarks are fairly memory intensive, but they are also good in measuring various other performance characteristics of high performance computing systems. There exists parallel and serial implementations of the various sizes of the NAS benchmarks. In our studies, we use serialized versions of class B.

The third set of technical workloads that we use are the SPEC2006fp benchmarks [68]. As depicted in Table 2.5, this benchmark suite consists of 17 scientific applications. SPEC benchmarks are considered the industry standard in evaluating performance of computer systems. This benchmark suite has both integer and floating point benchmark sets. We do not evaluate integer benchmarks because with

Program	Description
bt	Block-Tridiagonal Systems
cg	Conjugate Gradient
ep	Embarrassingly Parallel
ft	Fast Fourier Transform for Laplace Equation
is	Integer Sort
lu	Lower-Upper Symmetric Gauss-Seidel
mg	Multi-Grid Method for Poisson Equation
sp	Scalar Pentadiagonal Systems

Table 2.4: The NAS Benchmarks.

large caches of the Power5+, memory pressure of these benchmarks are low.

For the non-technical workloads, we use five commercial server applications, namely *tpcc*, *trade2*, *cpw2*, *sap*, and *notesbench*. Tpcc is an online transaction processing workload; cpw2 is a Commercial Processing Workload that simulates the database server of an online transaction processing environment; trade2 is an end-to-end web application that models an online brokerage; sap is a database workload; and notesbench is a tool that evaluates the performance of a set of systems which are running Lotus Notes.

Finally, we use a set of 14 microbenchmarks, which allows us to explore a wider range of memory controller configurations, and which allows us to explore in detail the behavior of our memory controllers. Each of our microbenchmarks uses a different Read/Write ratio, and each is named $xRyW$, indicating that it has x Read streams and y Write streams. These microbenchmarks represent most of the data streaming patterns that we expect to see in real applications. There are two other reasons that we use microbenchmarks. First, the simulation times for these benchmarks are very short, e.g. in the order of minutes. We need short simulation times to investigate a large number of design configurations. Second, our simulation environment has a limitation to perform multiple processor simulations only with this type of microbenchmarks.

Program	Application Area
bwaves	Fluid dynamics
gamess	Quantum chemistry
milc	Physics/Quantum chromodynamics
zeusmp	Physics
gromacs	Biochemistry/Molecular dynamics
cactusADM	Physics/General relativity
leslie3d	Fluid dynamics
namd	Biology/Molecular dynamics
dealll	Finite element analysis
soplex	Linear programming, optimization
povray	Image ray-tracing
calculix	Structural mechanics
GemsFDTD	Computational electromagnetics
tonto	Quantum chemistry
lbm	Fluid dynamics
wrf	Weather modeling
sphinx3	Speech recognition

Table 2.5: The SPEC2006fp Benchmarks.

2.3.1 Test Case Generation

For the Stream, NAS, and SPEC2006fp benchmarks, we create traces using an internal IBM tool. This tool generates, from an executable, as many instructions as we specify. The output can be a certain contiguous section of the instruction stream or the concatenation of uniformly sampled pieces. For the Stream benchmarks we use contiguous traces. However, the NAS and SPEC benchmarks are prohibitively long for a single trace file. For example, if not sampled, some SPEC programs runs for about 3 trillion instructions, which would require about 70 years of simulation time in our detailed simulators. Therefore, for the NAS and SPEC2006fp workloads we generate sampled traces. We first generate 50 uniformly distributed pieces, each having 2 million instructions, and then we combine those pieces to create a single trace of 100 million instructions. To evaluate the representativeness of the sampled traces, we compare the CPI's of the entire programs on an actual Power5+ to the simulator output of the traces. As we show in Figure 2.3 and Figure 2.4, our sampling approach creates a good match to the original CPI of the benchmarks.

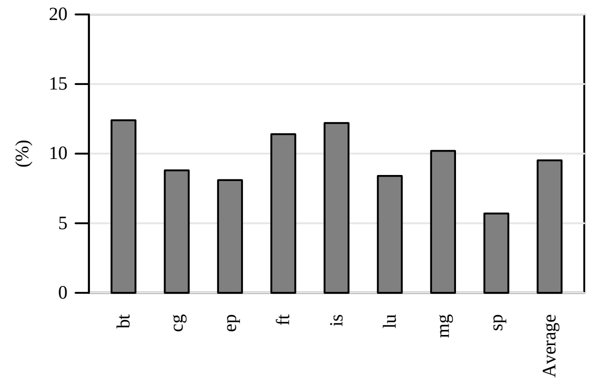

Figure 2.3: Percent error, in CPI, introduced by trace sampling, for the NAS benchmarks.

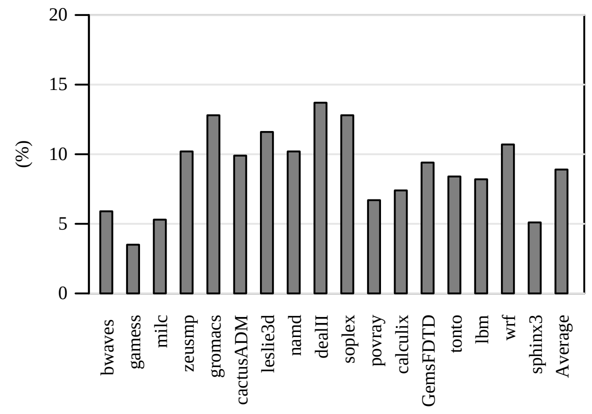

Figure 2.4: Percent error, in CPI, introduced by trace sampling, for the SPEC2006fp benchmarks.

For the commercial workloads, we use traces collected by special hardware. Finally, to generate microbenchmarks, we use a stream generator. This tool runs concurrently with the simulator and, as input, it takes the number of Read or Write streams, the length of each stream, and the offset among the streams. The offset among the streams affects the order of the commands going to memory, which may change the number of the bank or rank conflicts.

Chapter 3

Improving Memory Bandwidth with Smart Scheduling

Memory bandwidth is an increasingly important aspect of overall system performance. Early work for improving available bandwidth focused on streaming workloads, which place the most stress on the memory system. Early work also focused on avoiding bank conflicts, since bank conflicts typically lead to long stalls in the DRAM. In particular, numerous hardware and software schemes have been proposed for interleaving memory addresses [11], skewing array addresses [21, 13], and otherwise [7, 49, 50, 51, 52] attempting to spread a stream of regular memory accesses across the various banks of DRAM. Valero et al. [71, 57] describe a method of dynamically reordering memory commands so that the banks are accessed in a strict round-robin fashion. More recently, Rixner et al. [61] evaluate a set of simple heuristics for reordering memory commands, some of which consider additional DRAM structure, such as the rows and columns that make up banks. Rixner et al. do not identify a conclusive winner among their various heuristics, but they do find that simply avoiding bank conflicts performs as well as any of their other heuristics.

Recently, the need for increased memory bandwidth has begun to extend beyond streaming workloads. Faster processor clock rates and chip multi-processors increase the demand for memory bandwidth. Furthermore, to cope with relatively slower memory latencies, modern systems increasingly use techniques that reduce or hide memory latency at the cost of increased memory bandwidth demands. For example, simultaneous multi-threading hides latency by using multiple threads, and hardware-controlled prefetching speculatively brings in data from higher levels of the memory hierarchy so that it is closer to the processor. To accommodate more parallelism, modern DRAM's are also increasing in complexity. For example, the DDR2-533 SDRAM chips have a 5D structure and a wide variety of costs associated with access to the various sub-structures.

In the face of these technological trends, previous solutions are limited in two ways. First, it is no longer sufficient to focus exclusively on streams as a special case; we instead need to accommodate richer patterns of data access. Second, it is no longer sufficient to focus exclusively on avoiding bank conflicts; scheduling decisions instead need to consider other physical sub-structures of increasingly complex DRAM's.

Previous work is also limited in its avoidance of bottlenecks within the memory controller itself. To understand this problem, consider the execution of the daxpy kernel on the IBM Power5+'s memory controller. The daxpy kernel performs two reads for every write. If the scheduler does not schedule memory operations in the ratio of two reads per write, either the Read queue or the Write queue will become saturated under heavy traffic, creating a bottleneck. To avoid such bottlenecks, the scheduler should select memory operations so that the ratio of reads and writes matches that of the application.

In this chapter, we describe a new approach—adaptive history-based (AHB) memory scheduling—that addresses all three limitations by maintaining information about the state of the DRAM along with a short history of previously scheduled operations. Our solution avoids bank conflicts by simply holding in the reorder queue any command that will incur a bank conflict; history information is then used to schedule any command that does not have a bank conflict. Our approach provides three conceptual advantages: (1) it allows the scheduler to better reason about the delays associated with its scheduling decisions, (2) it is applicable to complex DRAM structures, and (3) it allows the scheduler to select operations so that they match the program's mixture of Reads and Writes, thereby avoiding certain bottlenecks within the memory controller.

A version of the AHB scheduler that uses one bit of history and that is tailored for a fixed Read-Write ratio of 2:1 has been implemented in the recently shipped IBM Power5+. Nevertheless, important questions about the AHB scheduler still exist. Perhaps the most important question is whether our solution will become more or less important to future systems, which we can study by altering various architectural parameters of the processor, the memory system, and the memory controller. For example, is the AHB scheduler effective for multi-threaded and multi-core systems? Is the AHB scheduler needed for DRAM's that will have many more banks and thus much more parallelism? If we increase the size of the memory controller's internal queues, would a simpler solution suffice? Finally, can the solution be improved by incorporating more sophisticated methods of avoiding bank conflicts? In this chapter, we answer these questions and others to demonstrate the flexibility and robustness of our solution, evaluating it in a variety of situations. In particular, this chapter makes the following contributions:

- We present the notion of adaptive history-based schedulers, and we provide algorithms for designing such schedulers.
- While most previous memory scheduling work pertains to cacheless streaming processors, we show that the same need to schedule memory operations applies to general purpose processors. In particular, we evaluate our solution in the

context of the IBM Power5+, which has a 5D structure (port, rank, bank, row, column), plus caches.

- We evaluate our solution using a cycle-accurate simulator for the Power5+. When compared with an in-order scheduler, our solution improves IPC on the NAS [3] benchmarks by a geometric mean of 16.8%, and it improves IPC on the Stream benchmarks [48] by 45.5%. When compared against one of Rixner et al.'s solution, our method sees improvements of 5.8% for the NAS benchmarks and 11.3% for the Stream benchmarks. In addition to NAS and Stream, we also evaluate our approach on commercial benchmarks, where we see 32.8% and 5.6% performance improvements compared to in-order and Rixner's approach, respectively.

- We show that multi-threaded workloads increase the performance benefit of our solution. This result may be surprising because multi-threading would seem to defeat our technique's ability to match the workload's mixture of Reads and Writes. However, we find that the increased memory system pressure increases the benefit of smart scheduling decisions. For example, when compared with the state of the art on a two processor system each running two threads, our approach improves performance of commercial benchmarks, compared to Rixner's approach, between 6% and 10%. We find the somewhat surprising result that for previous memory schedulers, the use of SMT processors can actually decrease performance because the DRAM becomes a bottleneck.

- We provide insights to explain why our solution improves the bandwidth of the Power5+'s memory system.

- We tune our solution and evaluate its sensitivity to various internal parameters. For example, we find that the criterion of minimizing expected latency is more important than of matching the expected ratio of Reads and Writes.

- We show that our solution tends to be more valuable in future systems. In addition to the multi-threading results, we show that our solution performs well as we alter various memory controller parameters, DRAM parameters, and system parameters.
- We explore the effects of varying parameters of the memory scheduler itself. We find that our AHB scheduler provides significant benefits in performance and hardware costs when compared with other approaches. In many cases, our technique is superior to other approaches even when ours is given a fraction of the resources.
- We show that the hardware cost of our approach is minimal.

This chapter is organized as follows. The next section presents our solution, followed by experimental evaluation and sensitivity analysis, then we discuss implementation cost of our approach and we provide concluding remarks.

3.1 Adaptive History-Based Memory Schedulers

This section describes our new approach to memory controller design, which focuses on making the scheduler both history-based and adaptive. A history-based scheduler uses the history of recently scheduled memory commands when selecting the next memory command. In particular, a finite state machine encodes a given scheduling goal, where one goal might be to minimize the latency of the scheduled command and another might be to match some desired balance of Reads and Writes. Because both goals are important, we probabilistically combine two FSM's to produce an scheduler that encodes both goals. The result is a history-based scheduler that is optimized for one particular command pattern. To overcome this limitation, we introduce adaptivity by using multiple history-based schedulers; our adaptive scheduler observes the recent command pattern and periodically chooses the most appropriate history-based scheduler.

3.1.1 History-Based Schedulers

In this section we describe the basic structure of history-based schedulers. Similar to branch predictors, which use the history of the previous branches to make predictions [11], history-based schedulers use the history of previous memory commands to decide what command to send to memory next. These schedulers can be implemented as an FSM, where each state represents a possible history string. For example, to maintain a history of length two, where the only information maintained is whether an operation is a Read or a Write, there are four possible history strings—*ReadRead*, *ReadWrite*, *WriteRead*, and *WriteWrite*—leading to four possible states of the FSM. Here, a history string xy means that the last command transmitted to memory was y and the one before that was x.

Unlike branch predictors, which make decisions based purely on branch history, history-based schedulers make decisions based on both the command history and the set of available commands from the reorder queues. The goal of the scheduler is to encode some optimization criteria to choose, for a given command history, the next command from the set of available commands. In particular, each state of the FSM encodes the history of recent commands, and the FSM checks for possible next commands in some particular order, effectively prioritizing the desired next command. When the scheduler selects a new command, it changes state to represent the new history string. If the reorder queues are empty, there is no state change in the FSM.

As an illustrative example, we present an FSM for an scheduler which uses a history length of three. Assume that each command is either a Read or a Write operation to either port number 0 or 1. Therefore, there are four possible commands, namely Read Port 0 (R0), Read Port 1 (R1), Write to Port 0 (W0), and Write to Port 1 (W1). The number of states in the FSM depends on the history length and the type of the commands. In this example, since the scheduler keeps the history of the last three commands and there are four possible command types, the total number of states in the FSM is 4×4×4=64. In Figure 3.1 we show an example of transitions from one particular state in this sample FSM. In this hypothetical example, we see that the FSM will first see if a W1 is available, and if so, it will schedule that event and transition into a new state. If this type of command is not available, the FSM will look for an R0 command as the second choice, and so on.

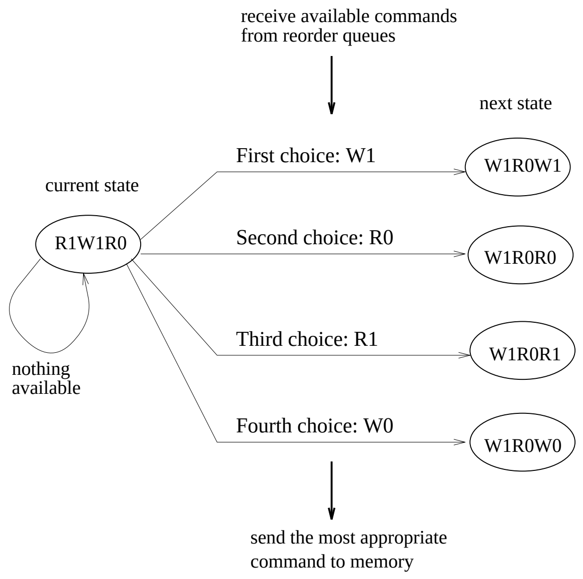

Figure 3.1: Transition diagram for the current state $R1W1R0$. Each available command type has different selection priority.

3.1.2 Design Details of History-Based Schedulers

As mentioned earlier, we have identified two optimization criteria for prioritization: the *amount of deviation* from the command pattern and the *expected latency* of the scheduled command. The first criterion allows an scheduler to schedule commands to match some expected mixture of Reads and Writes. mixture of Reads and Writes. The second criterion represents the mandatory delay between the new memory command and the commands already being processed in the memory. We first present algorithms for generating schedulers for each of the two prioritization goals in isolation. We then provide a simple algorithm for probabilistically combining two schedulers.

Optimizing for the Command Pattern

Algorithm 1 generates state transitions for an scheduler that schedules commands to match a ratio of x Reads and y Writes in the steady state. The algorithm starts by computing, for each state in the FSM, the Read/Write ratio of the state's command history. For each state, the algorithm then computes the Read/Write ratio of each possible next command. Finally, the next commands are sorted according to their Read/Write ratios. For example, consider an scheduler with the desired pattern of "one Read per Write", and assume that the current state of the FSM is $W1R1R0$. The first choice in this state should either be a $W0$ or $W1$, because only those two commands will move the Read/Write ratio closer to 1.

In situations where multiple available commands have the same effect on the deviation from the Read/Write ratio of the scheduler, the algorithm uses some secondary criterion, such as the expected latency, to make final decisions.

Optimizing for the Expected Latency

To develop a scheduler that minimizes the expected delay of its scheduled operations, we first need a cost model for the mandatory delays between various memory operations. Our goal is to compute the delay caused by sending a particular command, c_{new}, to memory. This delay is necessary because of the constraints between

Algorithm 1 command_pattern_scheduler(n)

```
// $n$ is the history length
1: for all command sequences of size $n$ do
2:    $r\_old$:=Read/Write ratio of the command sequence.
3:
4:    for each possible next command do
5:       $r\_new$:=Read/Write ratio.
6:    end for
7:    if $r\_old <$ ratio of the scheduler, $x/y$ then
8:       Read commands have higher priority.
9:    else
10:      Write commands have higher priority.
11:   end if
12:   if there are commands with equal $r\_new$ then
13:      Sort them with respect to expected latency.
14:      Pick the command with the minimum delay.
15:   end if
16:
17:   for each possible next command do
18:      Output the next state in the FSM.
19:   end for
20: end for
```

c_{new} and the previous n commands that were sent to memory. We refer to the previous n commands as c_1, c_2,…, c_n, where c_1 is the most recent command sent and c_n is the oldest command sent.

We define k cost functions, $f_{1..k}(c_x, c_y)$, to represent the mandatory delays between any two memory commands, c_x and c_y, that cause a hardware hazard. Here, both k and the cost functions are memory system-dependent. For our system, we have cost functions for "the delay between a Write to a different bank after a Read", "the delay between a Read to the same port after a Write", "the delay between a Read to the same port but to a different rank after a Read", etc.

We assume that the scheduler does not have the ability to track the number of cycles passed since the previously issued commands were sent. So, our algorithm assumes that those commands were sent at one cycle intervals. In the next step, the algorithm calculates the delays imposed by each c_x, $x \in [1, n]$ on c_{new} for each function, $f_{i..k}$, which is applicable to any (c_x, c_{new}) pair. Here, the term "applicable function" refers to a function whose conditions have been satisfied. We also define n final cost functions, $fcost_{i..n}$, such that

$$fcost_i(c_{new}) = max(f_j(c_i, c_{new})) - (i - 1)$$

where $i \in [1, n]$, $j \in [1, k]$, and $f_j(c_i, c_{new})$ is applicable

We take the maximum of f_j function values because any previous command, c_i, and c_{new} may be related by more than one f_j function. In this formula, the subtracted term $(i - 1)$ represents the number of cycles c_i that had been sent before c_{new}. Thus, the expected latency that will be introduced by sending c_{new} is

$$T_{delay}(c_{new}) = max(fcost_{1..n}(c_{new}))$$

Algorithm 2 generates a FSM for a scheduler that uses the expected latency, T_{delay}, to prioritize the commands. As with the previous algorithm, if multiple available commands have the same expected latency, we use a secondary criterion—

in this case the deviation from the command pattern—to break ties.

Algorithm 2 expected_latency_scheduler(n)

```
// $n$ is the history length
for all command sequences of size $n$ do

    for each possible next command do
        Calculate the expected latency, $T_{delay}$.
    end for
    Sort possible commands with respect to $T_{delay}$.
    for commands with equal expected latency value do
        Use Read/Write ratios to make decisions.
    end for

    for each possible next command do
        Output the next state in the FSM.
    end for
end for
```

A Probabilistic Scheduler Design Algorithm

To combine our two optimization criteria, Algorithm 3 weighs each criterion and produces a probabilistic decision. At runtime, a random number is periodically generated to determine the rules for state transitions as follows:

Algorithm 3 probabilistic_scheduler

```
if random_number < threshold then
    command_pattern_scheduler
else
    expected_latency_scheduler
end if
```

Basically, we interleave two state machines into one, periodically switching between the two in a probabilistic manner. In this approach, the threshold value is system dependent and should be determined experimentally.

3.1.3 Adaptive Selection of Schedulers

Our adaptive history-based scheduler is schematically shown in Figure 3.2. The memory controller tracks the command pattern that it receives from the processors and periodically switches among the schedulers depending on this pattern.

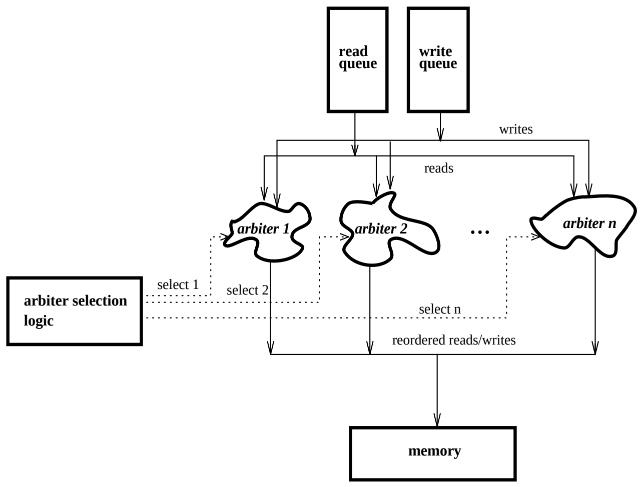

Figure 3.2: Overview of dynamic selection of arbiters in memory controller.

Detecting Memory Command Pattern

To select one of the history-based arbiters, our memory controller assumes the availability of three counters: *Rcnt* and *Wcnt* count the number of reads and writes received from the processor, and *Ccnt* provides the period of adaptivity. Every *Ccnt* cycles, the ratio of the values of *Rcnt* and *Wcnt* is used to select the most appropriate history-based scheduler. The Read/Write ratio can be calculated using left shift and addition/subtraction operations; since this computation is performed once every *Ccnt* cycles, its cost is negligible. To prevent retried commands from skewing the command pattern, we distinguish between new commands and retried commands, and only new commands affect the value of *Rcnt* and *Wcnt*. The values of *Rcnt* and *Wcnt* are set to zero when *Ccnt* becomes zero.

3.2 Experimental Results

In this section, we evaluate the AHB scheduler and compare its performance to the previous scheduling approaches. First, we identify a baseline by comparing previous scheduling methods. Then, using the Stream, NAS, and commercial benchmarks, we compare performance of our approach to the baseline. Finally, we use microbenchmarks to investigate performance bottlenecks in the memory subsystem. Our results show that the AHB scheduler is always superior to the previously proposed methods. We also see that the scheduler plays a critical role in balancing various bottlenecks in the system.

3.2.1 Evaluating Previous Approaches

We compare our AHB scheduler against a set of schedulers that use previously proposed ideas. To cover the full design space, we identify three main features of memory controllers: the approach to handle bank conflicts, the bank scheduling method, and the priorities for reads and writes.

The first feature specifies the scheduler's behavior when selected command has a bank conflict, of which two choices have been proposed: 1) the scheduler can hold the conflicting command in the reorder queues until the bank conflict is resolved, or 2) the scheduler can transmit the command to the CAQ.

The second feature, the bank scheduling method, provides a method of scheduling commands to banks. We consider three approaches: *in-order*, *LRU*, and *round-robin*. The first, in-order, implements the simple FIFO policy used by most general purpose memory controllers today. If implemented in a Power5+ system, this scheduler would transmit memory commands from the reorder queues to the CAQ in the order in which they were received from the processors. In terms of implementation cost, in-order scheduling is the simplest method among all three scheduling approaches. The second scheduling approach, LRU, gives priority to commands with bank numbers that were least recently scheduled. If there is more than one such commands, the scheduler will switch to the in-order approach and pick the oldest command. To obtain maximum advantage from the LRU method, we assume true-LRU, which may be unreasonably costly to implement. Finally, the round-robin scheduling technique tries to utilize banks equally by imposing a strict round-robin access to the banks. To guarantee forward progress, we implement a modified version of round-robin. In our implementation, if the reorder queues have no command to satisfy the bank sequence but they do have other commands, the round-robin scheduler picks a command that is closest to the optimal sequence. As with the LRU approach, if there are multiple commands to the bank, the scheduler uses an in-order policy and selects the oldest such command.

The third design feature describes how commands are selected from Read and Write reorder queues. We evaluate two approaches: 1) every read or write command has equal priority, and 2) reads have higher priority over writes. We believe, in general, that giving higher priority to reads will improve performance. To prevent starvation of writes, we evaluate Rixner et al.'s techniques in which writes are given higher priority if either of the following conditions exists: i) there is a write command that waited too long, or ii) the write reorder queue is about to become full. For both of these conditions the memory controller needs threshold values. Determining these thresholds is not straightforward and may be application dependent.

For our studies, we emphasize these three features as follows. Since bank conflict costs are high, our implementations use the first design feature to reduce the number of candidate commands in the reorder queues. Then, from each of the reorder queues, the scheduler identifies one command that satisfies the bank scheduling approach. Finally, the read/write priorities are used to select the command.

Since we identify three bank scheduling methods, two priority approaches, and two choices bank conflicts, we evaluate a total of twelve points in the design space. In the next subsection, we compare the performance of these twelve points in the design space and select the baseline to compare with our AHB scheduler.

We can now describe our AHB scheduler in relation to these three design features. The AHB scheduler holds the commands in the reorder queues if there is a bank conflict. Our scheduler then uses the adaptive history-based technique described in Section 3.1 to select the most appropriate command from among the remaining commands in the reorder queues. In other words, our adaptive history-based approach is used to handle rank and port conflicts, but not bank conflicts. Our method also combines the scheduling with and read/write priorities, so that it eliminates the need to determine thresholds for priority selection. In short, the AHB scheduler uses a single new mechanism to implement the first and the third design features and it uses a simple mechanism for deciding how to deal with bank conflicts.

In our implementation of the schedulers, we augment the previous proposals to make them suitable for the Power5+ memory controller. To determine the representative schedulers, we conduct experiments on one SMT processor using the Stream benchmarks.

Table 3.1 illustrates that out of the three criteria, bank hold policy has the greatest, up to 46%, effect on performance. We observe that any method that holds commands with bank conflicts is better than its counterpart that doesn't hold the commands. Among the six approaches that holds for bank conflicts, rd/wr priority seems more important than the bank scheduling method. Actually, effect of bank scheduling policy is as high as 45% among the methods, LRU being the best, that don't hold banks. However, performance gains from holding banks obviate the need for a complicated bank scheduling method. In terms of implementation complexity, fifo bank scheduling is the simplest approach. Therefore, we determine "hold, fifo,

bank hold, scheduler, rd/wr prio.	daxpy	copy	scale	vsum	triad	fill	sum	geom. mean
don't hold, fifo, equal prio. **(in-order)**	1.987	3.142	2.131	2.001	2.005	2.265	0.851	1.938
don't hold, fifo, read prio.	1.260	2.164	1.474	1.542	1.561	2.121	0.650	1.448
don't hold, lru, equal prio.	0.895	1.557	1.072	1.060	1.061	1.783	0.527	1.067
don't hold, lru, read prio.	0.856	1.467	1.006	1.003	1.004	1.825	0.864	1.105
don't hold, round-robin,eq. prio.	1.118	1.812	1.242	1.244	1.246	2.007	0.555	1.233
don't hold, round-robin,read prio.	1.119	1.776	1.211	1.213	1.219	2.018	0.555	1.219
hold, fifo, equal prio.	0.866	1.475	1.014	1.028	1.032	1.798	0.515	1.035
hold, fifo, read prio. **(memoryless)**	0.825	1.487	1.020	0.978	0.977	1.775	0.517	1.014
hold, lru, equal prio.	0.855	1.507	1.038	1.017	1.017	1.782	0.560	1.047
hold, lru, read prio.	0.846	1.463	0.999	0.982	0.980	1.800	0.515	1.014
hold, round-robin, equal prio.	0.808	1.463	1.001	0.956	0.957	1.786	0.569	1.014
hold, round-robin, read prio. **(best)**	0.824	1.478	1.013	0.973	0.969	1.783	0.521	1.011

Table 3.1: Performance (in CPI) of the Previous Scheduling Approaches for the Stream Benchmarks.

read priority" approach, which we call *memoryless*, as the first baseline for our study. Note that in our previous work [28, 29], we used the term memoryless for "hold, fifo, equal priority" method, which is a slightly inferior method.

In addition to the memoryless method, we also select "don't hold, fifo, equal priority" approach, i.e. *in-order*, as the second approach to compare with our AHB scheduler. We choose in-order scheduler as the second baseline, because most current processors implement this approach due to its simple implementation cost.

3.2.2 Tuning the AHB Scheduler

The AHB scheduler has three parameters, namely history length, epoch length, and the weighting of the two optimization criteria. In this subsection we tune these parameters using daxpy benchmark and assuming there are two active threads on one processor.

History Length. We compare four AHB schedulers whose history lengths range between 1 and 4. Table 3.2(a) shows that a history length of 2 is superior to history

(a) Effects of History Length	
History Length	CPI
1	0.743
2	0.696
3	0.684
4	0.684

(b) Effects of Epoch Length	
Epoch Length	CPI
100	0.712
500	0.703
1000	0.694
5000	0.696
10000	0.696

(c) Effects of Ratio for Optimization Criteria	
Weight of Expected Latency (%)	CPI
0	0.713
10	0.708
20	0.711
30	0.712
40	0.700
50	0.704
60	0.697
70	0.696
80	0.699
90	0.703
100	0.709

Table 3.2: Tuning of the AHB Scheduler.

length of 1 by 6.4%. However, using longer history lengths longer than 2 improves performance by only 1.8%. Therefore, considering the implementation cost, all experiments in this study use an AHB scheduler with a history length of 2.

Epoch Length. We vary epoch length from 100 to 10,000 processor cycles. Table 3.2(b) illustrates that any length over 1,000 cycles gives essentially the same performance. We choose 10,000 processor cycles as the epoch length in our study.

Ratio for Optimization Criteria. The AHB scheduler optimizes for two criteria, namely the expected latency and the command pattern. As we describe in Section 3.1, our approach combines two criteria probabilistically by giving weights to each criterion. Table 3.2(c) shows that we obtain the best performance when we assign the expected latency a weight of 70% and the command pattern a weight of 30%.

3.2.3 Benchmark Results

We now present simulation results for the AHB, in-order, and memoryless schedulers using the Stream, NAS, and commercial benchmarks. For the Stream and NAS benchmarks, we simulate one or two threads on one processor. For the commercial benchmarks, we simulate one or two threads on single or dual core systems.

We first compare the single thread performance of the three schedulers for the Stream benchmarks (see Table 3.3). The geometric means of the performance benefit of the AHB scheduler over the in-order and the memoryless schedulers are 45.5% and 11.3% respectively. For two threads on a processor, adaptive history-based scheduling improves execution time by an average of 55.6% over the in-order scheduler and 16.0% over the memoryless scheduler.

Our second set of results are for the NAS benchmarks, which provide a more comprehensive evaluation of overall performance. Table 3.4 shows that for the single thread experiments, the average improvement of our approach over the in-order method is 16.8%, and the average improvement over the memoryless method is 5.8%. In the SMT experiments, we use two threads of the same application, and the AHB scheduler improves performance by 25.6% and 9.7% over the in-order and memoryless schedulers, respectively.

Finally, in Table 3.5, we present the results for the commercial benchmark suite running on single and dual core systems, with one or two threads active on each processor, resulting in four different configurations. For the single threaded case on a single processor, the AHB scheduler has, on the average, a 12.6% performance advantage over the in-order scheduler and a 2.9% advantage over the memoryless

Benchmark	in-order	memoryless	AHB	gain over in-order (%)	gain over memoryless (%)
One Thread on One Processor					
daxpy	1.933	0.785	0.712	63.2	9.3
copy	3.576	1.578	1.312	63.3	16.9
scale	2.467	1.082	0.932	62.2	13.9
vsum	2.083	1.008	0.877	57.9	13.0
triad	2.088	1.007	0.884	57.7	12.2
fill	2.321	1.696	1.547	33.3	8.8
sum	0.854	0.793	0.730	14.5	7.9
Two Threads on One Processor					
daxpy	1.987	0.825	0.696	68.2	16.4
copy	3.142	1.487	1.212	64.5	19.4
scale	2.131	1.020	0.833	64.0	19.3
vsum	2.001	0.978	0.837	61.1	15.1
triad	2.005	0.977	0.838	61.1	14.9
fill	2.265	1.775	1.518	33.0	14.5
sum	0.851	0.517	0.447	47.5	13.6

Table 3.3: Comparison of CPI's of the AHB scheduler to the in-order and memoryless schedulers for the Stream benchmarks.

Benchmark	in-order	memoryless	AHB	gain over in-order (%)	gain over memoryless (%)
One Thread on One Processor					
bt	0.960	0.883	0.838	12.7	5.1
cg	1.841	1.712	1.582	14.1	7.6
ep	2.465	2.219	2.118	14.0	4.6
ft	2.743	2.277	2.074	24.4	8.9
is	2.370	1.990	1.861	21.5	6.5
lu	2.455	2.013	1.872	23.7	7.0
mg	1.327	1.155	1.088	18.0	5.8
sp	1.502	1.380	1.335	11.1	3.3
Two Threads on One Processor					
bt	1.005	0.781	0.721	28.3	7.7
cg	1.806	1.532	1.365	24.4	10.9
ep	2.151	1.971	1.798	16.4	8.8
ft	2.655	2.027	1.780	33.0	12.2
is	2.145	1.616	1.440	32.9	10.9
lu	2.012	1.732	1.561	22.4	9.9
mg	1.108	0.930	0.819	26.1	11.9
sp	1.365	1.086	1.012	25.9	6.8

Table 3.4: Comparison of CPI's of the AHB scheduler to the in-order and memoryless schedulers for the NAS benchmarks.

Benchmark	in-order	memoryless	AHB	gain over in-order (%)	gain over memoryless (%)
One Thread on One Processor					
tpcc	15.458	14.222	13.798	10.7	3.0
cpw2	15.366	14.092	13.738	10.6	2.5
trade2	15.728	14.326	14.052	10.7	1.9
sap	10.268	8.542	8.112	21.0	2.9
Two Threads on One Processor					
tpcc	11.572	9.304	8.890	23.2	4.4
cpw2	11.274	8.746	8.396	25.5	4.0
trade2	11.152	8.726	8.380	24.9	4.0
sap	8.406	5.506	5.206	38.1	5.4
One Thread on Each of the Two Processors					
tpcc	10.576	7.913	7.518	28.9	5.0
cpw2	10.611	7.760	7.335	30.9	5.5
trade2	10.431	7.749	7.291	30.1	5.9
sap	7.896	4.780	4.494	43.1	6.0
Two Threads on Each of the Two Processors					
tpcc	9.733	5.401	5.037	48.2	6.7
cpw2	9.744	5.153	4.773	51.0	7.4
trade2	9.593	5.100	4.766	50.3	6.5
sap	7.367	3.483	3.151	57.2	9.5

Table 3.5: Comparison of CPI's of the AHB scheduler to the in-order and memoryless schedulers for the commercial benchmarks.

scheduler. As the total number of threads increases to two, we observe that the AHB scheduler's advantage increases to 27.4% and 4.4% on a single core system, and to 32.8% and 5.6% on a dual core system. For two threads running on each of two processors, the gain from the AHB scheduler is 51.6% over the in-order scheduler and 7.5% over the memoryless scheduler.

In summary, our experiments with the Stream, NAS, and commercial benchmarks indicate that the AHB scheduler is superior to the in-order and memoryless schedulers. We also see that the benefit of our approach increases as the total number of threads in the system increases, because additional threads increase pressure on the single memory controller.

3.2.4 Understanding the Results

We now look inside the memory system to gain a better understanding of our results. To study a broader set of hardware configurations, we use a set of 14 microbenchmarks, ranging from 4 Read streams and 0 Write streams, to 0 Read streams and 4 Write streams. Figure 3.3 shows that for these microbenchmarks, the adaptive history-based method improves performance by 20-70% compared to in-order scheduler and by 17-20% compared to memoryless scheduler.

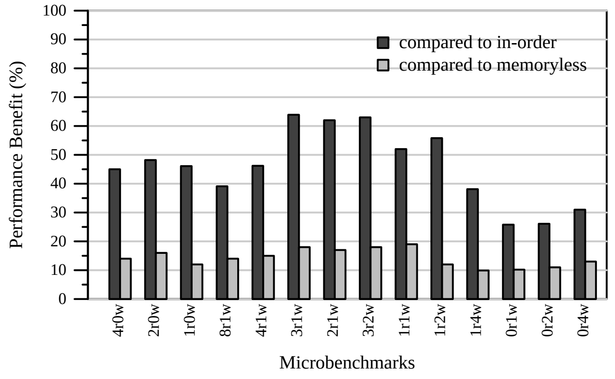

Figure 3.3: Performance comparison on our microbenchmarks.

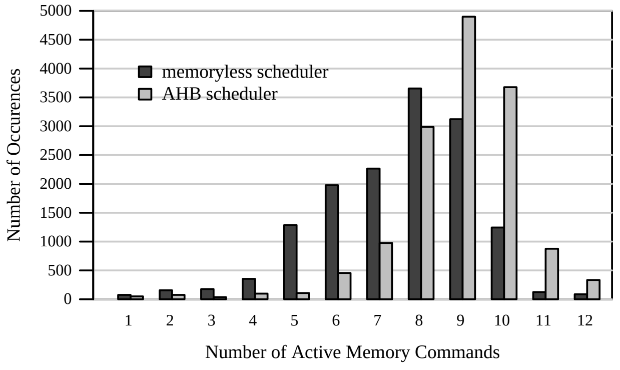

Figure 3.4: Utilization of the DRAM for the daxpy kernel.

The most direct measure of the quality of a memory controller is its impact on memory system utilization. Figure 3.4 shows a histogram of the number of operations that are active in the memory system on each cycle. We see that when compared against the memoryless scheduler, our scheduler increases the average utilization from 8 to 9 operations per cycle. The x-axis goes to 12 because the Power5+'s DRAM allows 12 memory commands to be active at once.

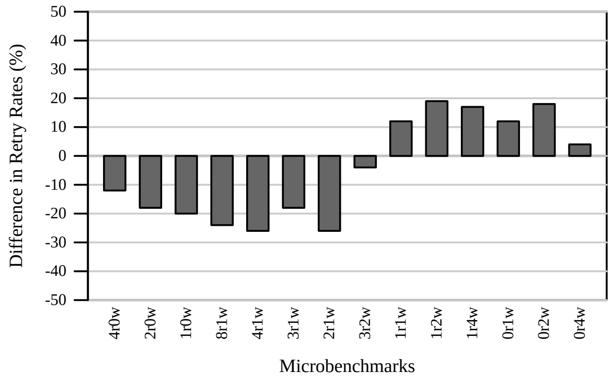

Figure 3.5: Comparison of retry rates.

Memory system utilization is also important when evaluating our results, because it is easier for a scheduler to improve the performance of a saturated system. We measure the utilization of the command bus that connects the memory controller to the DRAM, and we find that the utilization was about 65% for the Stream benchmarks and about 13%, on average, for the NAS benchmarks. We conclude that the memory system was not saturated for our workloads.

Bottlenecks in the System. To better understand why our solution improves DRAM utilization, we now examine various potential bottlenecks within the memory controller.

The first potential bottleneck occurs when the reorder queues are full. In this case, the memory controller must reject memory operations, and the CPU must retry the memory operations at a later time. The retry rate does not correlate exactly to performance, because a retry may occur when the processor is idle waiting for a memory request. Nevertheless, a large number of retries hints that the memory system is unable to keep up with the processor's memory demands. Figure 3.5 shows that the adaptive history-based method always reduces the retry rate when compared to the in-order method, but it sometimes increases the retry rate compared to the memoryless method.

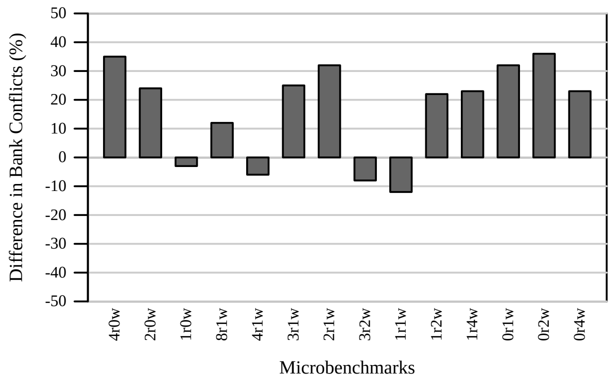

Figure 3.6: Comparison of the number of bank conflicts in the reorder queues.

A second bottleneck occurs when no operation in the reorder queues can be issued because of DRAM conflicts with previously scheduled commands. This bottleneck is a good indicator of scheduler performance, because a large number of such cases suggests that the scheduler has done a poor job of scheduling memory operations. Figure 3.6 compares the total number of such blocked commands for our method and for the memoryless method. This graph only considers cases where the reorder queues are the bottleneck, i.e., all operations in the reorder queues are blocked even though the CAQ has empty slots. We see that except for four microbenchmarks, our method substantially reduces the number of such blocked operations.

A third bottleneck occurs when the reorder queues are empty, starving the scheduler of work. Even when the reorder queues are not empty, low occupancy in the reorder queues is bad because it reduces the scheduler's ability to make good scheduling decisions. In the extreme case, where the reorder queues hold

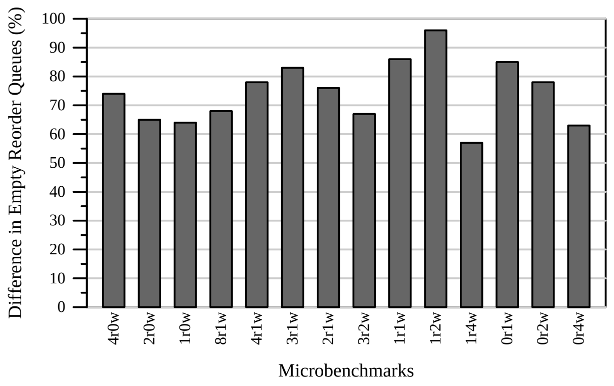

Figure 3.7: Reduction in the occurrences of empty reorder queues, which is a measure of the occupancy of the reorder queues.

no more than a single operation, the scheduler has no ability to reorder memory operations and instead simply forwards the single available operation to the CAQ. Figure 3.7 shows that our method significantly reduces the occurrences of empty reorder queues, indicating higher occupancy of these queues.

The final bottleneck occurs when the CAQ is full, forcing the scheduler to remain idle. Figure 3.8 shows that the adaptive history-based scheduler tremendously increases this bottleneck. The backpressure created by this bottleneck leads to higher occupancy in the reorder queues, which is advantageous because it gives the scheduler a larger scheduling window.

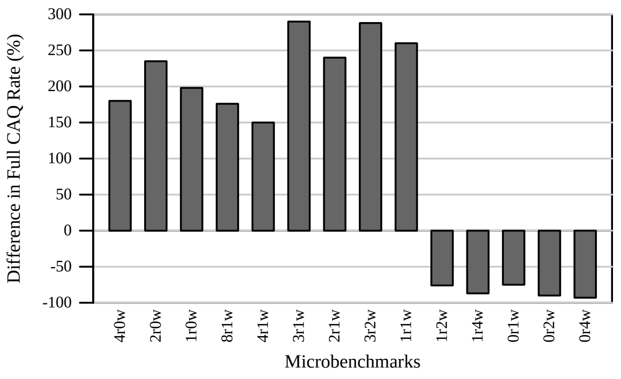

Figure 3.8: Increases in the occurrences where the CAQ is the bottleneck.

To test this theory, we conduct an experiment in which we increase the size of the CAQ. We find that as the CAQ length increases, the CAQ bottleneck decreases, the reorder queue occupancy falls, and the overall performance decreases.

In summary, our solution improves bandwidth by moving bottlenecks from outside the memory controller, where the scheduler cannot help, to inside the memory controller. More specifically, the bottlenecks tend to appear at the end of the pipeline—at the CAQ—where there is no more ability to reorder memory commands. By shifting the bottleneck, our solution tends to increase the occupancy of the reorder queues, which gives the scheduler a larger number of memory operations to choose from. The result is a smaller number of DRAM conflicts and increased bandwidth.

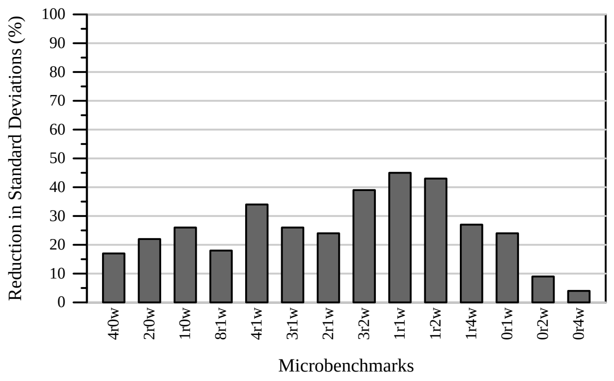

Figure 3.9: Reduction in standard deviations for 16-different address offsets.

Effects of Data Alignment. Another benefit of improved memory scheduling is a reduced sensitivity to data alignment. With a poor scheduler, data alignment can cause significant performance differences. The largest effect is seen where a data structure fits on one cache line when aligned fortuitously but straddles two cache lines when aligned differently. In such cases, the bad alignment results in twice the number of memory commands. If a scheduler can improve bandwidth by reordering commands, it can mitigate the difference between the well-aligned and poorly-aligned cases. Figure 3.9 compares the standard deviations of the adaptive history-based and memoryless schedulers when data are aligned on 16 different address offsets. We see that the adaptive history-based solution reduces the sensitivity to alignment.

3.3 Sensitivity Analysis

The previous section analyzed the performance of the AHB scheduler in the context of the IBM Power5+. This section explores the broader utility of our scheduler by analyzing its performance in the context of various derivatives of the Power5+.

There are three goals of this section. First, we would like to analyze the sensitivity and robustness of the AHB scheduler to various micro-architectural features. We will show that the AHB scheduler yields performance that is robust across a variety of micro-architectural parameters. We will also see that the other schedulers cannot achieve the performance of the AHB approach even if given additional hardware resources. Second, we identify optimal values for parameters related to the memory scheduler. We show that carefully determining memory system parameters has significant performance implications. And finally, we want to evaluate our approach for possible future architectural trends.

In the following subsections, we first investigate the performance effects of varying the parameters of the memory controller. Then, we analyze the effects of various DRAM parameters. And lastly, to explore the applicability of our approach in possible future systems, we compare the schedulers for systems with different processor frequencies and different data prefetching options.

We evaluate our scheduler with single and multiple-threads, and we make comparisons to the memoryless scheduler. We use daxpy benchmark in our experiments, because daxpy occurs very frequently in scientific workloads, and architectural parameters are considered difficult to tune for this benchmark.

3.3.1 Memory Controller Parameters

There are numerous memory controller design features that affect performance. In this subsection, we compare the AHB and the memoryless scheduling methods by varying memory controller features. Since the design space is large, we identify three important parameters to vary: the CAQ length, the reorder queue lengths, and the duration to block a command in the reorder queues when there is a bank conflict. We believe that these features are the most important parameters with respect to performance.

CAQ Length. The Central Arbiter Queue resides between the memory scheduler and DRAM. At each cycle, the scheduler selects an appropriate command from the reorder queues and feeds it to the CAQ. Since the CAQ acts as a buffer between the scheduler and DRAM, the length of this queue is critical to performance. Here, we examine the performance effects of the CAQ length. For various configurations and schedulers, we first determine the optimal length for the queue. We then analyze the sensitivity of the scheduling approaches to the changes in this length. Our experiments show that the AHB scheduler is superior to the memoryless scheduler for all CAQ lengths that we study.

The CAQ length may degrade performance if it is either too short or too long. If the queue is too short, it will tend to overflow frequently and lead to full reorder queues, which will cause the memory controller to reject memory commands from the processor and degrade overall performance. We can reduce the occurrence of CAQ overflows by increasing the CAQ length, but a long CAQ has its own disadvantages. First, it consumes more hardware resources, as the Power5+ memory controller's hardware budget is dominated by the reorder queues and CAQ. Second, as explained in Section 3.2.4, a long CAQ can reduce backpressure on the reorder queues, giving the scheduler a smaller effective scheduling window, which leads to suboptimal scheduling decisions. Therefore, the CAQ acts as a regulator for the rate of commands to be selected from the reorder queues, and there is a delicate balance between the CAQ length and performance.

We conduct experiments in which we vary the CAQ length from 2 to 16. In Figure 3.10, we show the effect of the CAQ length for both Single-Threaded (ST) and SMT environments. For the ST daxpy, the AHB scheduler gets the best performance for a queue length of 4. As the queue length increases beyond 4, there is a slight performance degradation. For the SMT case, a queue length of 3 gives the best performance for the AHB method. Similar to the ST case, as the CAQ length increases beyond the optimal value, we observe performance degradation. But unlike the ST case, the performance degradation is not small. For example, performance is 1.7% lower for the queue length of 4 compared to the length of 3. This performance difference goes up to 4.4% when the queue has 16 slots.

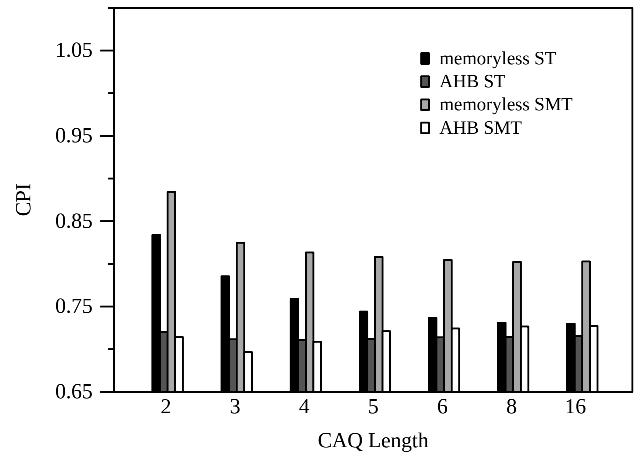

Figure 3.10: ST and SMT results for the memoryless and the AHB with varying lengths of the CAQ.

Figure 3.10 also shows that for the memoryless scheduler, longer CAQs always yield better performance, most likely because the memoryless scheduler has no way to exploit larger scheduling windows. For example in the ST case, the performance of the memoryless scheduler improves by 7.1% as the CAQ length increases from 3 to 16. However, even with this queue length, our approach is still superior over the memoryless scheduler. In the SMT experiments with the memoryless scheduler, we find that the performance gain from increasing the queue size to 16 is much smaller compared to the ST case.

In summary, the memoryless method improves as the CAQ gets longer, but it cannot achieve the performance of the AHB scheduler even if given a much longer CAQ. We also conclude that selecting the optimal queue length has significant performance effects.

Reorder Queue Lengths. As we show in Figure 2.1, the Power5+ has two reorder queues inside the memory controller: one for reads and one for writes. In the current design of the Power5+, each of these queues have equal length of 8. Here, we analyze the effect of the reorder queue lengths on the scheduling approaches.

The length of the reorder queus affects performance in two ways. First, retries occur when the reorder queues are full, so shorter reorder queues increase the number of retries and potentially decrease overall performance. Second, if the reorder queues are short, the scheduler will have limited optimization capability. In the extreme case, consider a reorder queue with just one slot. The scheduler will have no choice but select the command from that slot. We, therefore, expect that increasing the size of the reorder queues will improve the performance of any scheduling approach.

We perform simulations that vary the reorder queue lengths from 4 to 16. For simplicity, we always keep the lengths of the two queues the same. In Figure 3.11, we present the effects of the reorder queue lengths on performance for both the AHB and the memoryless schedulers. For the single threaded experiments, as we shorten the queue sizes from the Power5+'s current value of 8 to 4, the AHB scheduler loses

28.8% of its performance and memoryless scheduler loses 25.3%. The same reduction in the reorder queue lengths for the SMT experiments degrades performance 27.3% and 19.9% for the AHB and memoryless schedulers, respectively. On the other hand, for both of the scheduling approaches, when we increase the reorder queue lengths beyond the current value of 8, we obtain only very small performance improvements.

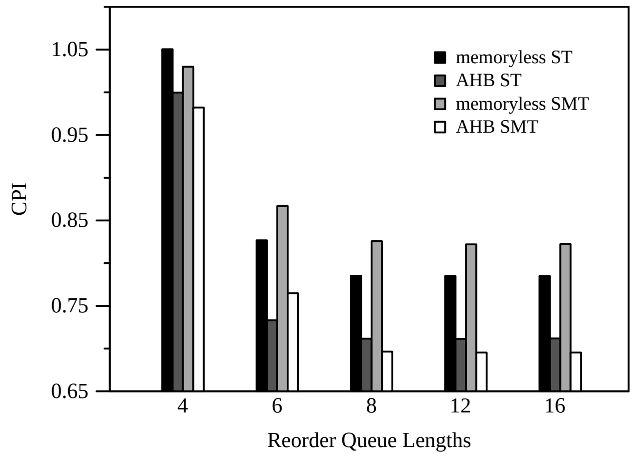

Figure 3.11: ST and SMT results for memoryless and AHB with various reorder queue lengths.

We conclude that for all the reorder queue sizes, the performance of the AHB approach is better than the memoryless method. As we expect, the advantage of the AHB method over the memoryless method increases as the queues become longer. We also observe that the current queue lengths are optimal for the Power5+. We cannot obtain any significant performance gains with longer queues regardless of the scheduling approach or the number of threads.

Wait Times for Commands with Bank Conflicts. In this section, we analyze the interaction between the scheduler and the blocking duration for commands with bank conflicts. We find that the AHB is less sensitive to this parameter and is always better than the memoryless scheduler regardless of the wait time.

Bank conflicts prohibit the entrance of new commands to DRAM. Since the

CAQ is a FIFO queue, if the command in front of the CAQ conflicts with a command in DRAM, all the commands in the CAQ are blocked until the conflict is cleared. To prevent this, the Power5+ holds commands in the reorder queues when they have bank conflicts. Even with an empty CAQ, a command in the reorder queues has to travel some distance before it is issued to DRAM. This distance is about 32 processor cycles in the current implementation. To avoid this 32 cycle delay, the Power5+ transmits commands to the CAQ some number of cycles before the bank conflict is expected to be resolved.

This *wait time* in the reorder queues is important to performance. If the wait time is too short, commands with bank conflicts will be scheduled early, yielding two possible effects: First, the CAQ may contain multiple commands to the same bank, and when one of these commands goes to DRAM, the others will be blocked for many cycles. Second, if the command is scheduled too early, the schedule may miss the opportunity to make a better scheduling decision when additional commands might become available in the reorder queues.

To investigate the effects of various wait times, we conduct experiments for the AHB and the memoryless schedulers with ST and SMT. As we see in Figure 3.12, the AHB scheduler is much less sensitive to the wait time. For the AHB scheduler, 95 processor cycles is the optimal wait time for both ST and SMT experiments. If a command waits until the bank conflict is cleared, this will degrade performance by 1.8% for ST and 3.5% for SMT. For the memoryless approach, 125 and 110 cycles are the optimal wait times for ST and SMT, respectively. The memoryless method with SMT has a 1.2% performance advantage when it uses 110 cycle wait time rather than 125 cycles.

In summary, we observe that the scheduler should be able to select a command from the reorder queues earlier than the bank conflict is cleared. We also find that for the ST case, the AHB approach is less sensitive to this parameter. For the

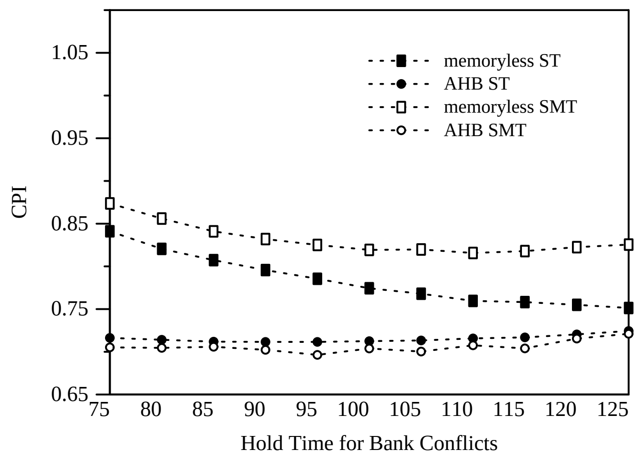

Figure 3.12: ST and SMT results for the memoryless and the AHB with varying wait times for bank conflicts.

SMT, both scheduling approaches show similar sensitivity. For all the wait times that we study, the AHB scheduler has better performance than the memoryless scheduler.

3.3.2 DRAM Parameters

In this section we vary DRAM system parameters. In particular, we evaluate the performance of the AHB and the memoryless methods by varying the memory address and data bus widths, the maximum number of commands that can be active in DRAM, and the number of banks available in a rank. We find that each of these three parameters significantly affects performance.

Address and Data Bus Widths. Memory bus width significantly affects a memory system's bandwidth, so we explore the effect of using both narrower and wider memory buses for the Power5+. The Power5+ memory controller is connected to memory chips via an address bus and a data bus. In the current implementation, the address bus is 32 bits wide. The data bus has 24 bits: 16 bits for Reads and 8 bits for Writes.

In Figure 3.13 the x-axis represents the relative ratio of the bus widths to the current values of the Power5+. For example, 0.5 represents a system with buses half the width of the current system. We find that reducing bus widths by 50% significantly degrades performance (20.9-26.6%) for both the AHB and memoryless schedulers. We also observe that increasing bus widths beyond the current values of the Power5+ has little effect on performance. For all the bus widths we study, the AHB's performance is higher than the memoryless.

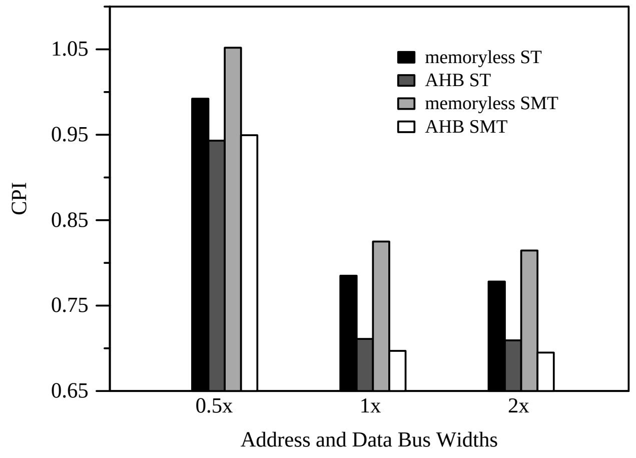

Figure 3.13: ST and SMT results for memoryless and AHB, varying memory address and data bus widths.

Maximum Number of Commands in DRAM. In the systems we examine, the DRAM is organized into 16 banks, so there can be a maximum of 16 concurrent commands in DRAM. However, the Power5+ designers choose to track at most 12 commands at any time. To explore the benefit of tracking more than 12 commands, we vary the number of commands tracked. In Figure 3.14, we show results for both ST and SMT workloads. We find that increasing beyond 12 the number of commands to track in DRAM does not increase performance. However, reducing its value by 4 reduces daxpy performance up to 7.9%.

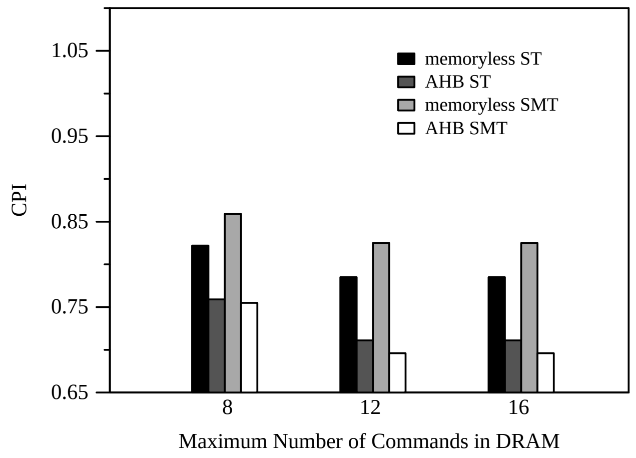

Figure 3.14: ST and SMT results for memoryless and AHB, varying the maximum number of DRAM commands.

Number of Banks in a Rank. Future memory systems are likely to provide increased parallelism in the form of a larger number of banks per rank. Figure 3.15 shows how performance is affected by changing the number of banks. Increasing the banks per rank from two to four improves performance in both the single threaded and the SMT experiments. The performance gain is 20.8%-21.7% and 18.1%-26.6% for the AHB and memoryless schedulers, respectively. On the other hand, further increasing the number of banks to eight does not improve the performance of the memoryless scheduler, and the performance gain for the AHB scheduler is between 1.9% and 4.6% for the single threaded and SMT experiments. In summary, our experiments indicate that the advantage of the AHB scheduler over the memoryless approach increases as the number of banks in a rank increases, *i.e.*, as the memory system admits more parallelism.

3.3.3 System Parameters

Processor Frequency. In addition to memory controller and DRAM parameters, we also explore the impact of higher clock rates for the processor. While increases in

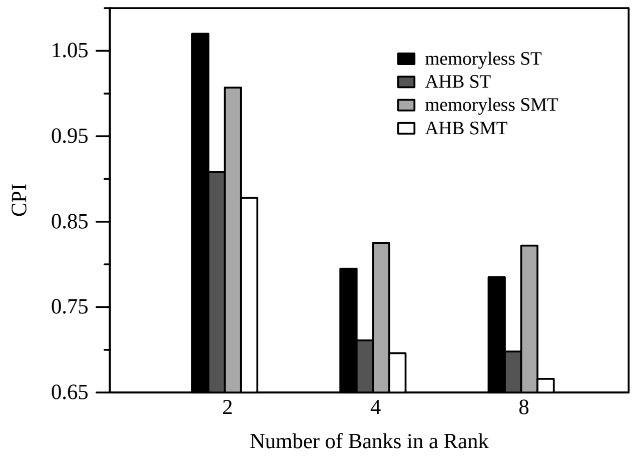

Figure 3.15: ST and SMT results for the memoryless and the AHB with varying number of banks in a rank.

clock rate have slowed, processor frequency continues to increase. In Figure 3.16, we present the differences between the AHB and the memoryless schedulers for systems with 1.5, 2, 3, and 4 times the processor frequency of the current Power5+ systems. As the ratio of the processor frequency to the DRAM frequency grows, we find that advantage of the AHB scheduler over the memoryless method also increases. For example, for the ST case, with the current processor frequency, the AHB scheduler is superior to the memoryless scheduler by 9.5%, but the advantage grows to 15.6% when the processor frequency doubles. Similarly, for the SMT case, AHB method's advantage increases from 15.5% to 22.0% with 2x processor frequency. We conclude that as the ratio of the processor/memory speeds increases, the significance of our approach will also increase because the importance of memory bandwidth grows.

Data Prefetching. We also investigate the effects of data prefetching on the scheduling approaches. We see that if we turn off the prefetch unit, the adaptive history-based method's benefit over the other two approaches is significantly diminished because the lower memory traffic reduces pressure on the memory controller. For example, for daxpy in the SMT case, the performance benefit of the

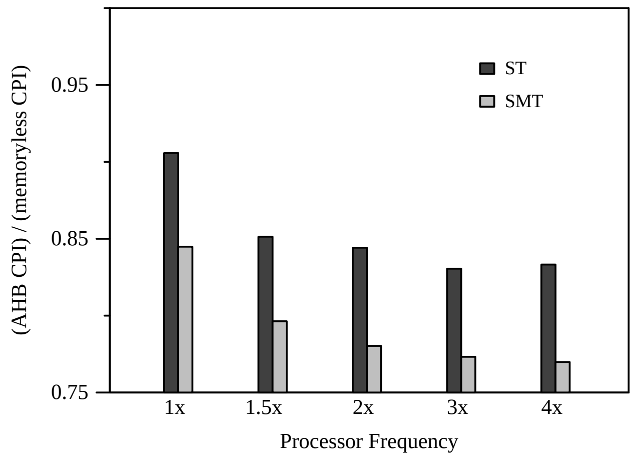

Figure 3.16: ST and SMT results for memoryless and AHB, with 1.5x, 2x, 3x, and 4x processor frequency.

AHB scheduler over the memoryless scheduler is reduced from 16.4% to 7.3% when the hardware prefetching unit is turned off.

3.4 Hardware Costs

To evaluate the cost of our solution, we need to consider the cost in terms of transistors and power. The hardware cost of the memory controller is dominated by the reorder queues, which dwarf the amount of combinational logic required to implement our adaptive history-based arbiter. To quantify these costs, we use the implementation of the Power5+ to provide detailed estimates of transistor counts. We find that the memory controller consumes 1.58% of the Power5+'s total transistors. The size of one memoryless arbiter is in turn 1.19% of the memory controller. Our adaptive history-based arbiter increases the size of the memory controller by 2.38%, which increases the overall chip's transistor count by 0.038%. Given the tiny cost in terms of transistors, we are confident that our solution has only negligible effects on power.

3.5 Summary

In this chapter, we have shown that memory access scheduling, which has traditionally been important primarily for stream-oriented processors, is becoming increasingly important for general-purpose processors, as many factors contribute to increased memory bandwidth demands. To address this problem, we have introduced a new scheduler that incorporates several techniques. We use the command history—in conjunction with a cost model—to select commands that will have low latency. We also use the command history to schedule commands that match some expected command pattern, as this tends to avoid bottlenecks within the reorder queues. Both of these techniques can be implemented using FSM's, but because the goals of the two techniques may conflict, we probabilistically combine these FSM's to produce a single history-based scheduler that partially satisfies both goals. Finally, because we cannot know the actual command-pattern *a priori*, we implement three history-based schedulers—each tailored to a different command pattern—and we dynamically select from among these three schedulers based on the observed ratio of Reads and Writes.

To place our work in historical context, we have identified three dimensions that describe previous work in avoiding bank conflicts, and we have explored this space to produce a single state-of-the-art solution that we refer to as the memoryless scheduler. We use this memoryless scheduler as a baseline to compare against.

In the context of the IBM Power5+, we have found that a history length of two is surprisingly effective. Thus, while our solution might appear to be complex, it is actually quite inexpensive, increasing the Power5+'s transistor count by only 0.038%. We evaluate the performance advantage of our technique using three benchmark suites. For SMT workloads consisting of the Stream benchmarks, our scheduler improves IPC by 55.6% over in-order scheduling and 16.0% over memoryless scheduling. For the NAS benchmarks, again with SMT workloads, the improvements are

25.6% over in-order scheduling and 9.7% over memoryless scheduling. For a set of commercial SMT workloads, the improvements are 51.6% over in-order scheduling and 7.5% over memoryless scheduling.

To explain our results, we have looked inside the memory system to provide insights about how our solution changes the various bottlenecks within the system. We find that an internal bottleneck at the CAQ is useful because it gives the scheduler more operations to choose from when scheduling operations. We have also explored the effects of varying parameters of the processor, the DRAM and the memory controller itself. We find that as memory traffic increases, the benefits of the AHB scheduler increase, even for multi-threaded workloads. We find that our solution is more robust than memoryless scheduling in the sense that our solution is less sensitive to changes in design parameters. We also find that the AHB scheduler is typically superior to the memoryless scheduler even when the latter is given additional hardware resources.

Chapter 4

Improving Memory Latency of Irregular Applications

Numerous hardware solutions have been proposed to hide long memory latencies. Early prefetching techniques [34, 65, 55, 2, 19] focused on exploiting streaming workloads. While regular forms of spatial locality are easy to predict, it has traditionally been difficult to exploit irregular patterns of spatial locality and even more difficult to exploit low amounts of spatial locality.

Recently, a class of aggressive prefetching techniques has arisen from the notion of a Spatial Locality Detection Table [32]. These techniques track accesses to regions of memory so that spatially correlated data can be prefetched together [32, 39, 9, 44, 67]. The chief advantage of these techniques is their ability to exploit irregular forms of spatial locality. Their chief disadvantage is their reliance on large tables that occupy chip area and consume power.

We propose a new solution, which uses a simple technique to augment the effectiveness of stream prefetchers. Our technique is based on two observations. First, memory intensive workloads with low amounts of spatial locality are likely to still contain many very short "streams," if "stream" can be defined to be as short as two consecutive cache lines. Second, stream prefetchers could effectively prefetch these short streams if they only knew when to be aggressive.

To understand this second point, recall that stream prefetchers look for accesses to k consecutive cache lines, at which point the $k+1^{st}$ cache line is prefetched; prefetching continues until a useless prefetch is detected. Thus, the value of k determines the prefetcher's aggressiveness, and this value is typically fixed at design time. Even with a small value of k, stream-based prefetchers do not fare well on short streams because they stop after a useless prefetch. For example, on a workload in which every stream is of length 2, a $k = 1$ policy would successfully prefetch the second cache line of each stream, but each successful prefetch would be followed by a useless prefetch, so 50% of its prefetches would be useless.

Our solution, *Adaptive Stream Detection*, guides the aggressiveness of the prefetch policy based on the workload's observed amount of spatial locality, as measured by a *Stream Length Histogram (SLH)*. An *SLH* is a dynamically computed histogram that attributes each memory access to a particular stream length. For example, if the *SLH* indicates that 70% of the memory requests were parts of streams of length 2 and that 30% of the memory requests were parts of streams of length 1, then an effective strategy would always prefetch the second cache line of a stream but never the third line. Thus, Adaptive Stream Detection can predict when to stop prefetching without incurring a useless prefetch. To adapt to changes in phase behavior, new Stream Length Histograms are computed periodically.

Adaptive Stream Detection provides two benefits. (1) It extends the notion of a stream to include streams as short as two cache lines. Thus, while it is inherently a stream-based approach, it provides benefits for workloads, such as commercial applications, that are not traditionally viewed as stream-based. (2) Because it is stream-based, it has low hardware costs, using small tables that have low static power leakage.

This chapter describes how Adaptive Stream Detection can be implemented in the memory controller. In this context, we introduce a second idea, Adaptive Scheduling, that adjusts the priority of prefetched commands based on the measured frequency of conflicts that prefetched commands have caused. This adaptivity is useful because any fixed priority may be excessively conservative for some workloads.

In this chapter we make the following contributions:

- We introduce Adaptive Stream Detection, a probabilistic prefetching technique that adjusts the aggressiveness of stream prefetching based on Stream Length Histograms, which are inexpensive to gather. This technique addresses the question of *what* to prefetch.

- We use the idea of Adaptive Stream Detection to design a prefetcher that resides in the memory controller and prefetches from DRAM into a small Prefetch Buffer. This prefetcher uses Adaptive Scheduling to modulate the relative priority of prefetch commands to regular commands. We show that a prefetch buffer that holds 16 cache lines is effective. We also see that this memory-side prefetcher (MS) complements the IBM Power5+'s existing stream prefetcher (PS), which performs processor-side prefetching.

- We evaluate Adaptive Stream Detection using the SPEC2006 floating point suite, the NAS benchmarks, and a set of five commercial benchmarks. For single threaded workloads, when we compare our technique to a stripped down Power5+ with no prefetching (NP), we improve the performance of the SPEC2006fp, NAS, and commercial benchmarks by 14.6%, 11.7%, and 9.3%, respectively. When MS is combined with PS, forming PMS, its improvements over NP are 32.7%, 24.2%, and 15.1%, respectively. The performance improvements for the commercial benchmarks are noteworthy because these benchmarks exhibit low amounts of spatial locality. We get similar results for SMT workloads.

- We evaluate the energy and power impact of our approach. For our three benchmark suites, we find that DRAM power consumption increases by 2.7%, 1.6%, and 2.8%, respectively, while DRAM energy consumption decreases by 9.8%, 7.9%, and 8.2%, respectively. For the four SPEC2006fp benchmarks that have low memory bandwidth requirements, the DRAM power impact is negligible: DRAM power increases by an average of 0.12%, while energy consumption decreases by 0.47%.

- We evaluate Adaptive Scheduling and show that it improves upon a set of conservative fixed-priority policies by about 2.9%.

In the next sections we describe our solution; we present empirical evaluation of our approach; and finally we summarize and provide concluding remarks.

4.1 Memory Prefetching Using Adaptive Stream Detection

This section describes our new prefetcher [30], which resides in the memory controller. This prefetcher addresses two major questions: (1) How can we reduce the number of unnecessary prefetch requests? (2) How can we reduce the opportunity cost of prefetches? Adaptive Stream Detection addresses the first issue, and Adaptive Scheduling addresses the second. To provide context, we first explain the basic idea behind Adaptive Stream Detection. After describing the mathematical details of how *SLH*'s are used, we discuss implementation issues, and present the organization of our prefetcher. Finally, we present details of Adaptive Scheduling.

4.1.1 Adaptive Stream Detection

Adaptive Stream Detection uses Stream Length Histograms, SLH, to capture spatial locality and guide prefetch decisions. For example, Figure 4.1 shows an SLH for one epoch of the *GemsFDTD* benchmark from the SPEC2006 suite. In an SLH, the height of the bar at location m represents the percentage of streams that have length m. Depending on the detected stream length of the current Read request, the prefetcher checks the SLH and determines how many, if any, sequential cache lines to prefetch.

In the example SLH of Figure 4.1, we see that 21.8% of all streams are of length 1, 43.7% of all stream are of length 2, etc. The rightmost bar indicates that 1.2% of all streams are length 16 or more. Given this information, when a Read request, R_n, arrives and is the first element of a new stream, a prefetch request should be issued because R_n is more likely to be the first element of a stream of length 2 or longer (78.2% probability) than to be part of a stream of length 1 (21.8%). On the other hand, if a Read request, R_n, is the second element of a stream, a prefetch should *not* be issued because there is a 43.7% probability that R_n is the second element of a stream of length 2, which is greater than the 34.5% likelihood that it is the second element of a longer stream. With similar reasoning, prefetches should be issued for any Read request whose current stream length is 3 or greater than 6. This example shows that the use of the SLH allows a prefetcher to make rather sophisticated prefetching decisions based on the length of an individual stream.

The prefetcher can also use the SLH to decide whether to generate multiple prefetches—although we do not evaluate this idea. For example, when R_n is part of a stream of length 1, the prefetcher decides whether to generate two consecutive prefetches by adding the probabilities of the first two bars and comparing the sum with the rest of the histogram. If the sum of the first two bars is less than the sum

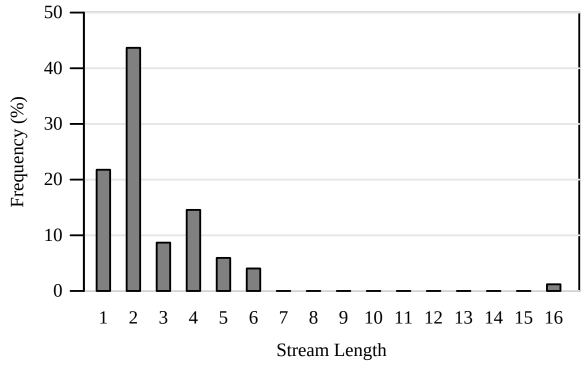

Figure 4.1: Stream Length Histogram (*SLH*) for an arbitrary epoch of the GemsFDTD benchmark.

of the other bars, and if the prefetcher has already decided to prefetch one line, it generates a prefetch for the second line as well.

Because memory access behavior typically varies over time, our solution periodically creates an SLH after every e Read requests, where e is known as an epoch. Thus, in every epoch, our method constructs a new SLH for use in the next epoch. Figure 4.2 shows how epochs can vary widely over time. To keep track of increasing or decreasing streams, we need one SLH for each direction.

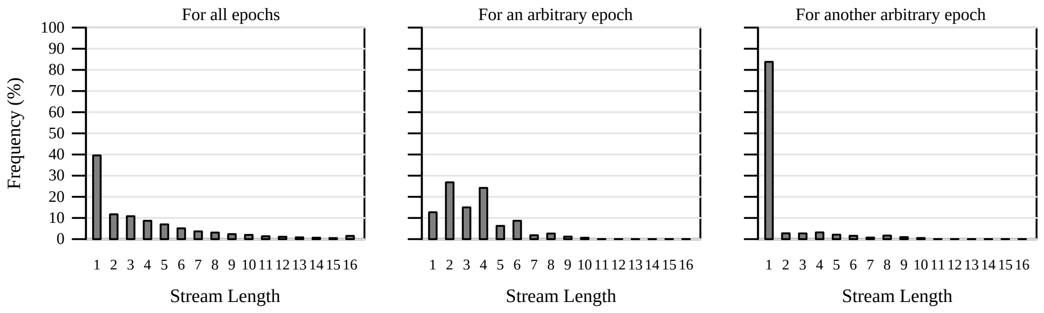

Figure 4.2: Stream Length Histograms (*SLH*) for the GemsFDTD benchmark from the SPEC2006fp suite show that the *SLH*'s vary widely at different points in time. Here the epoch length is 2000 reads.

4.1.2 Using the SLH to Detect Locality

Our probabilistic approach to prefetching makes decisions by comparing the likelihood that a Read request will be the last element of a stream against the likelihood that it will be part of a longer stream. In this subsection, we derive inequalities that guide these prefetch decisions. Our discussion also establishes the transition from the SLH concept to its implementation that we present later in Section 4.1.4.

Definitions. To describe our method, we define two functions, $lht()$ and $P()$, which can be used to compute an SLH, as follows:

$lht(i)$: the number of streams of length i or longer, where $1 \leq i \leq fs$ and fs is the maximum stream length that our method uses. For any $i > fs$, $lht(i) = 0$.

$P(i, j)$: the sum of probabilities that a Read is part of any stream of length k, where $i \leq k \leq j$ and $1 \leq i, j \leq fs$. We can define $P(i, j)$ in terms of $lht()$ as follows:

$$P(i,j) = (lht(i) - lht(j+1))/lht(1) \tag{4.1}$$

The value of the i^{th} bar of an SLH equals $P(i, i)$.

Prefetch Decision. To determine whether to issue a prefetch, we check whether the following condition is satisfied for a Read request, R_n, that is the i^{th} element of a stream:

$$P(i,i) < P(i+1, fs) \tag{4.2}$$

This inequality states that the probability that the most recent Read request, R_n, is the last element of a stream of length i is smaller than it being the i^{th} element of a stream of length longer than i. We can simplify the inequality (4.2) as follows:

$$P(i,i) < P(i+1, fs) \tag{4.3}$$

$$\equiv \frac{lht(i) - lht(i+1)}{lht(1)} < \frac{lht(i+1) - lht(fs+1)}{lht(1)} \tag{4.4}$$

$$\equiv lht(i) < 2 \times lht(i+1) \tag{4.5}$$

Our technique uses the inequality (4.5) to make next line prefetch decisions. We provide, without proof, a generalized version of (4.5) to prefetch k consecutive lines after R_n:

$$lht(i) < 2 \times lht(i+k) \tag{4.6}$$

4.1.3 Prefetcher Design

The organization of our prefetcher is shown in Figure 4.3, where the gray boxes represent our additions to the memory controller. Read commands enter the memory controller and are sent to both the original memory controller and to the Stream Filter. The Stream Filter keeps track of Read streams and generates the SLH. This information from the Stream Filter is then fed to the Prefetch Generator, which decides whether a prefetch command should be issued, and if so, places the prefetch command in the Low Priority Queue (LPQ), where the Final Scheduler can consider it, along with other commands in the LPQ and CAQ, when selecting commands to issue to DRAM. Any prefetched data are then stored in the Prefetch Buffer.

The Prefetch Buffer is checked twice. It is first checked before Read commands are placed in the CAQ, so that Read commands can be satisfied by the Prefetch Buffer, in which case the latency of going to DRAM is saved and the Read command is squashed. The Prefetch Buffer is checked again when the Final Scheduler selects a Read command from the CAQ to send to memory; this check is useful because the desired data may have arrived in the Prefetch Buffer while the Read command was resident in the CAQ.

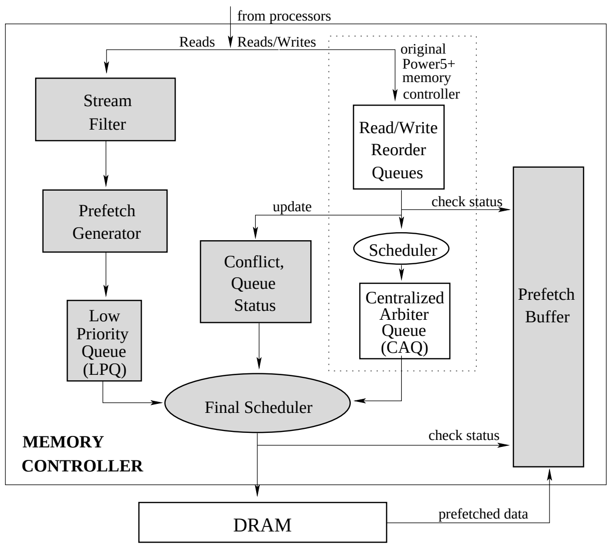

Figure 4.3: Overview of our prefetcher.

Stream Filter. To maintain information about Read streams, the Stream Filter uses one *slot* to track each Read stream. Each slot maintains (1) the last address accessed for this stream, (2) the length of the stream, (3) the stream's direction, and (4) the stream's *lifetime*, which indicates when the stream should be evicted. These slots are used as follows:

- If the Read, R_n, is not part of a stream and if there is a vacant slot in the Prefetch Filter, the last access field is set to the address of the Read request, the length field is initialized to 1, the lifetime is initialized to a predetermined value, and the direction is set to Positive.

- If R_n is not part of a stream and there is no available slot, no prefetch will be generated after R_n, but the *SLH* structure is updated as if a stream of length

1 had been detected.

- If R_n is the most recent element of a previously detected stream, the stream length is incremented by 1, the last access is set to the address of R_n, and the lifetime of the stream is incremented by a predetermined value.
- The direction of the stream is set to Negative if the length of the previous stream is 1 and the address of R_n is smaller than the last address of the stream.
- At every processor cycle, the lifetime fields are decremented by one. A stream is evicted from a slot when its lifetime expires. At this point, the SLH structure is updated using the length value in the Stream Filter.
- At the end of each epoch, all streams are evicted from the Stream Filter.

Prefetch Buffer. The Prefetch Buffer holds data that are fetched from memory by the memory-side prefetcher. We assume that this buffer is a set associative cache with an LRU replacement policy. When there is a write request to an address in the Prefetch Buffer, we invalidate the entry in the buffer. We also invalidate the entry if a regular Read request matches the address, because in such cases the data will likely be moved to the L1 or L2 cache, so it is unlikely to be useful in the Prefetch Buffer again.

4.1.4 Implementation of Adaptive Stream Detection

We now present details for implementing Adaptive Stream Detection. For simplicity, we restrict our explanation to streams with increasing addresses only, and we only discuss prefetching for one cache line. It is straightforward to generalize this approach to streams with decreasing addresses and multiple line prefetching.

Rather than implement the *SLH* explicitly, we construct the information in the *SLH* using two tables of length fs. These *Likelihood Tables*, *LHTcurr* and *LHTnext*, correspond to the *lht()* function discussed previously. A given epoch uses and updates information from LHTcurr and gathers information for the start of the next epoch in LHTnext. LHTnext is updated using the information from the Stream Filter. When an entry of length k in the Stream Filter is invalidated, LHTnext[i] is incremented by 1, for all i, where $1 \leq i \leq k$. At the end of an epoch, LHTnext is modified using the remaining valid entries in the Stream Filter; the contents of LHTnext are moved to LHTcurr; and LHTnext is re-initialized. Each entry of the tables is a $\log_2(m)$ bit counter, where m is the maximum epoch length.

LHTcurr is used to make prefetch decisions for the current epoch. This table has one comparator for each pair of consecutive table entries, *i.e.*, LHTcurr[i] and LHTcurr[i+1], for $1 \leq i < fs$. At the beginning of an epoch, the contents of LHTcurr are used to construct the *SLH*. As the epoch progresses, this information is modified using the observed stream lengths of the current epoch. When an entry of length k in the Stream Filter is invalidated, the value of LHTcurr[i] is decremented by 1, for all i, where $1 \leq i \leq k$.

When the Stream Filter observes that a Read request is part of a stream of length k, prefetch requests are generated using the output of the comparison of LHTcurr[k] and LHTcurr[k+1], as in inequality (4.5). Instead of multiplying LHTcurr[k+1] by 2, for any k, the comparator for the (LHTcurr[k], LHTcurr[k+1]) pair takes the left shifted value of LHTcurr[k+1] as input.

4.1.5 Adaptive Scheduling

Clearly, speculative prefetch commands should be given lower priority than regular commands. But because memory systems are becoming increasingly complex, and because the Final Scheduler must make decisions whose effects may not be seen until the future, it is not obvious what policy provides the best performance. For example, a conservative policy that always gives prefetch commands lower priority than regular commands may unnecessarily block prefetch commands behind regular commands that cannot issue due to conflicts in the memory system. Thus, rather than dictate a particular policy at design time, Adaptive Scheduling uses feedback to dynamically select from one of five policies in order of decreasing conservativeness: Only issue a command from the LPQ (1) if the CAQ is empty and the Reorder Queues are empty, (2) if the CAQ is empty and the Reorder queues have no issuable commands, (3) if the CAQ is empty, (4) if the CAQ has at most 1 entry and the LPQ is full, (5) if the first LPQ entry has an earlier timestamp than the first CAQ entry.

To choose from among these policies, the memory controller tracks the number of times that a regular command in the Reorder Queues cannot proceed to the CAQ because it conflicts in the memory system with a previously issued prefetch command. As the occurrences of these conflicts grows (or shrinks), the policy becomes more (or less) conservative. The policy is adjusted using the same epoch size that is used to compute Stream Length Histograms. Thus, this approach determines the priority of prefetch commands based on a measure of memory system performance, rather than on some instantaneous property such as occupancy of a queue.

4.2 Experimental Results

We evaluate Adaptive Stream Detection along several dimensions. We present overall performance and power results for all three benchmark suites. We then use a subset of the benchmarks to illustrate additional points, choosing the two best-case and the two worst-case benchmarks—in terms of PMS performance improvement—from the SPEC and commercial benchmarks.

4.2.1 Hardware Costs

We evaluate a prefetcher that is configured as follows: Each thread has a Stream Filter with 8 slots and LHTnext and LHTcurr tables that each hold 16 entries. Because streams are tracked in both the positive and negative directions, LHTnext and LHTcurr each require 32 counters per thread. In addition to these per-thread resources, the prefetcher has one 16 entry Prefetch Buffer (2KB) and an LPQ with the same number of entries—3—as the CAQ. The current Power5+ memory controller occupies about 1.61% of the entire chip area, with the dominant portion of the memory controller being control logic. Our extensions to the memory controller increase the area of the memory controller by about 6.08%, resulting in a 0.098% increase in the total chip area.

4.2.2 Benchmark Results

We now compare simulation results for four configurations: no-prefetching (NP), processor-side prefetching only (PS), memory-side prefetching only (MS), and processor- and memory-side prefetching together (PMS). In PMS, only the memory-side prefetcher uses Adaptive Stream Detection. In the following graphs, we present three different comparisons: (1) PMS vs. NP (2) MS vs. NP, and (3) PMS vs. PS.

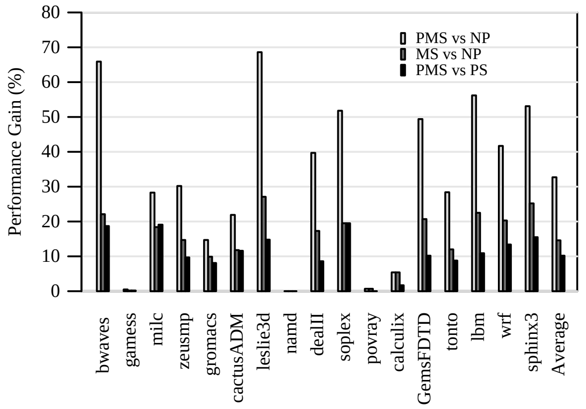

Figure 4.4: Performance improvements for the SPEC2006fp Benchmarks.

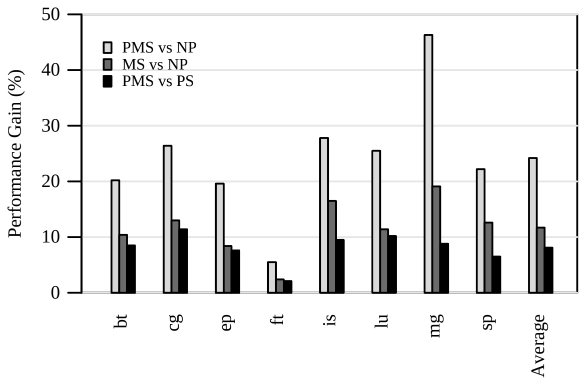

Figure 4.5: Performance improvements for the NAS Benchmarks.

We see that the PMS configuration performs best, and the benefits from memory-side and processor-side prefetching are largely complementary but not completely orthogonal.

For the SPEC2006fp benchmarks (Figure 4.4), we find that the performance benefit of PMS over NP is between 0-68.6%, with an average of 32.7%. MS improves performance over NP by an average of 14.6%, and PMS improves over PS by an average of 10.2%. For the NAS benchmarks (Figure 4.5), the PMS approach sees an average improvement of 24.2% over NP and 8.1% over PS. For the commercial benchmarks (Figure 4.6), the PMS approach sees an average improvement of 15.1% over NP and 8.4% over PS.

SMT Results. We have repeated the above experiments on a system that uses two SMT threads on the same processor. For these experiments, we leave the Prefetch Buffer size (16 cache lines) unchanged, but we double the size of the Stream Filter and the number of LHT tables, so that each thread can track its own set of streams. We find that SMT performance improvements are about the same as the single-threaded results. For example, PMS improves performance over PS by 10.7%, 9.2%, and 7.5%, respectively, for the SPEC2006fp, NAS, and commercial benchmarks. The improvements for PMS over NP are 28.5%, 20.4%, and 11.1%, respectively.

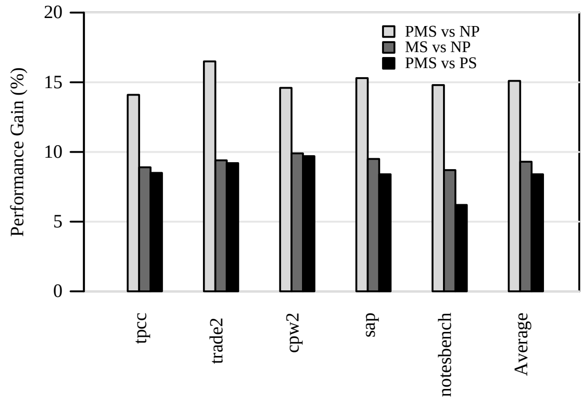

Figure 4.6: Performance improvements for the commercial benchmarks.

We find it critical to replicate the locality identification hardware—in our case the Stream Filter—for each thread. For our solution, this hardware is small, as opposed to many other solutions [44, 9, 67] for which large tables would have to be replicated.

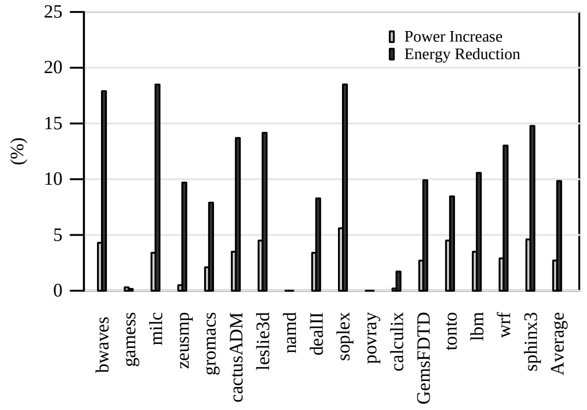

Figure 4.7: DRAM Power and Energy comparison for the SPEC2006fp benchmarks.

Power and Energy Effects. In Figures 4.7, 4.8, and 4.9, we compare PMS to PS in terms of DRAM power usage and energy consumption. We find that PMS increases power consumption, on the average, by 2.7%, 1.6%, and 2.8% for SPEC2006fp, NAS, and commercial benchmarks, respectively. For the same bench-

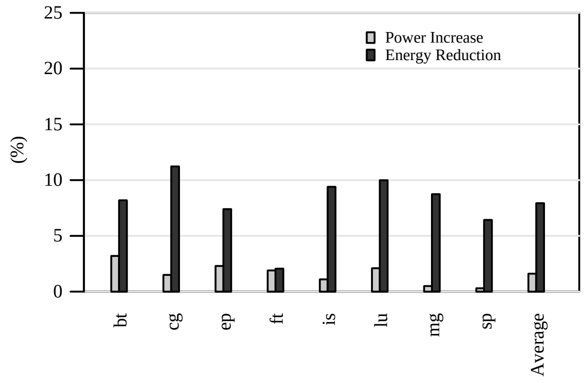

Figure 4.8: DRAM Power and Energy comparison for the NAS benchmarks.

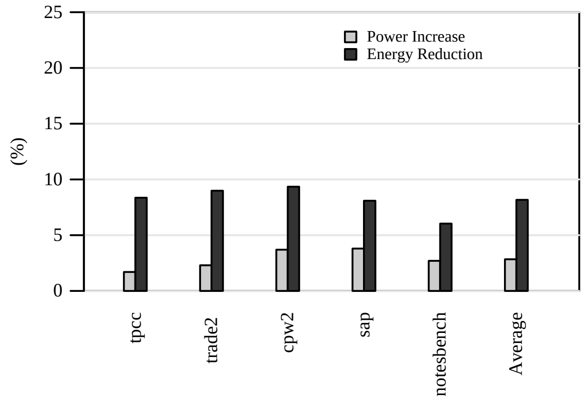

Figure 4.9: DRAM Power and Energy comparison for the commercial benchmarks.

marks, PMS reduces energy consumption by 9.8%, 7.9%, and 8.2%. For the four benchmarks that are not memory intensive—*gamess*, *namd*, *povray*, and *calculix*—the power increase is negligible. Again, for SMT workloads, the DRAM power and energy results are similar to the single threaded case.

Other Power Costs. Of course, the implementation of the prefetcher itself also consumes power. We do not have benchmark-specific analyses of this power usage, but an analysis of the Power5+ chip and an area-based estimation of the MS prefetcher provides the following figures. The memory controller on the Power5+ consumes about 1% of the chip's power. The MS prefetcher increases the power of the memory controller by approximately 6%, which is 0.06% of the chip's total power. As a reference, the Power5+ chip typically consumes roughly four times the power as the DRAM chips for our workloads.

By contrast, if we were to add a 64KB table for detecting spatial locality, as suggested by other approaches, we would add four such tables—one for each thread—for the Power5+. We believe that each 64KB table would consume up to 25% of the power of a 64KB L1 I-cache (Loads constitute roughly 25% of all instructions), which for the Power5+ is about 0.6% of the chip's power. To support four such tables would increase the chip's active power by about 2.4%. Moreover, as leakage power becomes more important to future systems, the power effects of large tables will become more significant.

4.2.3 Detailed Results

Importance of Adaptive Stream Detection and Adaptive Scheduling. Figure 4.10 shows that both Adaptive Stream Detection (ASD) and Adaptive Scheduling contribute to performance gain. In this figure, the first bars in each cluster represent normalized execution times for our PMS approach. The next five bars compare the PMS against the five scheduling policies that we discussed in Section 4.1.5. We see that the Adaptive Scheduling improves performance upon these fixed policies between 2.3% and 3.6%. We conclude that the impact of Adaptive Stream Detection is much more significant than that of Adaptive Scheduling.

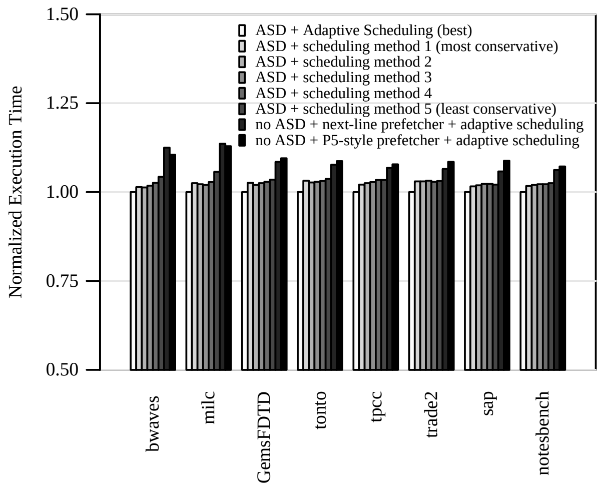

Figure 4.10: Impact of Adaptive Stream Detection and Adaptive Scheduling.

Figure 4.10 also provides a head-to-head comparison of Adaptive Stream Detection against both next-line prefetching (second bar from the right) and the Power5+'s processor-side prefetcher (rightmost bar) when all are implemented in the memory controller. We see that Adaptive Stream Detection provides performance that is 8.4% better than the next-line prefetcher. Somewhat surprisingly, in this context the Power5-style prefetcher yields worse performance than the next-line prefetcher.

Figure 4.11 shows that a significant portion of streams are of length five or shorter. These short streams are where Adaptive Stream Detection sees the most benefit. A next-line prefetcher generates useless prefetches for all streams of length one, and we see that the percentage of such streams is quite high for these benchmarks. There is also a significant number of streams of length 2-5, which is where a Power5-style stream-based prefetcher sees the worst performance: For these streams the useless prefetch that it issues before detecting the end of a stream represents a non-trivial fraction of the total prefetches. Finally, observe that even the four commercial benchmarks, which have poor spatial locality, have a significant percentage of streams of length 2-5: roughly 37% for *tpc-c*, 49% for *trade2*, 40% for *sap*, and 62% for *notesbench*. These percentages help explain why Adaptive Stream Detection is beneficial even for workloads with low spatial locality.

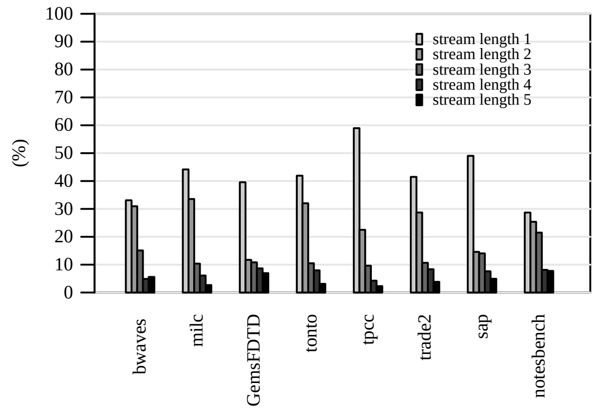

Figure 4.11: Stream Length Histograms of eight benchmarks. Streams of lengths between 1 and 5 constitute 78–96% of all streams.

Prefetch Efficiency. Figure 4.12 presents three measures of the effectiveness of Adaptive Stream Detection: (1) the percent of useful prefetches, (2) the prefetch coverage, that is, the percent of Read commands (including processor-side prefetches) that get its data from the Prefetch Buffer, and (3) the percentage of the regular memory commands—both Reads and Writes—that are delayed because of memory-side prefetches. These values pertain only to prefetches generated by the memory-side prefetcher, not the processor-side prefetcher. We see that the percentage of useful prefetches is between 82% and 91%. The coverage is between 19% and 34%, and only 1-3% of regular commands are delayed by the memory-side prefetch commands.

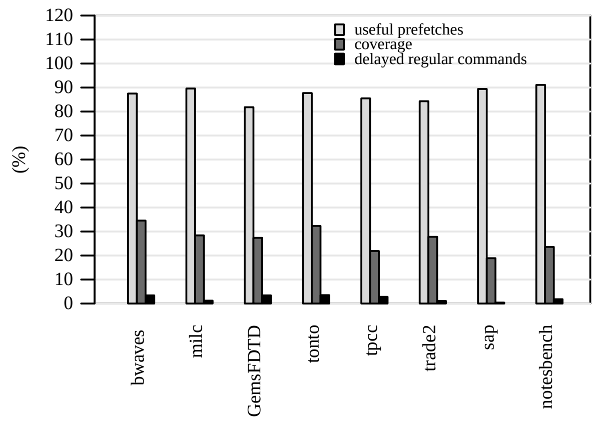

Figure 4.12: Effectiveness of our prefetching approach.

Sensitivity to Prefetch Buffer and Stream Filter Size. Figures 4.13 and 4.14 show, for our PMS approach, the performance effect of the size of the Prefetch Buffer and Stream Filter. In our simulations, we use a configuration with a 16-block prefetch buffer and an 8-entry stream filter. We find that increasing the size of the Prefetch Buffer or Stream Filter beyond this configuration improves performance but with diminishing returns.

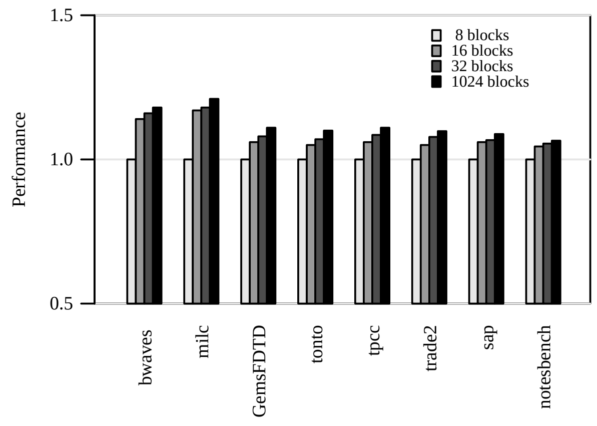

Figure 4.13: Sensitivity of PMS to prefetch buffer size.

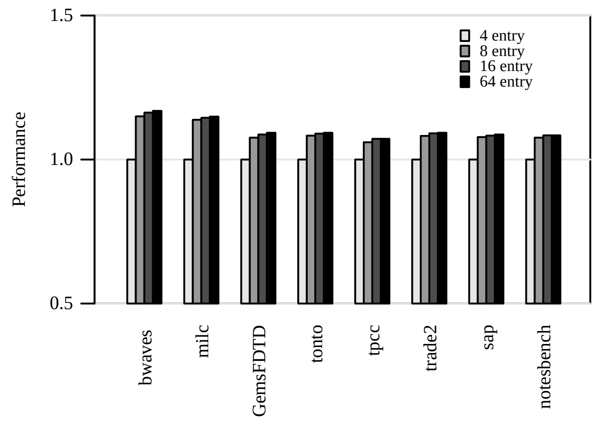

Figure 4.14: Sensitivity of PMS to stream filter size.

Further Improvement Opportunities for Latency Hiding. Figure 4.15 compares our prefetching approach to a perfect memory-side prefetcher. We assume that the perfect prefetcher can predict what to prefetch and when to issue prefetch requests such that x% of all Read requests find their data in the prefetch buffer, and no memory commands are delayed because of the prefetch requests. We analyze the relationship between our ASD prefetcher and the perfect prefetcher by varying x between 0% and 100%, where x=100% represents the *ideal* memory-side prefetcher.

In Figure 4.15, we see that for all benchmarks, the performance improvement of the ASD prefetcher is below the perfect prefetcher curve and it is far from the ideal prefetcher. In other words, although our prefetching approach improves performance significantly, it does not eliminate the memory latency problem completely. For example, for the *GemsFDTD* benchmark, the ASD prefetcher has a coverage of 32.4% and improves performance by 10.2%. However, for the same benchmark, the ideal memory-side prefetcher improves performance by 38.9%. The ASD prefetcher achieves, on average, 21.3%, 24.6%, and 18.7% of the coverage, and 17.4%, 20.9%, and 14.1% of the performance improvement of the ideal prefetcher for the SPEC2006fp, NAS, and commercial benchmarks, respectively.

There are three possible ways to make the performance of our prefetching

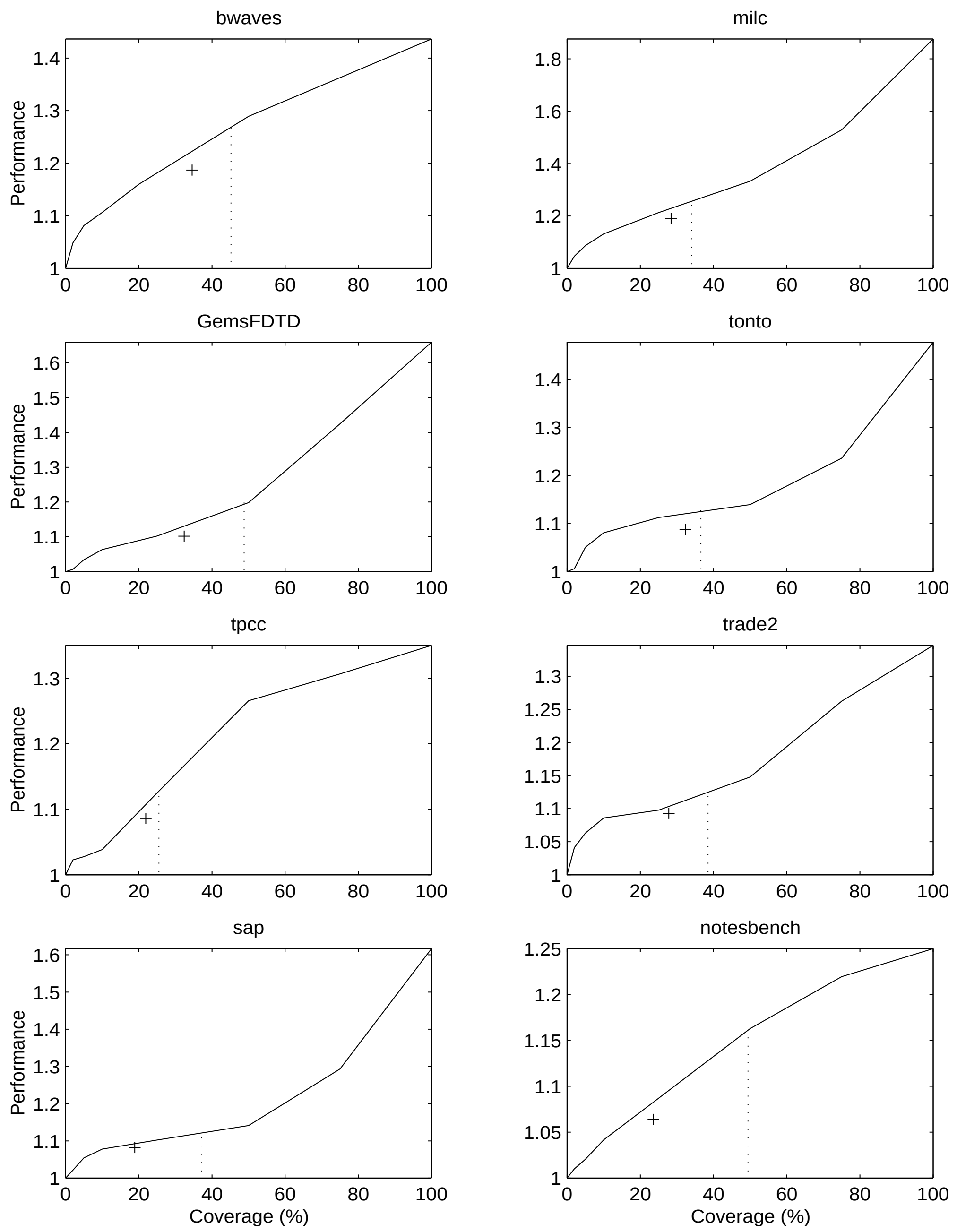

Figure 4.15: Performance effects of coverage rate. Solid line represents the perfect prefetcher, "+" represents our ASD prefetcher, dotted line is for the maximum coverage that a memory-side prefetcher can achieve without prefetching the first elements of streams, and 100% coverage corresponds to the ideal prefetcher.

method closer to the ideal prefetcher. First, we can try to increase available memory bandwidth and/or to improve the Adaptive Scheduling technique further, so that side effects of prefetch requests over regular memory commands are diminished. Reducing side effects moves the performance point ("+" sign) of our prefetcher, in Figure 4.15, upwards. Second, to move the performance point to the right, that is to increase coverage, we can attempt to improve (including capacity increases for the stream filter and prefetch buffer) the Adaptive Stream Detection method. The current ASD approach does not prefetch first elements of streams. Therefore, for the benchmarks in Figure 4.15, the maximum coverage we can get (dotted vertical line) is the percentage of the non-first elements of streams, which is between 25.7% and 49.4% of which we achieve 18.9-34.5%. Note that to obtain the maximum possible performance (top point of the dotted line), a prefetching mechanism needs to be supported by increased memory bandwidth. Otherwise, coverage may increase at the expense of increased bandwidth requirements, which may or may not result improved performance. Finally, the third option to improve performance is to develop hardware and/or software techniques to prefetch the first elements of streams. Because, any coverage rate to the right of the dotted line in Figure 4.15 requires prefetching of the first elements of streams, which constitute a significant portion (50.6-74.5%) of all Read requests.

Our focus in this dissertation has been to hide the latency between the memory controller and DRAM. Reducing latency inside the processor is beyond the scope of this study, and we leave it as a future work.

Accurately Constructing Frequency Histograms. The success of Adaptive Stream Detection depends on the accuracy of the computed Stream Length Histograms, which are computed using the Stream Filter. Because the Stream Filters have finite size, the computed SLH is actually an approximation of a complete SLH. We have found that this approximation of the SLH closely matches the actual *SLH*, as shown in Figure 4.16, which is a sample epoch in the *GemsFDTD* benchmark.

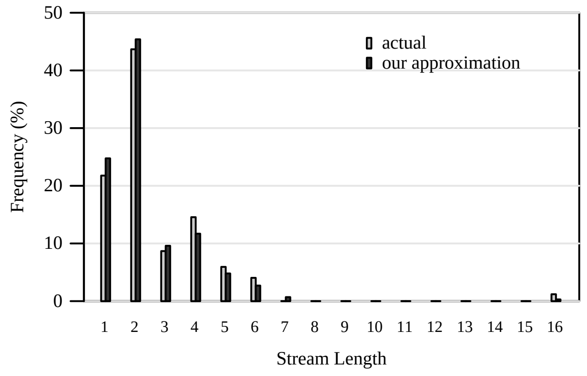

Figure 4.16: Accuracy of calculating Stream Length Histograms.

Interaction with the Memory Scheduler. The impact of a prefetcher can be sensitive to the choice of memory scheduler that is used. For the results presented in this chapter, we use the Adaptive History-Based memory scheduler (AHB), but to investigate the interaction between memory scheduling algorithms and our new prefetching technique, we also study two less sophisticated memory schedulers, *in-order* and *memoryless*, which provide reduced DRAM bandwidth compared to the AHB scheduler. When a simple in-order scheduler is used, the performance gain of our prefetcher is reduced by about 5%. For the better memoryless scheduler, the performance gain of our prefetcher is reduced by about 1%. These results indicate that the benefit of our prefetching approach increases as other bottlenecks in the memory subsystem are reduced.

We also find that our adaptive history-based memory scheduling approach and the new prefetching method that we have introduced complements each other. When compared with a system where neither of these two improvements exist, i.e. with memoryless scheduling and without any memory-side prefetching, combined implementation of our two techniques improves performance of the SPEC2006fp, NAS, and the commercial benchmarks by 14.3%, 13.7%, and 11.2%, respectively.

4.3 Summary

We have introduced a new stream-based prefetching technique that is effective for streams of any length, including extremely short streams. The key idea is to monitor the amount of spatial locality in a program's execution to adjust the aggressiveness of a basic stream prefetcher. By capturing such spatial locality in a Stream Length Histogram, our prefetcher can probabilistically decide when to start and stop prefetching based on the recently observed behavior. A secondary contribution is the notion of Adaptive Scheduling, which adapts the aggressiveness of the prefetcher based on the observed number of conflicts between prefetch commands and regular commands. Previous techniques [43] have monitored specific aspects of the memory system, but we show that such fixed policies can be overly conservative.

Using extremely accurate simulators for a modern microprocessor and its memory system, we have shown that Adaptive Stream Detection and Adaptive Scheduling provide significant performance improvements, even for commercial workloads that have low spatial locality. This solution also has low DRAM power costs and modestly improves DRAM energy consumption. If implemented in the Power5+, our solution increases the area of the chip by less than 0.1%. Compared to other prefetching strategies, the hardware cost of our approach is minimal. Moreover, because its spatial locality detection component is small, the cost advantage of Adaptive Stream Detection improves—relative to other approaches that require large tables—as the number of hardware threads increases.

Chapter 5

DRAM Power Optimizations

In the previous two chapters we developed techniques with small modifications to the memory controller to improve memory bandwidth and memory latency. Because power is now a first order concern, and because DRAM can consume up to 45% of a system's power [42], it's natural to ask whether memory controllers can improve power utilization, as well. In particular, there are two possible goals with respect to power: (1) maximize performance for a given power threshold; (2) achieve good energy efficiency. This second goal is important for large servers where energy efficiency translates into lower energy bills. This second goal is difficult because it requires us to consider the tradeoffs between power reduction and performance reduction. In this chapter, we present and evaluate new techniques for managing both aspects of DRAM power. We assume that the DRAM supports a *power-down* command, which puts a portion of the DRAM into a low-power mode, which can be found on today's DRAM's.

A basic mechanism for reducing power is to put memory devices into a low-power mode when they are idle. Unfortunately, the overuse of this mechanism can limit performance, as there are associated entrance and exit latencies for a particular low power mode. An intelligent memory scheduler would seem to be a natural partner with these low power modes, but the scheduling goal of low power and good performance are at odds. For good performance, the scheduler typically selects commands that avoid hardware conflicts, essentially spreading the commands across many physical memory devices. However, to reduce power consumption, the scheduler would like to cluster commands to a subset of the physical devices, allowing one or more of them to be put into low-power mode.

In this chapter we study three aspects of the solution space. First, we study the benefit of powering-down portions of the DRAM when they become idle and powering them back up on demand. Second, we study the impact of modifying the memory scheduler so that it issues commands in response to the state of the DRAM, that is, with cognizance of the powered-down ranks. This modified memory scheduler is a natural extension of our previously studied adaptive history-based (AHB) memory scheduler. Finally, given a power budget, we develop a throttling method to accurately estimate the length of time during which commands should be blocked in the reorder queues, allowing DRAM ranks to be powered-down.

This chapter makes the following contributions:

1. We present a power-down mechanism for the memory controller in the context of server-class memory systems.

2. We present simple modifications to the previously described adaptive history-based schedulers. These modifications optimize for power by clustering commands to the same rank to create rank locality, thereby increasing the periods during which other ranks can be powered down.

3. We evaluate our new Power-Aware AHB scheduler, along with three previously proposed memory schedulers. Our detailed simulators provide results for performance and energy efficiency, as well as for power consumption. We see that for the daxpy kernel, our new Power-Aware AHB scheduler reduces

DRAM power by 42.6% and improves performance by 53.5% when compared with a standard FIFO scheduler with no power-down mechanism. We find that our Power-Aware AHB improves the energy efficiency of the Stream and NAS benchmarks by a factor of 5. The simplicity and success of our modifications argue that the adaptive history-based scheduler provides a powerful framework for all aspects of memory scheduling.

4. We present a throttling approach that actively reduces DRAM power by blocking memory commands. The goal of this method is to estimate the throttling delay such that DRAM power consumption falls below a predetermined power budget and show that performance degradation is as small as possible.

In the next sections we describe our new solutions regarding DRAM power consumption, we present experimental results, and finally we conclude and summarize our work.

5.1 Power- and Performance-Aware Memory Controllers

This section describes our new approach to memory controller design, which makes the memory controller both power-aware and performance-aware. We present three additions to current memory controllers: a power-down unit to schedule rank power-down signals, an augmented form of adaptive history-based schedulers that includes power criteria, and a throttling mechanism to manage power requirements.

5.1.1 Power-Down Unit in the Memory Controller

The IBM Power5+ memory controller uses a command bus to transmit memory commands to DRAM. Every command on this bus has a command type and an address. We propose a new type of power-down command, in which the rank to be powered down is encoded in the address bits.

In the power-down unit of the memory controller, we maintain two extra components for each rank: a *rank-lowpower* bit and a counter. The rank-lowpower bit is set when the rank is in low power mode. The counter maintains the number of cycles remaining until the rank becomes idle. Each time a regular command (a Read or a Write) is sent to any bank of a powered-down rank, the rank's counter is initialized to the maximum of the current value and the latency of the new command.

The overuse of power-down commands can degrade performance in two ways. First, power-down commands consume command bus bandwidth. Second, there will be unnecessary switches between low and high power modes in DRAM, which will waste two DRAM cycles. Finally, in most modern DRAM chips, when a rank enters low power mode, it has to stay in that mode for a certain number of cycles. Thus, powering down a rank prematurely can increase the latency for memory commands waiting for the powered-down rank.

We now present a protocol to decide when to send a power-down command to DRAM. At every cycle, the power-down unit checks rank counters, rank-lowpower bits, and the commands waiting in the CAQ. A power-down command is sent to a rank that meets the following conditions: (1) The rank counter is zero, which indicates that the rank is idle. (2) The rank-lowpower bit is zero, because otherwise a new power-down command for the rank will be redundant and will unnecessarily occupy the command bus. (3) There is no command for the rank waiting in the CAQ; this condition avoids powering down a rank if a Read or Write to that rank is imminent. (4) The command at the front of the CAQ cannot be issued in this cycle. To reduce performance degradation, we give priority to regular commands over power-down commands.

The memory controller can send only one power-down command at any cycle, so at each cycle, the power-down unit checks for the above conditions starting at a random rank number. Randomization eliminates any bias in cases where more than one rank satisfies the power-down conditions.

5.1.2 Power-Aware Adaptive History-Based Schedulers

We now describe how the adaptive history-based memory schedulers can be adapted to include power information. As we described in Chapter 3, a history-based scheduler uses the history of recently scheduled memory commands when selecting the next memory command. In particular, scheduling goals are encoded in finite state machines. Previously, two scheduling goals were considered to improve performance: (1) minimize the latency of the scheduled command, and (2) match some desired balance of Reads and Writes. By scheduling commands to match an expected ratio of Reads and Writes, the scheduler avoids bottlenecks that arise from uneven Read and Write reorder queues.

We modify these AHB schedulers by adding power savings as a new goal. We do this by creating a state machine where power usage is the first optimization goal, which we describe below. Because both performance and power goals are important, we probabilistically combine the three FSM's to produce a scheduler that encodes all goals. The result is a history-based scheduler that is optimized for both performance and power, but for one particular mix of Read/Writes. To accommodate a wide variety of Read/Write mixes, we use adaptivity in the same sense as the original adaptive history-based scheduler, namely, our adaptive scheduler observes the recent command pattern and periodically chooses the most appropriate of three history-based schedulers.

Optimizing for Power

Our *Power-Aware History-Based scheduler* uses power as the first optimization criterion. The basic idea is to group commands for the same rank as closely as possible in the CAQ. This will reduce the number of power-down operations while providing the same amount of power savings. In the state machine for the scheduler, we define the priorities for each possible command in the reorder queues as follows: The set of commands to the same rank with the last command sent to the CAQ has the highest priority, the set of commands to the same rank with the second from the last command has the second priority, and so on. Since there may be more than one command in each of these sets, our approach breaks ties using performance as the second criterion. Algorithm 4 depicts this process.

Algorithm 4 power_scheduler(n)

```
// n is the history string size
 1: for all command sequences of size n do
 2:
 3:     for each possible next command do
 4:         Calculate priority with respect to power.
 5:     end for
 6:     Sort possible commands with respect to priorities.
 7:     for commands with equal priority in terms of power do
 8:         Use expected_latency to make decisions.
 9:     end for
10:     Sort possible commands with respect to expected_latency.
11:     for commands with equal power priority and expected_latency do
12:         Use Read/Write ratios to make decisions.
13:     end for
14:
15:     for each possible next command do
16:         Output the next state in the FSM.
17:     end for
18: end for
```

Combining State Machines Probabilistically

As with the original AHB scheduler, we probabilistically combine our multiple optimization goals to form a single history-base scheduler. Algorithm 5 weights each criterion and produces a probabilistic decision. At runtime, a random number is periodically generated to determine the rules for state transitions as follows:

Algorithm 5 probabilistic_scheduler

```
1: if random_number < threshold1 then
2:    command_pattern_scheduler
3: else
4:    if random_number < threshold2 then
5:       expected_latency_scheduler
6:    else
7:       power_scheduler
8:    end if
9: end if
```

The algorithm basically interleaves three state machines into one, periodically switching among the three in a probabilistic manner, where the threshold values are system-dependent and are determined experimentally.

5.2 Evaluation of the Power-Down Mechanism

To evaluate the effects of the power-down mechanism that we have introduced, we first present detailed results for the daxpy kernel. Then, for the Stream and NAS Benchmarks, we compare our Power-Aware AHB approach to the in-order, memoryless, and AHB schedulers. To measure performance, we use simulated execution time as our metric. To measure power, we use Watts as our metric. Finally, to measure efficiency, we use 1/Joules.

5.2.1 DAXPY Results

Figure 5.1 shows how three previously studied memory schedulers—in-order, memoryless, and adaptive history-based—compare in terms of power (left graph) and performance (right graph). We see that the more sophisticated schedulers provide better performance but at the expense of higher average power consumption.

Figure 5.2 compares the power and performance of these three schedulers when combined with our Power-Down mechanism. These results are all normalized

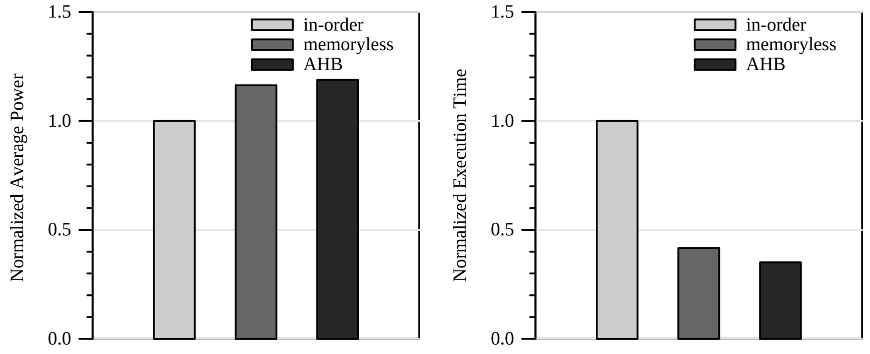

Figure 5.1: Left: Power consumption of Inorder, Memoryless, and Adaptive History-Based schedulers (without the Power-Down mechanism). Right: Performance of these three schedulers.

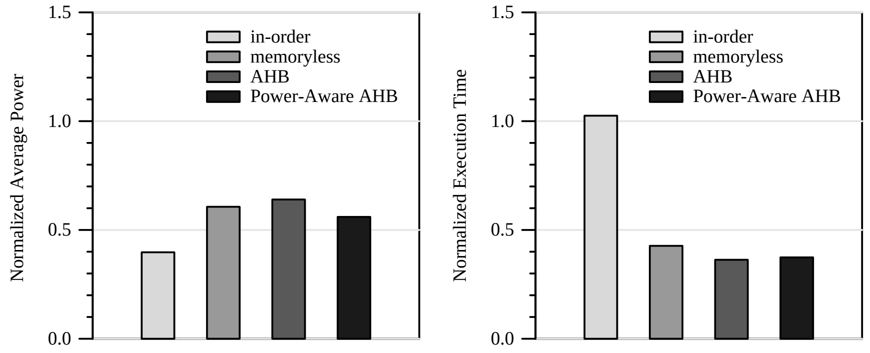

Figure 5.2: Left: Power consumption of Inorder, Memoryless, and Adaptive History-Based schedulers with the Power-Down mechanism. Right: Performance of these schedulers with the Power-Down mechanism.

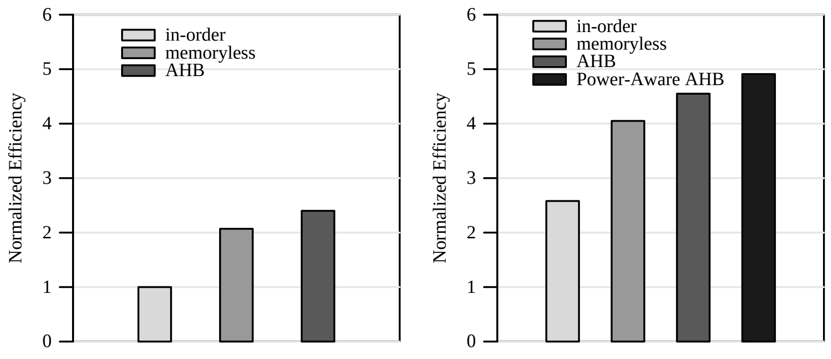

Figure 5.3: Efficiency Comparison, Left: no Power-Down, Right: with Power-Down.

with respect to the in-order scheduler without the Power-Down mechanism, so we can see that the Power-Down mechanism reduces power consumption by 40-60%. Comparing the right graphs of Figures 5.1 and 5.2, we see that the Power-Down mechanism has a small effect on performance. Execution time increases by 2.5% for the in-order scheduler, by 2.1% for the memoryless scheduler, and 3.7% for the AHB scheduler.

Figure 5.2 also shows results for our new Power-Aware AHB scheduler, which when compared with the AHB scheduler (with the Power-Down mechanism) degrades performance by 1.6% and reduces power by 10.8%.

From these figures, it is difficult to understand how the schedulers compare in terms of energy efficiency. Figure 5.3 shows these same results using energy efficiency as a metric. We see that the AHB scheduler with the Power-Down mechanism is 4.9 times more efficient than the baseline in-order scheduler that does not use the Power-Down mechanism, and the Power-Aware AHB scheduler is an additional 9.4% more efficient than the AHB scheduler.

We conclude that, for daxpy, our power-aware adaptive history-based scheduler reduces power usage considerably and gives the best results in terms of efficiency.

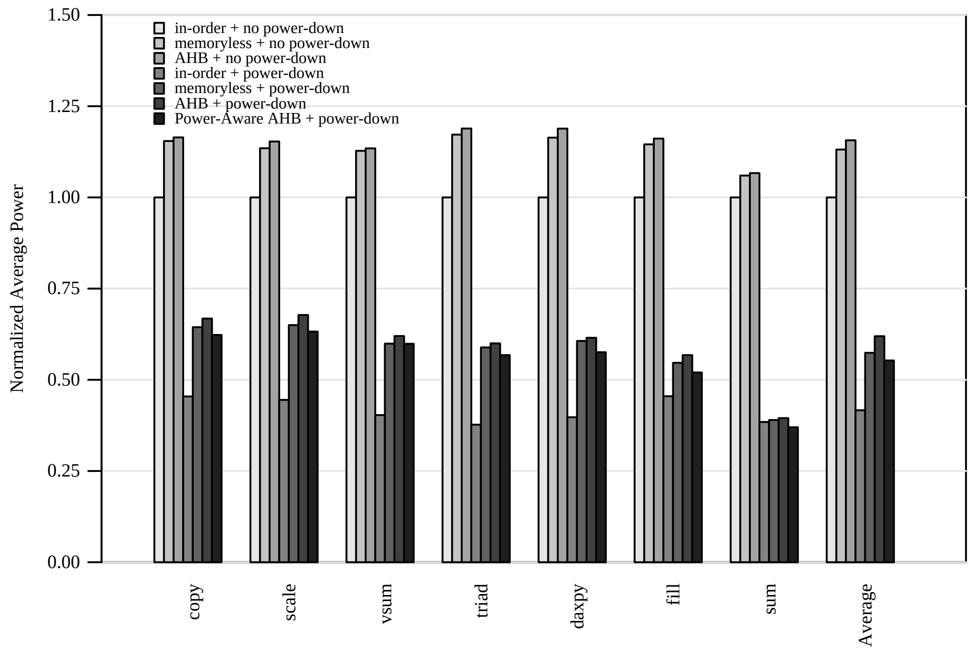

Figure 5.4: Comparison of power consumption for the Stream Benchmarks.

5.2.2 Stream and NAS Results

Figure 5.4 compares the four schedulers with and without the Power-Down mechanism. We see that the Power-Aware AHB gives the best power consumption results in each benchmark. On average the PA-AHB scheduler's power consumption is 5% better than the baseline in-order scheduler, and it is 5% better compared to the AHB scheduler. We compare the efficiency of the schedulers in Figure 5.5.

The NAS benchmarks are not as memory intensive as the Stream benchmarks, so the original AHB scheduler does not provide as much performance improvement (5-16%). On the other hand, because the memory system is less heavily utilized, when the Power-Down mechanism is added to the AHB, we see substantial power savings (Figure 5.6). As a result, our Power-Aware AHB scheduler significantly improves efficiency, as well (Figure 5.7).

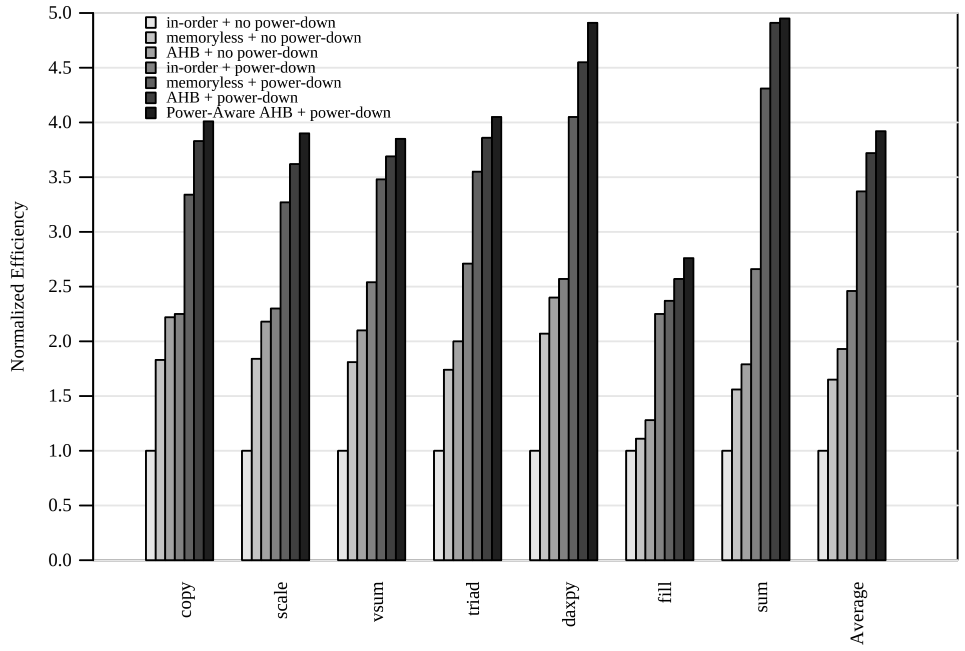

Figure 5.5: Efficiency comparison for the Stream Benchmarks.

5.3 Throttling Mechanism

The power-down mechanism that we presented can reduce power consumption to certain degree, but for additional power savings, we now describe a throttling mechanism that blocks commands to the DRAM.

Our throttling approach blocks commands for all ranks for some fixed period of T cycles. Other implementations could power-down single ranks at a time, but we do not explore this option here. Commands that are blocked cannot proceed to the CAQ, so they accumulate in the reorder queues, reducing bandwidth between the memory controller and the DRAM. When combined with our power-down mechanism, this throttling allows a rank to be powered-down for almost T cycles. If T is sufficiently long, the reorder queues become filled with commands for the blocked rank, and the system stalls. Thus, by changing the value of T, we can arbitrarily

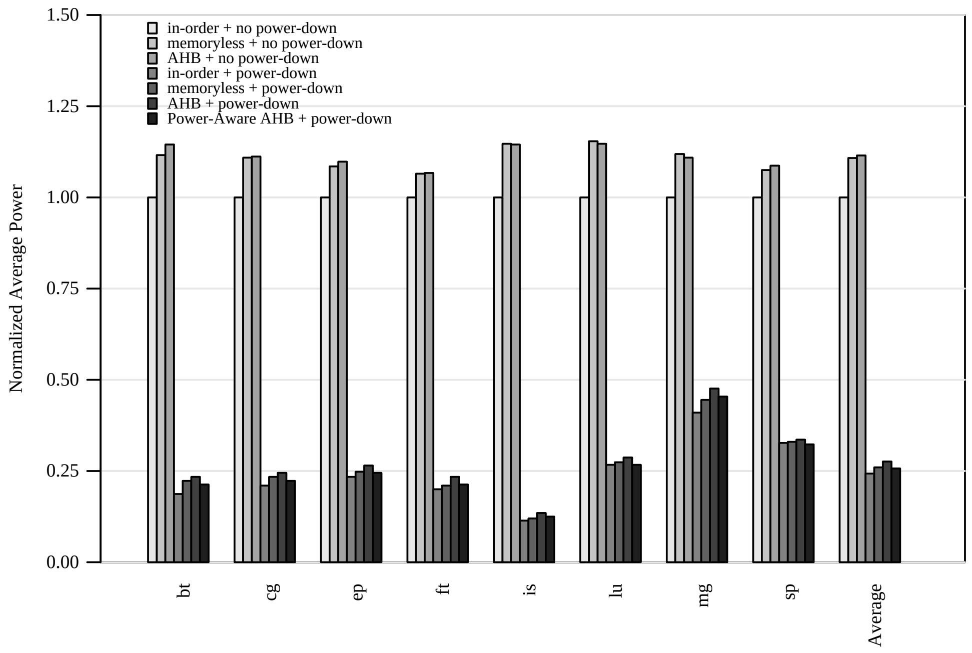

Figure 5.6: Comparison of power consumption for the NAS Benchmarks.

lower our system's average power consumption.

5.3.1 Estimating the Throttling Delay

To reduce DRAM power consumption to a target level, accurate estimation of the throttling delay, T, is crucial. An inaccurate model for T can cause two problems: (1) if T is overestimated, power consumption will be lower than the target, but at the same time performance will degrade more than it is necessary, (2) if T is underestimated, power consumption will be higher than the target. This second problem can be solved by choosing a lower target for power when estimating T. However, this conservative approach also will degrade performance unnecessarily.

In this section, we explain how we can accurately estimate the throttling delay that will reduce DRAM power consumption to a predetermined level, thereby causing as small a performance degradation as possible. Our method develops a

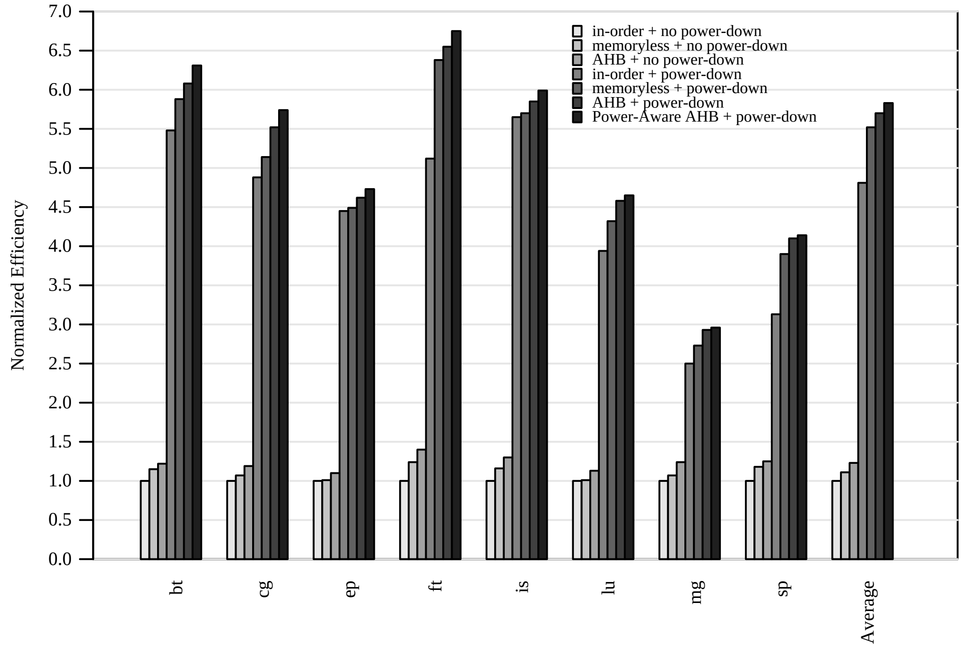

Figure 5.7: Efficiency comparison for the NAS Benchmarks.

regression model for estimating T and records the model coefficients in firmware. The memory controller, depending on the memory command pattern and a power budget, uses the model coefficients to calculate the throttling delay. We assume that the time period is sufficiently long for which a calculated T be valid that the overhead of the calculation is negligible. Note that the model coefficients vary depending on the processor frequency and DRAM properties. Thus, if system configuration changes, these coefficients should be regenerated.

To describe and evaluate our model generation method, we first investigate the relationship between power consumption and throttling delay for various benchmarks. We then explain how to develop various models for throttling delay; we discuss the metrics used to statistically evaluate our models; and finally, we present the comparison of the model results.

5.3.2 Relationship Between Power and Throttling Delay

To determine the interaction between DRAM power consumption and the throttling delay, we conduct experiments on the Stream benchmarks, which represent a wide variety of memory access patterns. For each benchmark, we perform simulations by varying T between 100 and 9,000 processor cycles for every 10,000 cycle interval. We also investigate the effect of data alignment by varying offsets between data vectors to generate 16 different versions of each benchmark. Figure 5.8 depicts the results for the benchmarks individually and also for all seven of them combined. In this figure, we observe that the relationship between power and T varies depending on both the benchmark and the offset between vectors in the same benchmark. For example, in the figure for all the benchmarks, we see that if the target power consumption is 40 Watts, depending on the benchmark and the offset value, the appropriate value of T varies between about 500 and 5,000 cycles. Thus, our experiments indicate that the relationship between power consumption and T is non-linear and that using only target power level to predict T will cause unnecessary performance degradation.

5.3.3 Models for Throttling Delay

Since the relationship between power and T is not linear, instead of trying to find a direct relationship between these two variables, we determine other features that can be used to relate them, and we use those features together with power to generate models for T. In Figure 5.8 we observe that the relationship between DRAM power consumption and T depends on the number of Reads, the number of Writes, and the offset between data streams.

To predict T for a given power target P, our baseline model is T1=f1(P,a), where a is a constant. This model lacks information about the number of Reads, Writes, and the offset between data streams. To examine a more detailed model, we create T2=f2(P,R,W,a) which includes the number of Reads and Writes in addition

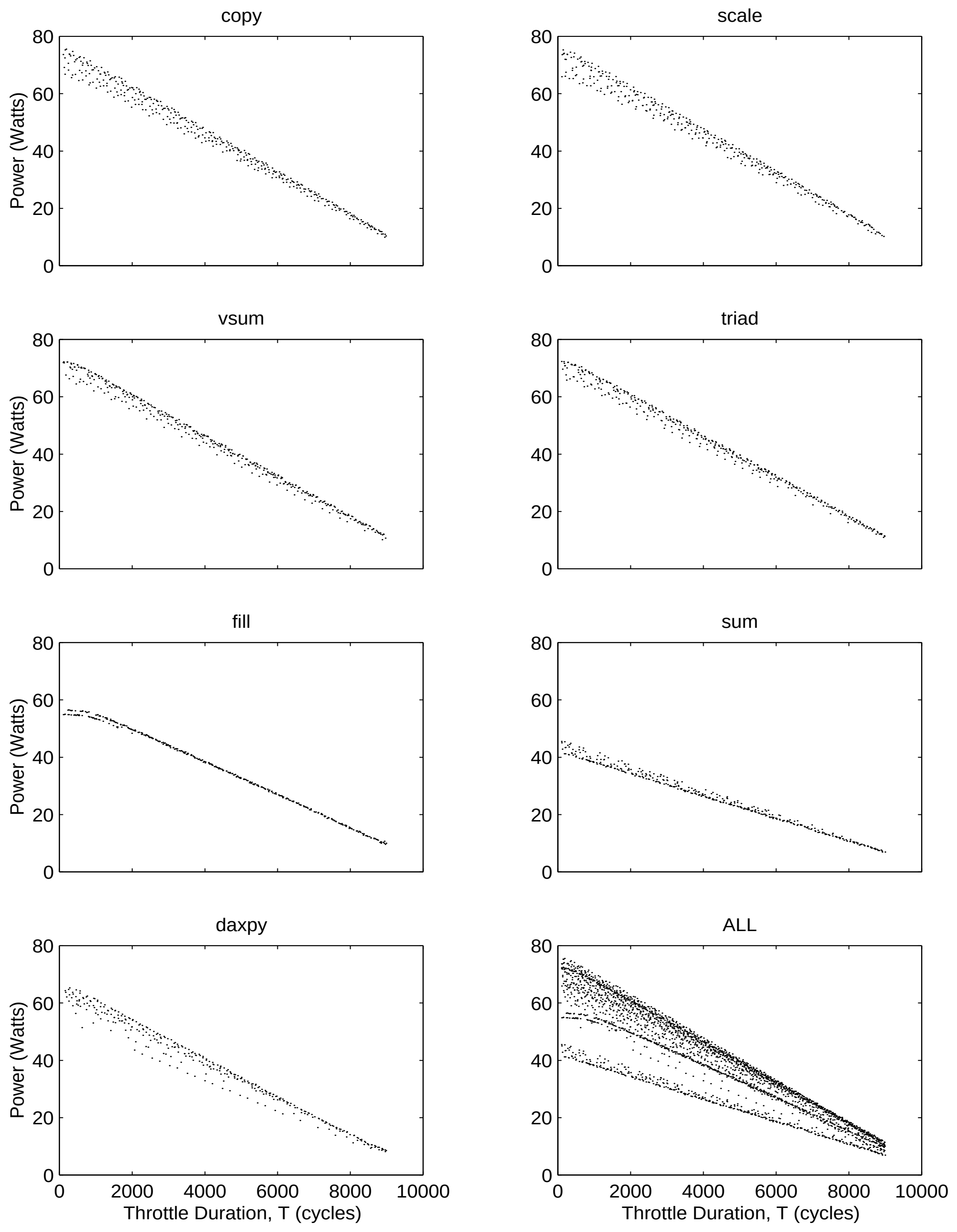

Figure 5.8: Relationship between DRAM power consumption and the throttling delay, for the Stream benchmarks.

to power information. And finally, we create T3=f3(P,R,W,B,a), which adds the number of bank conflicts, B, to the model T2. Our conjecture is that the number of bank conflicts, together with the number of Reads and Writes, will be a good representation for the power effects of the offset between data streams.

To determine coefficients for these models, we use our measurements for the Stream benchmarks, and we perform linear regression.

5.3.4 Regression Models

We now explain how linear regression can be used to develop models for throttling delay. We set up a system of equations where the known values are measured DRAM power, throttling delay, number of Reads, Writes, and bank conflicts. The unknowns in the system are the model coefficients. Solving this system gives us the values of the model coefficients that we are looking for.

The data used to determine unknown coefficients in regression analysis will be referred to as the *training set*, and the data used for testing the performance of models is known as the *test set*. The best way to evaluate the performance of a model is to use test sets that are independent from the training set.

Linear regression models for the throttling delay can be defined as

$$y_i = \beta_0 + \beta_1 \Phi_{i1} + \beta_2 \Phi_{i2} + ... + \beta_p \Phi_{ip}, \qquad i = 1, 2, ..., n. \tag{5.1}$$

where n is the number of elements in the training set, p is the number of coefficients less one (the degrees of freedom) in the model, and the y_i's are the measured throttling delays. This equation can also be stated in matrix form as:

$$\mathbf{y} = \mathbf{\Phi} \beta \tag{5.2}$$

The elements of the $\mathbf{\Phi}$ matrix are known. Each column of this matrix (sometimes called basis functions) represents one feature of the model. For example, for the model we propose in (5.1) the first column represents the measured DRAM power, the second column the number of Reads, the third column the number Writes, and the fourth column the number of bank conflicts. The values of $\mathbf{y}$ are the measured throttling delays from our training set. To find the value of the β vector, the coefficients of our model, we use a least squares method, which is defined as

$$\beta = \mathbf{\Phi}^{+}\mathbf{y} \tag{5.3}$$

where $\mathbf{\Phi}^{+}$ is the pseudo-inverse of $\mathbf{\Phi}$ [6].

The models we have discussed thus far are called first-order regression models, because the exponent of each Φ_j is one. Alternatively we can define second-order models which include quadratic, Φ_j^2, and cross-product, $\Phi_j\Phi_k$, terms. These models are called complete second-order models. Higher order models may sometimes provide better fit, but these might not generalize well. Thus, in our study we do not evaluate second-order models.

5.3.5 Statistical Analysis

To assess the adequacy of the models for T, we use *coefficient of determination*, R^2, which is probably the most extensively used measure of goodness for regression models. There are various definitions of R^2, each with its potential pitfalls [40]. We use the following definition, as suggested by Mason et al. [47]:

$$R^2 = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y}_i)^2} \tag{5.4}$$

In assessing the model accuracy R^2 is equal to unity when the model is as good a predictor of the target data as the simple model $\hat{y} = \bar{y}$, and it equals to zero if the model predicts the data values exactly [6]. For classification problems an R^2 value of 0.01 is generally acceptable, while for regression problems we need smaller values.

5.3.6 Comparison of the Model Results

The R^2 values for the test data set are 0.1659, 0.1344, and 0.0026 for the models T1, T2, and T3, respectively. Clearly, model T3 achieves the best accuracy, and it is also the the only model that satisfies the <0.01 requirement for the R^2. In Figure 5.9, we present the errors for predicting T for each of the three models. As the R^2 results suggest, we see that the model T3 predicts T much more accurately than the other two models.

More accurate estimation of the throttling delay results in more accurate estimation for DRAM power consumption as well. In Figure 5.10, we show the power effects of the three throttling delay models. This figure suggests that when we use T3, power consumption will in the range of +/- 3% of the target. However, for the other two models, the error range is about +/- 20%. The experiments and regression results confirm our conjecture that the number of bank conflicts, together with the number of Reads and Writes, create a good representation for DRAM power.

5.4 Summary

In this chapter we have shown how memory controllers can be used to improve power consumption as well as performance. We have evaluated three techniques. First, we show that a passive power-down mechanism that does not reorder memory commands can significantly reduce power consumption at the expense of a degradation of performance of less than 2.5%. This mechanism works well for all of the memory schedulers that we studied. Second, we introduce the Power-Aware Adaptive

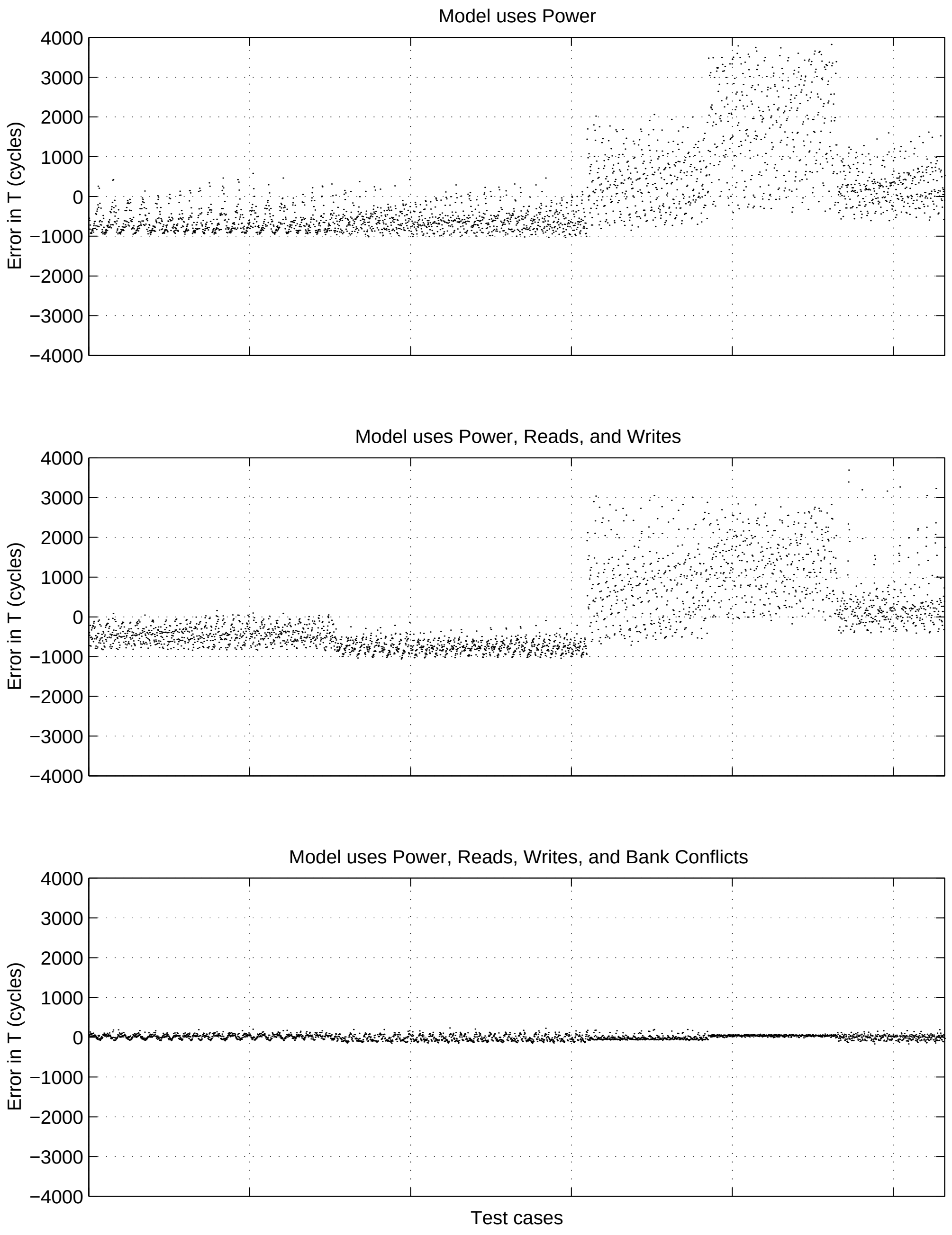

Figure 5.9: Errors in predicting the throttling delay, T.

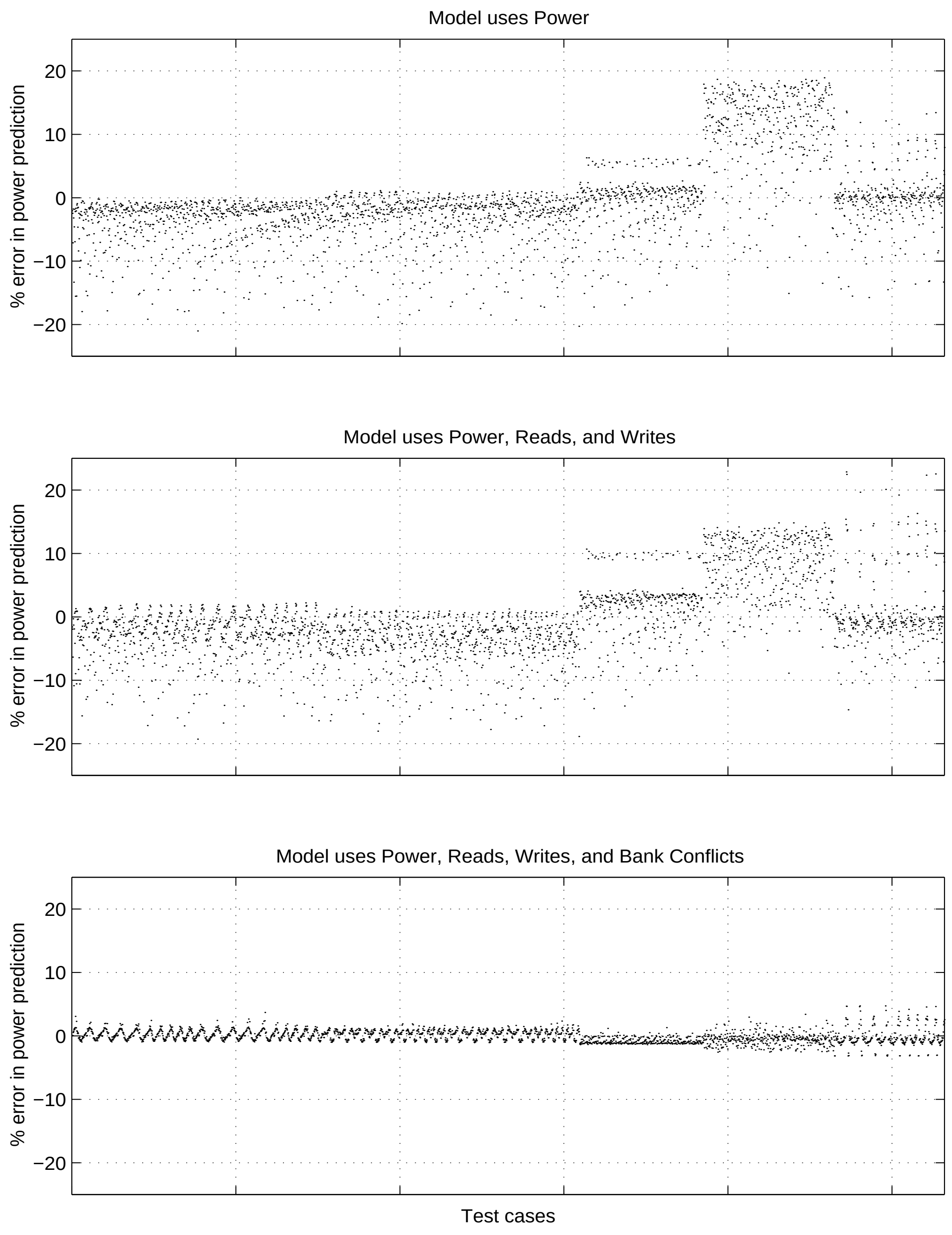

Figure 5.10: Proximity to the target DRAM power.

History-Based scheduler, a small modification of the previously studied Adaptive History-Based scheduler. This Power-Aware AHB scheduler improves the energy efficiency of the Stream and NAS benchmarks by an average of 400% compared to the in-order scheduler. The simple and effective changes to the original AHB scheduler support the claim that the AHB scheduler is a powerful framework for a variety of scheduling concerns. Finally, we present a throttling mechanism, which actively blocks commands in the reorder queues and can further decrease power consumption. This throttling mechanism might prove useful when memory systems must stay beneath some peak power threshold.

Chapter 6

Related Work

6.1 Methods to Improve Bandwidth

To increase sustained memory bandwidth, memory systems are organized as multiple banks that can be accessed simultaneously. In banked memory systems, simultaneous access is achieved by implementing some sort of interleaving [11]. Interleaved memory systems considerably improve bandwidth, but restrictions on accesses to banks, i.e. bank conflicts, prevent the system from attaining the maximum available bandwidth. Elimination of bank conflicts has been extensively studied for several decades. There are basically two broad classes of techniques to avoid bank conflicts: *static* approaches and *dynamic* methods.

6.1.1 Static Methods

Static bank conflict avoiding techniques, such as skewing [21, 13] or prime memory systems [60, 58], attempt to arrange the order of memory commands to minimize bank conflicts. Unfortunately, these static methods are effective for reducing only intra-stream bank conflicts, i.e. conflicts caused by one stream. There are also compiler-based methods such as data padding and loop transformations. For example, Moyer [53] presents a compiler-based approach, in which loops are unrolled and instructions are reordered to improve memory locality. But Moyer's technique applies specifically to stream-oriented workloads in cacheless systems.

6.1.2 Dynamic Methods

Dynamic conflict avoiding techniques have been proposed by various research groups [7, 71, 57, 52, 51, 50, 49, 61] to alleviate both intra- and inter-stream bank conflicts. As an example, the Impulse memory system by Carter et al. [7] improves memory performance by dynamically remapping physical addresses, but it requires modifications to the applications and the operating system.

There are also various heuristics that have been proposed to reorder memory commands. Valero et al. [71, 57] describe a memory reordering technique that dynamically eliminates bank conflicts by enforcing a strict round robin ordering of bank accesses. This ordering maximizes the average distance between any two consecutive accesses to the same bank and thus reduces the stalls due to bank conflicts. However, this technique considers only bank conflicts, and it can only eliminate bank conflicts if the requests are fairly uniformly distributed among banks.

McKee et al. [52, 51, 50, 49] propose a memory subsystem, Stream Memory Controller (SMC), to maximize bandwidth for streaming applications. Their design includes three main components: stream buffers, caches and a memory command scheduler. The compiler detects streams in the code and generates non-cacheable memory requests that bypass caches at run time and go directly to the stream buffers, which are essentially FIFO queues. The memory scheduler dynamically selects commands from either the stream buffers or from the caches. McKee et al. observe two issues in reordering commands in SMC: selecting the memory bank to which the next access to schedule, and selecting the FIFO queue which has a command for that particular bank. They examine and evaluate various dynamic ordering heuristics, but they don't propose an algorithm. The bank selection and FIFO selection policies that they evaluate are versions of a round robin scheduler. The memory controller considers each stream buffer in sequential fashion, streaming as much data as possible to the current buffer before going to the next buffer. This approach may reduce conflicts among streams, but it does not reorder references within a single stream.

Similar to static approaches, the preceding dynamic reordering studies are also restricted to bank conflicts. Valero et al.'s and McKee et al.'s approaches can be complementary to our approach in the sense that an AHB scheduler can use these methods as another optimization criteria. For example, when there are multiple commands in the reorder queues to choose from and when all the other optimization criteria are equal, an AHB scheduler can select the command that matches a predetermined sequence rather than choosing the oldest command.

Rixner et al. [61] explore several heuristics for reordering accesses on the Imagine stream processor [38]. Each of these heuristics reorder memory operations by considering the characteristics of modern DRAM systems and modern memory controllers. For example, one policy gives row accesses priorities over column accesses, and another gives column accesses priorities over row accesses. None of these simple policies is shown to be best in all situations, and none of them uses the command history when making decisions. Furthermore, these policies are not easily extended to more complex memory systems with a large number of different types of hardware constraints.

6.2 Hardware Prefetching for Irregular Applications

One line of hardware prefetching research has extended next-line prefetching [65, 34] by adding non-unit strides [55], by predicting strides [2, 19], and by supporting irregular strides using Markov predictors [33, 62]. Nesbit and Smith [54] introduce the *Global History Buffer* to improve prefetch effectiveness and reduce table sizes. None of these prefetchers has successfully exploited low amounts of spatial locality.

Another line of research focuses on detecting and exploiting spatial locality without tracking individual streams [32, 39, 44, 9]. Instead, variations of the *Spatial Locality Detection Table*, introduced by Johnson et al., track accesses to individual regions of memory so that spatially correlated data can be prefetched together. A problem with these approaches is the need for large tables to detect locality. Somogyi et al. [67] show how much smaller tables can be used by correlating spatial locality with the program counter in addition to parts of the data address. As a result, *Spatial Memory Streaming* can use tables as small as 64KB. Moreover, Somogyi et al. show performance improvements for commercial workloads, indicating that their technique can handle locality patterns that span large regions of memory. By contrast, our approach cannot prefetch as aggressively across irregular locality patterns but instead attempts to use a much smaller amount of hardware to prefetch the very small streams that likely make up these larger patterns.

Scheduled Region Prefetching (SRP) [43] prefetches large regions of memory, such as 4KB at a time, and introduces mechanisms for reducing the opportunity cost of prefetches. Prefetches to open banks are given priority, prefetched data are brought into the LRU position of the L2 sets, and prefetch commands are given low priority in the memory controller. In particular, the SRP prioritizer receives feedback from the memory system and issues prefetch commands only if the channels are idle and there is no pending request from the L2 cache. By contrast, our method uses feedback from the memory system to select from among five different prioritization policies, where its most conservative policy is roughly equivalent to the SRP prioritization policy. Our scheduling technique can improve performance because for some workloads the most conservative policy unnecessarily inhibits prefetches. For example, there may be pending demand requests that will not conflict with a prefetch command because they target different memory banks.

One issue with SRP is the high memory bandwidth pressure that it incurs because of its large regions. Wang et al. [73] solve this problem by using the compiler to trigger the prefetches selectively. Our solution instead uses a modest amount of hardware to prefetch at a much finer granularity.

Others have studied memory-side prefetching [1, 7, 75, 76, 66] and have shown that memory-side prefetching is largely orthogonal to processor-side prefetching [7, 26]. Unlike our approach, previous methods do not monitor the status of the memory system, so they can increase latencies for regular memory accesses.

6.3 DRAM Power Optimizations

Power consumption of the memory subsystem has recently received considerable attention. Power optimization techniques in DRAM can be classified in three categories [4]: hardware-based methods inside memory controller, compiler or operating system-directed techniques, and hybrid approaches.

6.3.1 Hardware-Based Approaches

Delaluz et al. [16] show, in the context of cacheless systems with Rambus DRAM, that the power-down idea offers good power savings for in-order scheduling. Their goal is to try to match predicted idle time with a low-power mode that has the appropriate latency to resume activity, however they do not evaluate this method in systems with caches. Fan et al. [18] extend this work to systems with 2-level caches. Irani et al. [31] give a theoretical analysis of dynamic power management in memory controllers. All of these methods basically monitor usage of memory sections and move to a different power level if the usage exceeds a threshold level. Since threshold values are system and application dependent, these algorithms are difficult to tune.

Previous hardware-based approaches for power savings assume in-order scheduling of the memory commands. We show that performance of memory system can be improved dramatically if commands are reordered [28, 29, 27]. As reordering improves performance, it naturally reduces the length of the gaps between memory commands. Since threshold-based predictive algorithms passively monitor memory traffic to decide when to power-down a memory section, we expect that shorter gaps will make those algorithms less effective. In contrast, our work takes an active approach and tries to reorder commands to save power while preserving performance.

6.3.2 Compiler- or Operating System-Based Approaches

Compiler-directed approaches aim to group memory accesses to the same memory sections to increase the size of idle periods. This goal is achieved by loop transformations [37], data layout optimizations [36], instruction scheduling [74, 46, 56], or with combinations of these methods [15]. In cacheless single processor systems, compile-time techniques can help the memory controller make better predictions for idle periods of memory sections. However, in systems with multi-level caches or with shared memory controllers [69, 35], the role of the compiler for power savings is limited.

Various studies have explored operating system support for power savings. Vahdat et al. [70] suggest incorporating energy efficiency as a first order design criteria for operating systems. Lu et al. [45] propose shutting down unused system components to save energy. By controlling the set of physical devices that are in active use, the actual power consumption for their access can be controlled by putting inactive devices into low-power mode. Zhou et al. [77] use this approach and change the size of allocated memory for processes by tracking page miss rate vs. memory size curve.

Other OS-based approaches rely on improving the placement of data in physical memory. Better page allocation policies can also save energy. By allocating new pages to memory that is already in use, the number of active memory devices can be kept to a minimum [41, 17]. One performance optimization is to have the operating system activate memory used by a newly scheduled process during a context switch, thus largely hiding the latency of exiting low-power mode [17, 23]. Intelligent page migration [14, 24], where data is moved from one memory device to another to reduce the number of active memory devices, has also been proposed. Recent work by Huang et al. [24] proposes an OS-based approach which reshapes memory traffic at the page granularity. This property of their method is similar to our approach of reordering memory commands.

Our scheduling methods and OS-based approaches may be complementary to each other, because our approach operates at a much finer granularity compared to OS-based techniques. However, with the use of large page sizes [35], OS-based techniques which require data migration may degrade performance considerably.

Of course, any approach that minimizes the number of active memory devices also reduces the available memory bandwidth. Accesses previously performed in parallel to different memory devices may need to be performed serially to the same memory device. Most previous work does not accurately model the performance loss that stems from such serialization. By contrast, our detailed simulators allow us to model such effects accurately.

6.3.3 Hybrid Approaches

Recent studies have shown the importance of addressing DRAM power consumption in large server systems [42, 5]. Huang et al. propose a cooperative software-hardware approach that tracks process-specific idle periods to exploit DDR's low-power modes for ranks of DRAM devices [25]. Felter et al. [20] jointly manage processor and DRAM power by attempting to maximize system performance for a given total power budget, which is particularly useful when either the CPU or DRAM is significantly less utilized than the other. Our approach is transparent to software, which we believe is critical for successful adoption.

Chapter 7

Conclusions and Future Work

In the last few decades, because of increasing memory latencies and increasing bandwidth demands, memory systems have become a major performance bottleneck for computer systems. More recently, power consumption of DRAM chips has also become a first order concern. Previous proposals for improving latency, bandwidth, or power aspects of memory systems have significantly increased the complexity of processors and/or memory organizations. Although processor and memory systems have been explored extensively, the interface between them, the memory controller, had received relatively less attention. As processors and memory systems become increasingly complex, it is natural to explore ways that the memory controller can be made more sophisticated. Therefore, in this dissertation, we have concentrated on the memory controller, and we have proposed novel solutions to all three aspects of memory systems. We have evaluated our techniques in the context of the memory controller of a highly tuned modern processor, the IBM Power5+. Our evaluation for both technical and commercial benchmarks in single-threaded and simultaneous multi-threaded environments has shown that our techniques for latency hiding, bandwidth increase, and power reduction achieve significant improvements.

This dissertation makes the following contributions:

- To increase available bandwidth between the memory controller and DRAM, we have introduced a scheduling approach that incorporates several novel techniques. In this approach, we use the command history to select commands that reduce delays due to resource conflicts. We use the command history also to schedule commands that match some expected command pattern. Because the goals of these two techniques may conflict, we probabilistically combine them in a single history-based scheduler that partially satisfies both goals. Finally, we implement three history-based schedulers—each tailored to a different command pattern—and we dynamically select from among those based on the observed ratio of Reads and Writes.

  Our new scheduling approach improves the performance of the Stream, NAS, and a set of commercial benchmarks over a scheduler that does not change the order of commands by 55.6%, 25.6%, and 51.6%, respectively. When compared to the best approach proposed so far, for the same benchmarks, our scheduler is better by 16.0%, 9.7%, and 7.5%, respectively.

  To explain our results, we have looked inside the memory system to provide insights about how our solution changes the various bottlenecks within the system. We have found that our solution is more robust than previous scheduling approaches in the sense that our solution is less sensitive to changes in design parameters. We have also found that the AHB scheduler is superior to the previous schedulers even when the other schedulers are given additional hardware resources.

- To hide memory latency, we have introduced a new stream-based prefetching technique, Adaptive Stream Detection, which is effective for streams of any length, including very short streams. By monitoring the amount of spatial locality in a program's execution in a Stream Length Histogram, our prefetcher can probabilistically decide when to start and stop prefetching based on the recently observed behavior. A secondary contribution of our prefetching approach is the notion of Adaptive Scheduling, which adapts the aggressiveness of the prefetcher based on the observed number of conflicts between prefetch commands and regular commands.

  We have shown that when implemented as a memory-side prefetcher, our prefetching approach provides significant performance improvements, even for commercial workloads that have low spatial locality. When we combine our scheduling and prefetching methods, we obtain 14.3%, 13.7%, and 11.2% performance improvements for the SPEC2006fp, NAS, and the commercial benchmarks, respectively.

- We have shown how memory controllers can be used to improve power consumption as well as performance. We have made three contributions. First, we have presented details of how to implement a DRAM power-down mechanism with as small a performance degradation as possible. Second, we have modified our scheduling method to include power consumption as a new criterion during scheduling. Finally, we have introduced a throttling mechanism, which actively blocks commands in the reorder queues. To accurately calculate the duration of throttling for a given power budget, we have developed a methodology which uses regression models based on the measurement data.

In addition to providing substantial performance and power improvements, our techniques are superior to the previously proposed methods in terms of cost as well. For example, a version of our scheduling approach has been implemented in the Power5+, and it has increased the transistor count of the chip by only 0.02%. Similarly, we estimate that our prefetching approach will increase the transistor count of the chip by approximately 0.12%, which is much less than the cost of the previously proposed methods.

This dissertation has shown that without increasing the complexity of neither the processor nor the memory organization, all three aspects of memory systems can be significantly improved with low-cost enhancements to the memory controller.

Although we have evaluated our solutions in the context of the IBM Power5+, our solutions should apply to other modern general purpose processors too. Because, most modern systems use a common DRAM technology, therefore, the assumptions that our solutions make about DRAMs are true for other systems as well. In particular, our solutions rely on the following assumptions: (1) complex DRAM structure with multiple units of sub-organization, and (2) existence of a power-down mechanism in DRAM. Because of increasing bandwidth demands, we should expect more parallelism in future DRAM organizations. And because of increasing importance of power consumption, we should also expect DRAMs to continue having power-down mechanisms. Therefore, our solutions are likely to apply to future systems as well.

The current trend in computer architecture is to use simultaneous multi-threading and to design multi-processor chips. This trend increases the pressure on the memory system. Thus, memory controllers, and therefore our solutions, are likely to become more important in the future.

There are two possible ways to extend this research: (1) we can try to further improve the techniques that we have presented, and (2) we can implement our techniques in places other than the memory controller.

Although our techniques provide significant improvements, they are far from obtaining the performance of the ideal memory system, which has zero latency and infinite bandwidth. Indeed, the ideal memory system will further improve the performance of the SPEC2006fp, NAS, and commercial benchmarks by 44.2%, 37.6%, and 52.9%, respectively, over the combined use of our latency and bandwidth improvement techniques.

We have shown that our memory scheduling approach achieves more than

95% of the bandwidth of a perfect scheduler. Therefore, there is not much headroom to improve this method on the Power5+. However, for other systems, incorporating bank conflicts into the scheduler can be considered at the expense of costlier design. Despite our scheduling approach, the prefetching method that we have introduced has headroom for further improvements. A major improvement to our method may occur if the compiler generates prefetch instructions for streams of length one and our prefetching technique gives special attention to those prefetches. Modifying cache replacement policies may also affect the occurrence of single element streams. Another improvement opportunity is to extend our prefetching method by designing multiple prefetchers and selecting one by using certain bits of the memory address and/or program counter. Also, in this dissertation, we have evaluated the implementation of only single line prefetching. As another improvement to our prefetching technique, implementation of multiple line prefetching can be considered.

Finally, in this dissertation, we have focused to improve the bandwidth and latency between the memory controller and DRAM. However, similar concerns exist in other parts of systems as well. A natural extension of our work is the application of our techniques into the L2 cache controller to improve bandwidth and latency inside the chip.

Bibliography

[1] T. Alexander and G. Kedem. Distributed prefetch-buffer/cache design for high-performance memory systems. In *HPCA '96: Proceedings of the 2nd International Symposium on High Performance Computer Architecture*, pages 254–263. IEEE Computer Society, 1996.

[2] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5):609–623, 1995.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks (94). Technical report, RNR Technical Report RNR-94-007, March 1994.

[4] L. Benini, A. Macii, and M. Poncino. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *Transactions on Embedded Computing Systems*, 2(1):5–32, 2003.

[5] R. Bianchini and R. Rajamony. Power and energy management for server systems. Technical Report DCS-TR-528, Rutgers University, June 2003.

[6] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[7] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *HPCA' 99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 70–79. IEEE Computer Society, 1999.

[8] A. Charlesworth, N. Aneshansley, M. Haakmeester, D. Drogichen, G. Gilbert,

R. Williams, and A. Phelps. The starfire SMP interconnect. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–20. ACM Press, 1997.

[9] C. F. Chen, S.-H. Yang, B. Falsafi, and A. Moshovos. Accurate and complexity-effective spatial pattern prediction. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 276–287. IEEE Computer Society, 2004.

[10] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson. Design and implementation of the Power5 microprocessor. In *Proceedings of the 41st Annual Conference on Design Automation*, pages 670–672, 2004.

[11] H. G. Cragon. *Memory Systems and Pipelined Processors.* Jones and Bartlett, 1996.

[12] Z. Cvetanovic. Performance analysis of the Alpha 21364-based HP GS1280 multiprocessor. In *ISCA' 03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 218–229. ACM Press, 2003.

[13] I. D. T. Harper and J. R. Jump. Performance evaluation of vector accesses in parallel memories using a skewed storage scheme. In *ISCA '86: Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 324–328. IEEE Computer Society, 1986.

[14] V. Delaluz, M. Kandemir, and I. Kolcu. Automatic data migration for reducing energy consumption in multi-bank memory systems. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 213–218. ACM Press, 2002.

[15] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *CASES '00: Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 138–147. ACM Press, 2000.

[16] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. Irwin. DRAM energy management using software and hardware directed power mode control. In *HPCA '01: Proceedings of the 7th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 2001.

[17] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Scheduler-based DRAM energy management. In *DAC '02: Proceedings of the 39th Conference on Design Automation*, pages 697–702. ACM Press, 2002.

[18] X. Fan, C. Ellis, and A. Lebeck. Memory controller policies for DRAM power management. In *ISLPED '01: Proceedings of the 2001 International Symposium on Low-Power Electronics and Design*, pages 129–134. ACM Press, 2001.

[19] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143. ACM Press, 1997.

[20] W. Felter, K. Rajamani, C. Rusu, and T. Keller. A performance-conserving approach for reducing peak power consumption in server systems. In *ICS '05: Proceedings of the 19th ACM International Conference on Supercomputing*, pages 293–302. ACM Press, 2005.

[21] Q. S. Gao. The Chinese remainder theorem and the prime memory system. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 337–340. ACM Press, 1993.

[22] http://www.micron.com. Technical report.

[23] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *USENIX 2003 Annual Technical Conference*, 2003.

[24] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making DRAM less randomly accessed. In *ISLPED '05: Proceedings of the 2005 International Symposium on Low-Power Electronics and Design*, August 2005.

[25] H. Huang, K. G. Shin, C. Lefurgy, K. Rajamani, T. Keller, E. V. Hensbergen, and F. Rawson. Cooperative software-hardware power management for main memory. In *Proceedings of the Power-Aware Computer Systems: 4th International Workshop*, pages 61–77, 2004.

[26] C. Hughes and S. Adve. Memory-side prefetching for linked data structures. Technical Report UIUCDCS-R-2001-2221, University of Illinois at Urbana-Champaign, 2001.

[27] I. Hur. Method and system for creating and dynamically selecting an arbiter design in a data processing system. *US patent filed by International Business Machines*, September 2004.

[28] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 343–354. IEEE Computer Society, December 2004 (Winner, Best Paper Award).

[29] I. Hur and C. Lin. Adaptive history-based memory schedulers for modern processors. *IEEE Micro (Top Picks Issue)*, 26(1):22–29, 2006.

[30] I. Hur and C. Lin. Memory prefetching using adaptive stream detection. In *Proceedings of the 39th Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE Computer Society, December 2006.

[31] S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *Transactions on Embedded Computing Systems*, 2(3):325–346, 2003.

[32] T. L. Johnson, M. C. Merten, and W.-M. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64. IEEE Computer Society, 1997.

[33] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263. ACM Press, 1997.

[34] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373. ACM Press, 1990.

[35] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[36] M. Kandemir. Impact of data transformations on memory bank locality. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, page 10506. IEEE Computer Society, 2004.

[37] M. Kandemir, U. Sezer, and V. Delaluz. Improving memory energy using access pattern classification. In *ICCAD '01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 201–206. IEEE Computer Society, 2001.

[38] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, 2001.

[39] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA '98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368. IEEE Computer Society, 1998.

[40] T. O. Kvalseth. Cautionary note about R2. *The American Statistician*, 39(4):279–285, November 1985.

[41] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–116. ACM Press, 2000.

[42] C. Lefurgy, K. Rajamani, F. L. Rawson III, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, December 2003.

[43] W. F. Lin, S. K. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *HPCA '01: Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 301–312. IEEE Computer Society, 2001.

[44] W. F. Lin, S. K. Reinhardt, D. Burger, and T. R. Puzak. Filtering superfluous prefetches using density vectors. In *ICCD '01: Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*, pages 124–132. IEEE Computer Society, 2001.

[45] Y.-H. Lu, L. Benini, and G. D. Micheli. Operating-system directed power reduction. In *ISLPED '00: Proceedings of the 2000 International Symposium on Low-Power Electronics and Design*, pages 37–42. ACM Press, 2000.

[46] C.-G. Lyuh and T. Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. In *DAC '04: Proceedings of the 41st Annual Conference on Design Automation*, pages 81–86. ACM Press, 2004.

[47] R. L. Mason, R. F. Gunst, and J. L. Hess. *Statistical Design and Analysis of Experiments.* John Wiley & Sons, 1989.

[48] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, http://www.cs.virginia.edu/stream/.

[49] S. A. McKee. Hardware support for dynamic access ordering: Performance of some design options. Technical Report CS-93-08, University of Virginia, September 1993.

[50] S. A. McKee. *Maximizing Memory Bandwidth for Streamed Computations.* PhD thesis, University of Virginia, May 1995.

[51] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, and A. P. Batson. Smarter memory: Improving bandwidth for streamed references. *IEEE Computer*, pages 54–63, July 1998.

[52] S. A. McKee, W. A. Wulf, J. H. Aylor, M. H. Salinas, R. H. Klenke, S. I. Hong, and D. A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, 2000.

[53] S. A. Moyer. *Access ordering and effective memory bandwidth.* PhD thesis, University of Virginia, 1993.

[54] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, pages 96–105, 2004.

[55] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *ISCA '94: Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33. IEEE Computer Society, 1994.

[56] P. R. Panda and L. Chitturi. An energy-conscious algorithm for memory port allocation. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 572–576. ACM Press, 2002.

[57] M. Peiron, M. Valero, E. Ayguade, and T. Lang. Vector multiprocessors with arbitrated memory access. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 243–252. ACM Press, 1995.

[58] R. Raghavan and J. P. Hayes. On randomly interleaved memories. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, pages 49–58. IEEE Computer Society, 1990.

[59] K. Rajamani. Memsim users' guide, IBM research report. Technical Report RC23431, October 2004.

[60] B. R. Rau. Pseudo-randomly interleaved memory. In *ISCA '91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 74–83. ACM Press, 1991.

[61] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access

scheduling. In *ISCA '00: Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, June 2000.

[62] S. Sair, T. Sherwood, and B. Calder. A decoupled predictor-directed stream prefetching architecture. *IEEE Transactions on Computers*, 52(3):260–276, March 2003.

[63] H. Schwetman. CSIM19: a powerful tool for building system models. In *WSC '01: Proceedings of the 33nd Conference on Winter Simulation*, pages 250–255. IEEE Computer Society, 2001.

[64] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS-VII: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 26–36. ACM Press, 1996.

[65] A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Transactions on Computers*, 11(12):7–12, December 1978.

[66] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *ISCA '02: Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 171–182, 2002.

[67] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial memory streaming. In *ISCA '06: Proceedings of the 33th Annual International Symposium on Computer Architecture*, pages 252–263. ACM Press, 2006.

[68] Standard Performance Evaluation Corporation. *SPEC CPU 2006, http://www.spec.org*, August 2006.

[69] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Lee, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[70] A. Vahdat, A. Lebeck, and C. S. Ellis. Every joule is precious: the case for revisiting operating system design for energy efficiency. In *EW 9: Proceedings of the 9th Workshop on ACM SIGOPS European Workshop*, pages 31–36. ACM Press, 2000.

[71] M. Valero, T. Lang, J. M. Llaber, M. Peiron, E. Ayguade, and J. J. Navarra. Increasing the number of strides for conflict-free vector access. In *ISCA '92: Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 372–381. ACM Press, 1992.

[72] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance

optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–35. IEEE Computer Society, 2002.

[73] Z. Wang, D. Burger, K. S. McKinley, S. K. Reinhardt, and C. C. Weems. Guided region prefetching: a cooperative hardware/software approach. In *ISCA '03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 388–398. ACM Press, 2003.

[74] Z. Wang and X. S. Hu. Power aware variable partitioning and instruction scheduling for multiple memory banks. In *DATE '04: Proceedings of the Conference on Design, Automation and Test in Europe*, page 10312. IEEE Computer Society, 2004.

[75] C.-L. Yang and A. R. Lebeck. Push vs. pull: data movement for linked data structures. In *ICS '00: Proceedings of the 14th International Conference on Supercomputing*, pages 176–186. ACM Press, 2000.

[76] L. Zhang, Z. Fang, M. Parker, B. Mathew, L. Schaelicke, J. Carter, W. Hsieh, and S. McKee. The Impulse memory controller. *IEEE Transactions on Computers*, 50(11):1117–1132, November 2001.

[77] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 177–188. ACM Press, 2004.

Vita

Ibrahim Hur was born in Izmir, Turkey, on March 29, 1968, the son of Hamza Hur and Mufide Hur. After receiving his high school diploma from Izmir Ataturk Lisesi in Izmir, he took the annual national university entrance examination, in which his score ranked him 40^{th} among about one million students. He studied Computer Science and Engineering at Ege University, Izmir. After receiving his Bachelor of Science degree in 1991, he worked as a systems analyst for two years in a project for NATO. In 1993, he received a scholarship from Turkish government for graduate studies, and he came to the United States. He completed the degree of Master of Science in Computer Science at Southern Methodist University, Dallas, Texas, in 1995, and he entered the Graduate School at The University of Texas at Austin. In 1997, he joined the International Business Machines Corporation. He is currently employed by the IBM Systems and Technology Group in Austin, where he works in the areas of computer architecture and performance analysis. During his graduate studies, Ibrahim was supported by teaching and research assistantships, and he received the IBM Ph.D. Fellowship in 2000 and 2001.

Permanent Address: 247 Sokak No.2/2 D.15, Bornova, Izmir, Turkey

This dissertation was typeset with $\LaTeX\,2_\varepsilon$by the author.