Adaptive History-Based Memory Schedulers

Ibrahim Hur[†]
ihur@cs.utexas.edu

Calvin Lin[‡]
lin@cs.utexas.edu

[†]Dept. of Electrical and Computer Engr.
The University of Texas at Austin
Austin, TX 78712

[†]IBM Corporation
Austin, TX 78758

[‡]Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

As memory performance becomes increasingly important to overall system performance, the need to carefully schedule memory operations also increases. This paper presents a new approach to memory scheduling that considers the history of recently scheduled operations. This history-based approach provides two conceptual advantages: (1) it allows the scheduler to better reason about the delays associated with its scheduling decisions, and (2) 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. We evaluate our solution using a cycle-accurate simulator for the recently announced IBM Power5. When compared with an in-order scheduler, our solution achieves IPC improvements of 10.9% on the NAS benchmarks and 63% on the data-intensive Stream benchmarks. Using microbenchmarks, we illustrate the growing importance of memory scheduling in the context of CMP's, hardware controlled prefetching, and faster CPU speeds.

1 Introduction

As the gap between processor speeds and memory speeds continues to increase, memory system performance is becoming increasingly important. To address this problem, memory controllers typically use internal buffering and parallelism to increase bandwidth. To date, memory controllers for most general purpose processors simply pass instructions to DRAM in FIFO order. While these in-order schedulers are simple, they can yield poor bandwidth when memory operations must stall because of hardware hazards. For example, when a memory bank is accessed, subsequent accesses to that bank must typically stall for many cycles until the first access is completed.

Rixner, et al. [12] show that for cacheless media processors, bandwidth can be significantly improved by reordering memory operations in much the same way that instructions are scheduled in a pipelined multiprocessor. In particular, Rixner et al. propose various simple techniques that are sensitive to the physical characteristics of a DRAM's banks, rows, and columns. In the example of a busy memory bank, memory operations would be reordered to allow operations to other banks to proceed, thus increasing throughput.

The simple techniques that have been previously used [12] suffer from two limitations. First, they are essentially greedy solutions that are unable to consider the longer term effects of a scheduling decision. For example, if there are multiple pending operations that do not have bank conflicts, the operations are scheduled oldest first. This policy avoids an immediate bank conflict, but it may lead to an unavoidable conflict on the next cycle. Second, the notion of scheduling for the characteristics of the hardware is insufficient; a good scheduler should also consider the behavior of the application.

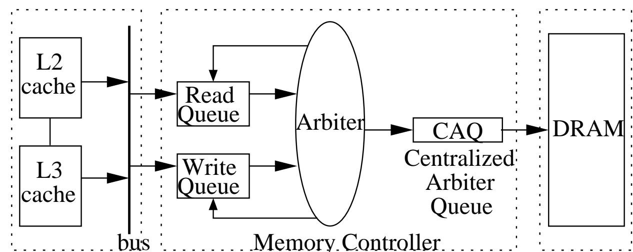

Figure 1. The Power5 memory controller.

To understand this second point, consider the execution of the daxpy kernel on the IBM Power5's memory controller. As shown in Figure 1, this memory controller sits between the L2/L3 caches and DRAM; it has two reorder queues (Read Queue and Write Queue), an arbiter that se-

lects operations from the reorder queues, and a FIFO Centralized Arbiter Queue, which buffers operations that are sent to the DRAM. The daxpy kernel performs two reads for every write. If the arbiter 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 arbiter should select memory operations so that the ratio of reads and writes matches that of the application.

This paper shows how command history[1] can address both limitations. By tracking the recently scheduled operations—those that have moved from the CAQ to DRAM—an arbiter can consider the longer term effects of scheduling decisions. An arbiter can also use such information to schedule operations to match some pre-determined mixture of Reads and Writes. We show how *history-based* arbiters can be implemented as finite state machines (FSM's). The use of FSM's provides a mechanism for encoding cost models that reflect the complexities of modern memory systems. Moreover, if one FSM encodes a scheduling policy that attempts to minimize latency and another encodes a policy that attempts to match a given mixture of Reads and Writes, a third FSM can probabilistically choose from among the first two to partially satisfy both scheduling goals.

These history-based arbiters, however, are limited because they are each tailored to one particular pattern of Reads and Writes. To support a variety of workloads, we introduce the notion of an *adaptive history-based* arbiter, which adaptively chooses from among multiple history-based arbiters. This paper evaluates a simple solution that uses three history-based arbiters, each with a history length of two. We show that this short history length works surprisingly well.

This paper extends previous work in several ways.

1. We present the notion of adaptive history-based arbiters, and we provide algorithms for designing such arbiters.

2. 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.

3. We evaluate our solution using a cycle-accurate simulator for the Power5. When compared with an in-order arbiter, our solution improves IPC on the NAS benchmarks by a geometric mean of 10.9%, and it improves IPC on the Stream benchmarks [9] by over 63%. When compared against one of Rixner et al.'s solution, our solution sees improvements of 5.1% for the NAS benchmarks and over 18% for the Stream benchmarks. The performance improvements for the NAS benchmarks are conservative because they simulate a single-processor system, whereas the Power5 is a dual-processor SMT system with a single memory controller. As multiple threads and multiple processors are used, memory traffic will increase, magnifying the effects of a good arbiter. To quantify some of these effects, we illustrate the impact of increased CPU speed, of hardware-controlled prefetching, and of chip multiprocessors.

4. We show that a history length of two produces good results. The hardware cost of our solution is minimal, and in the context of the Power5, our solution achieves 95-98% of the performance of a perfect DRAM that never experiences a hardware hazard in memory.

5. Finally, we provide insights to explain why our solution improves the bandwidth of the Power5's memory system.

[1] In this paper we use the term "memory operation" and "memory command" synonymously to refer to Reads and Writes to DRAM. We also use the terms "scheduler" and "arbiter" interchangeably.

The remainder of this paper is organized as follows. Section 2 describes related work. Section 3 provides background for understanding our paper, describing the relevant portions of the Power5 microarchitecture. Section 4 describes our new solution, which we empirically evaluate in Section 5. Finally, we conclude in Section 6.

2 Related Work

Most previous work on memory scheduling comes from work on streaming processors. Such systems do not exhibit the same complexity as today's memory hierarchies, and none of the previous work used a history-based approach to scheduling.

Rixner et al. [12] explore several simple policies for reordering accesses on the Imagine stream processor [8]. These policies reorder memory operations by considering the characteristics of modern DRAM systems. For example, one policy gives row accesses priorities over column access, 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 use 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.

Moyer [11] uses compiler transformations to change the order of the memory requests generated by the processor

to obtain better memory system performance. Loops are unrolled and instructions are reordered to improve memory locality. Moyer's technique applies specifically to stream-oriented workloads in cacheless systems. McKee [10] [17] uses a runtime approach to order the accesses to streams in a stream memory controller. This approach uses a simple ordering mechanism: the memory controller considers each stream buffer in round-robin 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.

Valero et al. [15] describes a technique that dynamically eliminates bank conflicts on vector processors. Elements of vectors are created out of order to eliminate memory bank conflicts. This approach assumes uniform memory access time. The Impulse memory system by Carter et al. [2] improves memory system performance by dynamically remapping physical addresses. This approach requires modifications to the applications and operating system.

3 Background

3.1 A Modern Architecture: The IBM Power5

The Power5 [4, 7] is IBM's recently introduced successor to the Power4 [14]. The Power5 chip has 276 million transistors and is designed to address both scientific and commercial workloads. Some improvements in the Power5 over the Power4 include a larger L2 cache, simultaneous multithreading, power-saving features, and an on-chip memory controller.

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 has hardware data prefetching units that prefetch from memory to L2, and from L2 to L1.

Each memory controller is shared by two processors. The Power5 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.

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 processor'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.

3.2 The Power5 Memory System

The Power5 systems that we consider use DDR2-266 DRAM 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 different constraints. For example, port conflicts, rank conflicts, and bank conflicts each incur their own delay (see Section 5.4 for details), and the costs of these delays depends on whether the operations are Reads or Writes. In this system, bank conflict delays are much longer than the delays introduced by rank or port conflicts.

4 Our Solution

This section describes our new approach to memory controller design, which focuses on making the scheduler both history-based and adaptive. A history-based arbiter 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 arbiter that encodes both goals. The result is a history-based arbiter that is optimized for one particular command pattern. To overcome this limitation, we introduce adaptivity by using multiple history-based arbiters; our adaptive arbiter observes the recent command pattern and periodically chooses the most appropriate history-based arbiter.

4.1 History-Based Arbiters

In this section we describe the basic structure of history-based arbiters. Similar to branch predictors, which use the history of the previous branches to make predictions [5], history-based arbiters use the history of the previous memory commands to decide what command to send to memory next. These arbiters 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 arbiters make decisions based on both the command history and the set of available commands from the reorder queues. The goal of the arbiter 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 arbiter 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 arbiter 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 arbiter 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 2 we show 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.

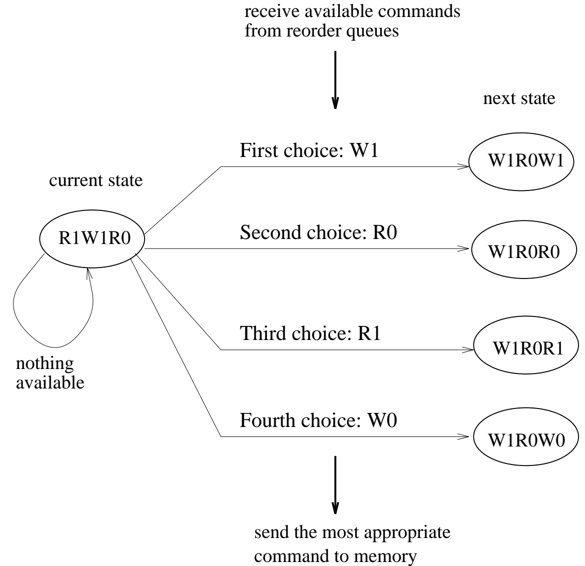

Figure 2. Transition diagram for the current state $R1W1R0$. Each available command type has different selection priority.

4.2 Design Details of History-Based Arbiters

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 arbiter to schedule commands to match some expected 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 arbiters for each of the two prioritization goals in isolation. We then provide a simple algorithm for probabilistically combining two arbiters.

4.2.1 Optimizing for the Command Pattern

Algorithm 1 generates state transitions for an arbiter 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, we then compute the Read/Write ratio of each possible next command. Finally, we sort the next commands according to their Read/Write ratios. For example, consider an arbiter 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 arbiter, the algorithm uses some secondary criterion, such as the expected latency, to make final decisions.

4.2.2 Optimizing for the Expected Latency

To develop an arbiter 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 c_{new} and the previous n commands that were already 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

Algorithm 1 command_pattern_arbiter(n)

// n is the history string size

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 arbiter, 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**

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 arbiter does not have the ability to track the number of cycles passed since the commands c_1, $c_2, \ldots, c_n$ were sent. Therefore, we assume that those commands were sent one cycle apart from each other. In the next step, we calculate 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 an FSM for an arbiter 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 the secondary criterion—in this case the deviation from the command pattern—to break ties.

Algorithm 2 expected_latency_arbiter(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 the expected latency, T_{delay}.
5: **end for**
6: Sort possible commands with respect to T_{delay}.
7: **for** commands with equal expected latency value **do**
8: Use Read/Write ratios to make decisions.
9: **end for**
10:
11: **for** each possible next command **do**
12: Output the next state in the FSM.
13: **end for**
14: **end for**

4.2.3 A Probabilistic Arbiter Design Algorithm

Since there may be no *a priori* way to prioritize one optimization criterion over the other, Algorithm 3 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 3 probabilistic_arbiter

1: **if** random_number $<$ threshold **then**
2: command_pattern_arbiter
3: **else**
4: expected_latency_arbiter
5: **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.

4.3 Adaptive Selection of Arbiters

A schematic of our adaptive history-based arbiter is shown in Figure 3. The memory controller tracks the command pattern that it receives from the processors and periodically switches among the arbiters depending on this pattern. Figure 3 depicts the overall process.

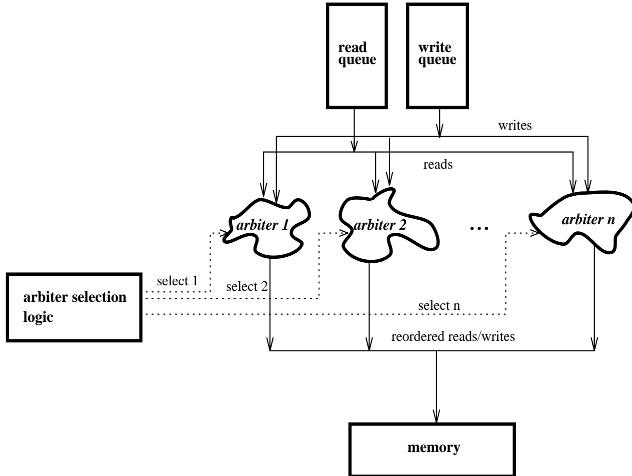

Figure 3. Overview of dynamic selection of arbiters in memory controller.

4.3.1 Detecting Memory Command Pattern

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 arbiter. 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.

5 Results

5.1 Benchmarks and Microbenchmarks

We first evaluate the performance of the three memory schedulers using the Stream [9] and NAS benchmarks [1]. The Stream benchmarks, which others have used to measure the sustainable memory bandwidth of systems [6, 13, 3, 16], consist of 4 simple vector kernels: Copy, Scale, Sum, and Triad. The NAS benchmarks are well known scientific benchmarks, which are fairly data-intensive.

We then use a set of 14 microbenchmarks, which allows us to explore a wider range of machine 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.

5.2 Schedulers Studied

We compare three memory arbiters. The first, *in-order*, implements the simple FIFO policy used by most general purpose memory controllers today. If implemented in a Power5 system, this arbiter would transmit memory commands from the reorder queues to the CAQ in the order in which they were received from the processors.

The second arbiter implements one of the policies proposed by Rixner et al. [12]. We refer to this as the *memoryless* arbiter because it does not use command history information. This arbiter avoids long bank conflict delays by selecting commands from the reorder queues that do not conflict with commands in DRAM. When there are multiple eligible commands, the oldest command is chosen.

The third arbiter is our *adaptive history-based scheduler*. Because bank conflicts cause much longer delays than rank or port conflicts, our scheduler implements the same simple approach to avoiding bank conflicts as the memoryless arbiter. Our scheduler then implements the adaptive history-based technique described in Section 4 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. Conceptually, our history-based approach could also be used to prioritize bank conflicts, as well, but to be effective on a system like the Power5, we would need to use much longer history lengths than we evaluate in this paper.

5.3 Simulation Methodology

To evaluate performance, we use a cycle-accurate simulator for the IBM Power5, which has been verified to within 1% of the performance of the actual hardware. This simulator, which is one of several simulators used by the Power5 design team, simulates both the processor and the memory system in one of two modes. The first mode can simulate arbitrary applications using execution traces, but it can only simulate a single-processor system. The second mode can simulate dual-processor systems that share a memory controller, but it can only simulate microbenchmarks whose behavior can be concisely described without execution traces.

We simulate the microbenchmarks in their entirety. To simulate our benchmarks, which have billions of dynamic instructions, we use uniform sampling, taking 50 uniformly chosen samples that each consist of 2 million instructions.

5.4 Simulation Parameters

We simulate a Power5 running at 1.6GHz. The Power5 gives priority to demand misses over prefetches, so the memory controller does not have to deal with these priorities.

Our simulator models all three levels of the cache. The L1D cache is 64KB with 4-way set associativity and the L1I cache is 128KB with 2-way set associativity. The L2 cache is 3×640KB in size, with 10-way set associativity and a line size of 128B. The off-chip L3 cache is 36MB.

We simulate the DDR2-266 SDRAM chips running at 266MHz. This DRAM is organized as 4 ranks, where each rank consists of 4 banks. Bank conflicts incur a 75ns delay and rank conflicts a 30ns delay.

We use three types of history-based arbiters. The first, $1R2W$, is optimized for data streams with twice as many Writes as Reads. The second, $1R1W$, is optimized for streams with equal numbers of Reads and Writes. The third, $2R1W$, is optimized for streams with twice as many Reads as Writes. These arbiters use history lengths of 2 and consider commands that Read or Write from either of two ports, so each arbiter uses a 16 state FSM.

The adaptive history-based arbiter combines these three history-based arbiters by using the $2R1W$ arbiter when the Read/Write ratio is greater than 1.2, by using the $1R1W$ arbiter when the Read/Write ratio is between 0.8 and 1.2, and by otherwise using the $1R2W$ arbiter. The selection of these arbiters is performed every 10000 processor cycles. (Our results were not very sensitive to this period, as long as the period was greater than about 100.)

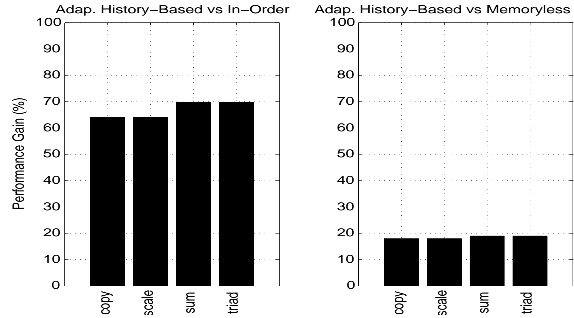

Figure 4. Performance comparison of the adaptive history-based approach against the in-order arbiter (left) and the memoryless arbiter (right) for the Stream benchmarks.

5.5 Benchmark Results

Figure 4 shows results for the Stream benchmarks. We see that the adaptive history-based arbiter improves execution time by 65-70% over the in-order arbiter and by 18-20% over the memoryless arbiter. Since Copy and Scale have both two Reads per one Write, their improvements are the same. Sum and Triad both have three Reads per Write and thus see the same improvements.

The NAS benchmarks provide a more comprehensive evaluation of overall performance. The top two graphs in Figure 5 show that improvements over the in-order method are between 6.6% and 21.4%, with a geometric mean of 10.9%. Improvements over the memoryless method are between 2.4% and 9.7%, with a geometric mean of 5.1%. The bottom two graphs show results when the CPU has 4 times the clock rate, showing that future systems will benefit more from these intelligent arbiters. Here, the geometric mean improvement is 14.9% over the in-order arbiter and 8.4% over the memoryless arbiter.

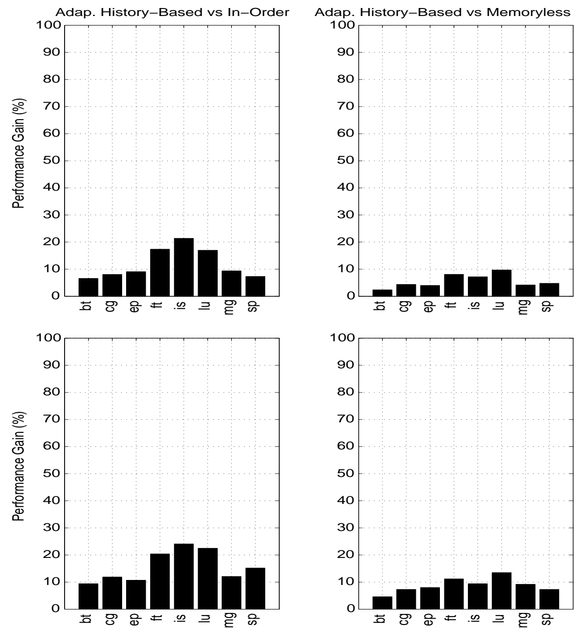

Figure 5. Performance comparison for the NAS benchmarks. The bottom graphs show results for a processor that is 4 times faster than the current IBM Power5.

5.6 Microbenchmark 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 6 shows that for these microbenchmarks, the adaptive history-based method yields performance results of 20-70% compared to in-order method and of 17-20% compared to memoryless method. The performance gain over in-order drops from the 60-65% range to the 45-50% range as the Read/Write ratio drops from 1.5 to 1.0. The gain over the memoryless approach is consistently in the 17-20% range across all microbenchmarks.

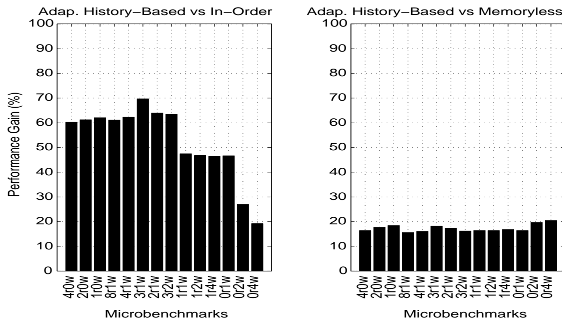

Figure 6. Performance comparison on our microbenchmarks.

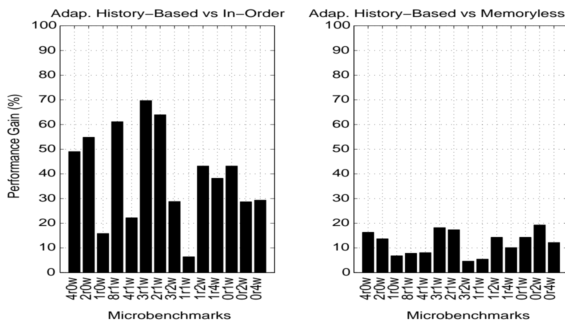

Figure 7. Performance comparison for an architecture with no data prefetching.

By comparing Figure 7 with Figure 6, 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. Figure 8 shows similar effects if we move to a one-processor system, again with prefetching turned off.

Figure 9 shows the performance difference of our three history-based approaches for each microbenchmark. Our adaptive selection mechanism chooses the best of these history-based arbiters in all cases except for the 3r2w benchmark, which has a 1.5 Read/Write ratio. For the 3r2w case, our adaptive mechanism will select the third arbiter instead of the second, yielding a 1% performance loss.

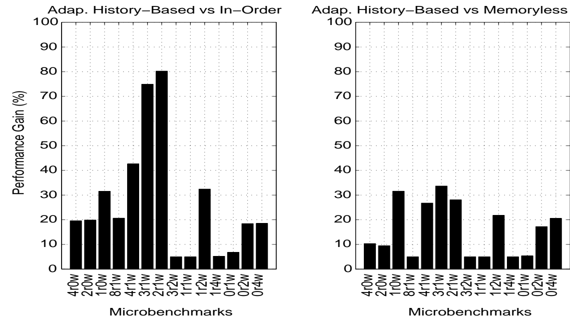

Figure 8. Performance comparison for a one-processor system and no prefetching.

In summary, our microbenchmark results show significant benefit for the adaptive history-based approach on a dual-processor system with prefetching. Features that increase memory traffic, such as dual-processors and prefetch units, increase the impact of good memory scheduling. The NAS benchmark results presented in Section 5.5 are conservative because they simulate a single-processor system with prefetching, instead of a dual-processor system. Finally, we see that the ability to choose from among 3 different history-based arbiters typically yields a few percentages of performance gain.

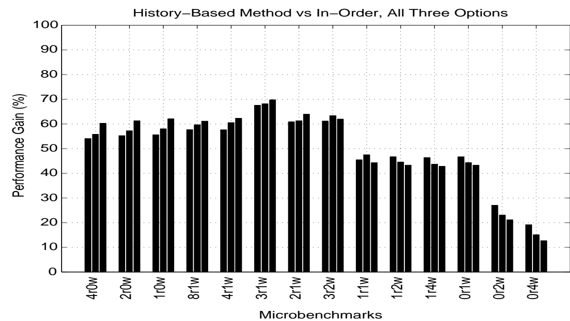

Figure 9. Performance comparison of our three history-based arbiters.

5.7 Understanding the Results

We now look inside the memory system to gain a better understanding of our results. The most direct measure of the quality of a memory controller is its impact on memory system utilization. Figure 10 shows a histogram of the number

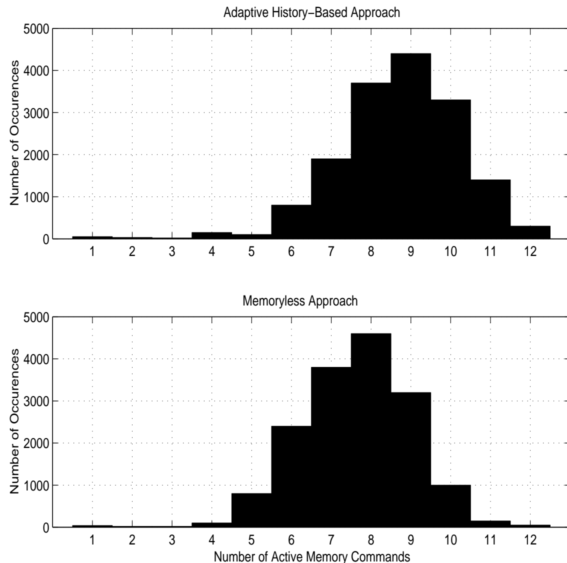

Figure 10. Utilization of the DRAM for the daxpy kernel.

of operations that are active in the memory system on each cycle. We see that when compared against the memoryless arbiter, our arbiter increases the average utilization from 8 to 9 operations per cycle. The x-axis goes up to 12 because the Power5's DRAM allows 12 memory commands to be active at once.

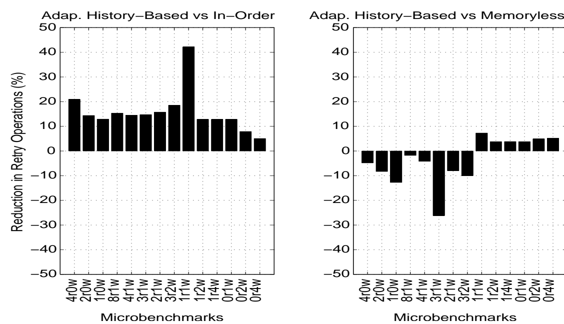

Figure 11. Comparison of retry rates.

Memory system utilization is also important when evaluating our results, because it is easier for an arbiter to produce performance improvements to a saturated system. We measured the utilization of the command bus that connects the memory controller to the DRAM, and we found 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 understand better 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 11 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.

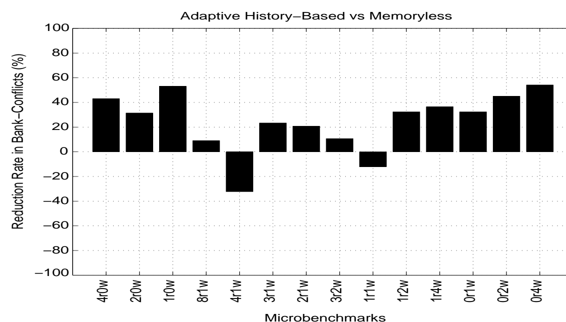

Figure 12. Comparison of the number of bank conflicts in the reorder queues.

A second bottleneck occurs when no operation in the reorder queues can be issued because of bank conflicts with previously scheduled commands. This bottleneck is a good indicator of arbiter performance, because a large number of such cases suggests that the arbiter has done a poor job of scheduling memory operations. Figure 12 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 two microbenchmarks, our method substantially reduces the number of such blocked operations.

A third bottleneck occurs when the reorder queues are empty, starving the arbiter of work. Even when the reorder queues are not empty, low occupancy in the reorder queues is bad because it reduces the arbiter's ability to schedule operations. In the extreme case, where the reorder queues hold no more than a single operation, the arbiter has no ability to reorder memory operations and instead simply forwards the single available operation to the CAQ. Figure 13 shows that our method significantly reduces the occurrences of empty reorder queues, indicating higher occupancy of

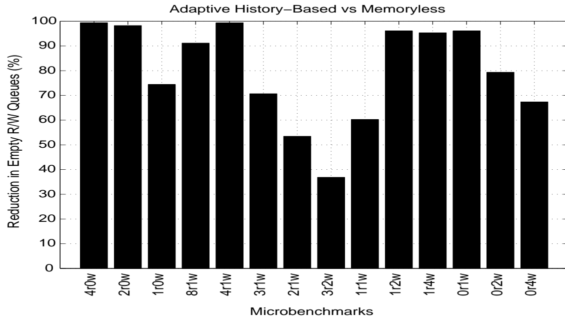

Figure 13. Reduction in the occurrences of empty reorder queues, which is a measure of the occupancy of the reorder queues.

these queues.

The final bottleneck occurs when the CAQ is full, forcing the arbiter to remain idle. Figure 14 shows that the adaptive history-based arbiter tremendously increases this bottleneck. The backpressure created by this bottleneck leads to larger reorder queues, which are advantageous because they give the arbiter a larger scheduling window.

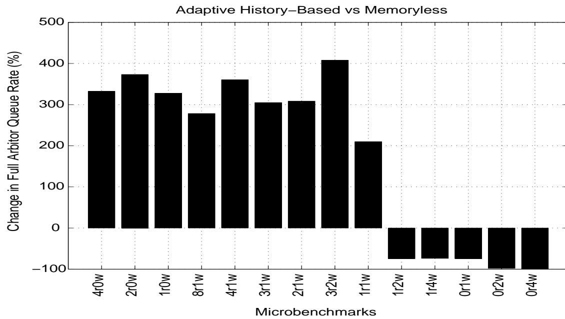

Figure 14. Increases in the occurrences where the CAQ is the bottleneck.

To test this theory, we conducted an experiment in which we increased the size of the CAQ. We found that as the CAQ length increased, the CAQ bottleneck decreased, the reorder queue occupancy fell, and the overall performance decreased.

In summary, our solution improves bandwidth by moving bottlenecks from outside the memory controller, where the arbiter 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 arbiter a larger number of memory operations to choose from. The result is a smaller number of bank conflicts and increased bandwidth.

5.8 Evaluating Our Solution

Perfect DRAM Results. We have so far evaluated our solution by comparing against previous solutions. To see how much room there is for further improvement, we compare the performance of our new arbiter against a perfect DRAM in which there are no hardware hazards. We find that for our benchmarks, our solution achieves 95-98% of the performance of the perfect DRAM.

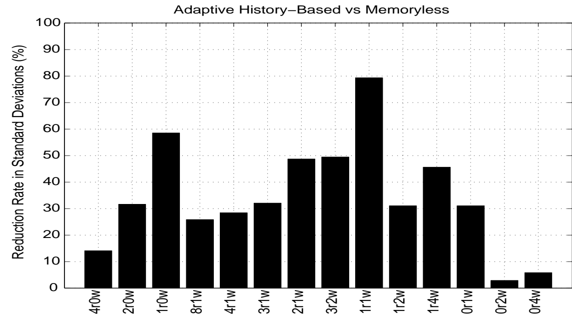

Figure 15. Ratio of standard deviations for 16-different address offsets.

Sensitivity to 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 15 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.

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.

Queue Length Effects. The results that we have presented have all used the same queue lengths as the Power5. As mentioned in Section 5.7, we have studied the effects of varying the CAQ length and found that the Power5's length of four produced the best results. We also experimented with larger reorder queues, but these had only small effects on performance. Moreover, longer reorder queues would be extremely expensive in terms of silicon, because each entry of the Write Queue holds the addresses and contents of an L2 cache line (or part of an L3 cache line), and the Read Queue devotes considerable space to buffers to return the results of Reads.

6 Conclusions

This paper has 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 arbiter 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 arbiter that partially satisfies both goals. Finally, because we cannot know the actual command-pattern *a priori*, we implement three history-based arbiters—each tailored to a different command pattern—and we dynamically select from among these three arbiters based on the observed ratio of Reads and Writes.

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%. Our solution is also quite effective, achieving 95-98% of the IPC of an idealized DRAM that never experiences a hardware hazard. In terms of benchmark performance, our technique improves IPC for the Stream benchmarks by 63% over in-order scheduling and 19.1% over memoryless scheduling. For the NAS benchmarks, the improvements are 10.9% over in-order scheduling and 5.1% over memoryless scheduling.

Finally, 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 arbiter more operations to choose from when scheduling operations. We have also explored the effects of various external factors, such as increased clock rate, prefetching, and the use of multiple processors. As expected, the benefit of good memory scheduling increases as the memory traffic increases.

Acknowledgments. We thank Bill Mark, E Lewis, and the anonymous referees for their valuable comments on previous drafts of this paper. We also thank Steve Dodson for his technical comments. This work was supported by NSF grants ACI-9984660 and ACI-0313263, an IBM Faculty Partnership Award, and DARPA contract F33615-03-C-4106.

References

[1] 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.

[2] 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 *Proceedings of the The Fifth International Symposium on High Performance Computer Architecture*, page 70. IEEE Computer Society, 1999.

[3] 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.

[4] 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.

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

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

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

[8] 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.

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

[10] 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 Trans. Comput.*, 49(11):1255–1271, 2000.

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

[12] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, June 2000.

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

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

[15] 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 *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 372–381. ACM Press, 1992.

[16] 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 Press, 2002.

[17] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the 14th International Conference on Supercomputing*, pages 167–175. ACM Press, 2000.