Memory Scheduling for Modern Microprocessors

IBRAHIM HUR
The University of Texas at Austin and IBM Corporation
and
CALVIN LIN
The University of Texas at Austin

The need to carefully schedule memory operations has increased as memory performance has become increasingly important to overall system performance. This article describes the adaptive history-based (AHB) scheduler, which uses the history of recently scheduled operations to provide three conceptual benefits: (1) it allows the scheduler to better reason about the delays associated with its scheduling decisions, (2) it provides a mechanism for combining multiple constraints, which is important for increasingly 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.

We have previously evaluated this scheduler in the context of the IBM Power5. When compared with the state of the art, this scheduler improves performance by 15.6%, 9.9%, and 7.6% for the Stream, NAS, and commercial benchmarks, respectively. This article expands our understanding of the AHB scheduler in a variety of ways. Looking backwards, we describe the scheduler in the context of prior work that focused exclusively on avoiding bank conflicts, and we show that the AHB scheduler is superior for the IBM Power5, which we argue will be representative of future microprocessor memory controllers. Looking forwards, we evaluate this scheduler in the context of future systems by varying a number of microarchitectural features and hardware parameters. For example, we show that the benefit of this scheduler increases as we move to multithreaded environments.

Categories and Subject Descriptors: C.1.1 [**Processor Architectures**]: Single Data Stream Architectures

General Terms: Performance

Additional Key Words and Phrases: Memory system performance, memory scheduling, adaptive history-based scheduling

This research was supported by NSF grants ACI-9984660 and CNS-0509354 and an IBM Faculty Partnership Award.
Authors' addresses: I. Hur and C. Lin, The University of Texas at Austin, Department of Computer Sciences, 1 University Station C0500, Austin, TX 78712; email: {ihur,lin}@cs.utexas.edu.

ACM Reference Format:
Hur, I. and Lin, C. 2007. Memory scheduling for modern microprocessors. ACM Trans. Comput. Syst. 25, 4, Article 10 (December 2007), 36 pages. DOI = 10.1145/1314299.1314301 http://doi.acm.org/10.1145/1314299.1314301

1. INTRODUCTION

Memory bandwidth is an increasingly important aspect of overall system performance. Early work in this area has focused on streaming workloads, which place great stress on the memory system. Early work has 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 [Cragon 1996], skewing array addresses [D. T. Harper and Jump 1986; Gao 1993], and otherwise [McKee 1993, 1995; McKee et al. 1998; Carter et al. 1999; McKee et al. 2000] attempting to spread a stream of regular memory accesses across the various banks of DRAM.

Previous work [Valero et al. 1992; Peiron et al. 1995] describes a method of dynamically reordering memory commands so that the banks are accessed in a strict round-robin fashion. More recent work [Rixner et al. 2000] evaluates a set of 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 multiprocessors 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 traffic. For example, simultaneous multithreading 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 provide more parallelism, and hence more bandwidth, 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 three 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 the other physical substructures of increasingly complex DRAM's. Third, as memory controllers themselves become more complex, it also becomes important for schedulers to avoid bottlenecks within the memory controller itself.

To understand this third problem, consider the execution of the daxpy kernel on the IBM Power5's memory controller. As shown in Figure 1, this memory

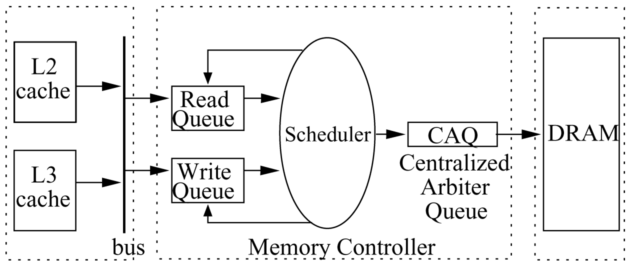

Fig. 1. The Power5 memory controller.

controller sits between the L2/L3 caches and DRAM; it has two reorder queues (the Read Queue and the Write Queue), a scheduler that selects operations from the reorder queues, and a FIFO Centralized Arbiter Queue, which buffers operations that are sent to DRAM. The daxpy kernel performs two Reads for every Write, so 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 article, we describe a new approach—adaptive history-based 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.

We have previously [Hur and Lin 2004, 2006; Hur 2007] introduced the adaptive history-based (AHB) memory scheduler and evaluated it in the context of the IBM Power5. Since then, 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 its utility will become more or less important to future systems, which we study by altering various architectural parameters of the processor, the memory system, and the memory controller. For example, is the AHB scheduler effective for multithreaded and multicore 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 article, we answer these questions and others to demonstrate the flexibility and robustness of the AHB scheduler, evaluating it in a variety of situations. In particular, this article makes the following contributions:

—We identify three aspects of the memory scheduling problem: (1) How to schedule bank operations, (2) what to do when a bank conflict arises, and (3) how to avoid bottlenecks in the Read and Write reorder queues? Most previous work focuses on the first aspect; Rixner et al. [2000] introduce heuristics for dealing with all three aspects; and we study this space more thoroughly, presenting evidence that a simple solution to the second question obviates the need to answer the first question.

—We show that multithreaded workloads increase the performance benefit of the AHB scheduler. This result may be surprising because multithreading would seem to defeat the scheduler'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.

—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 evaluate the AHB scheduler on a set of commercial benchmarks. When compared with the state of the art on a two processor system each running two threads, the AHB scheduler improves performance between 6.5% and 9.5%.

—We tune the AHB scheduler 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 the benefits of the AHB scheduler tend to be more valuable in future systems. In addition to the multithreading results, we show that the AHB scheduler performs well as we alter various memory controller parameters, DRAM parameters, and system parameters.

—We present results for the more advanced DDR2-533 SDRAM and with the processor frequency of the most recent Power5+ systems.[1]

—We explore the effects of varying parameters of the memory scheduler itself. We find that the 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 of the others.

This article is organized as follows. The next section summarizes prior work. Section 3 describes the IBM Power5 architecture and its memory system, which is useful for understanding our solution, which we present in Section 4. We

[1]The results in this article differ slightly from those presented previously [Hur and Lin 2004, 2006], since this article uses the architectural parameters of the Power5+ instead of the Power5. In addition to various architectural differences, the Power5+ has a higher processor frequency and uses faster DRAM chips.

present our experimental methodology in Section 5, followed by our experiment evaluation in Sections 6 and 7. Finally, we discuss the hardware costs of our solution in Section 8, and we provide concluding remarks in Section 9.

2. RELATED WORK

Memory systems are typically organized as multiple banks that can be accessed simultaneously. By implementing some sort of interleaving [Cragon 1996], banked memory systems can considerably improve bandwidth, as long as bank conflicts—concurrent accesses to the same bank—are avoided. Techniques for eliminating bank conflicts have been studied for several decades, and the techniques can be divided into two broad classes: *static* approaches and *dynamic* approaches.

Static bank conflict avoidance techniques, such as skewing [D. T. Harper and Jump 1986; Gao 1993], prime memory systems [Raghavan and Hayes 1990; Rau 1991], or compiler transformations [Moyer 1993] attempt to arrange the order of memory commands to minimize bank conflicts. These static methods are effective for reducing only intra-stream bank conflicts, that is, conflicts caused by a single stream.

Dynamic conflict avoiding techniques [Valero et al. 1992; Peiron et al. 1995; Carter et al. 1999] can alleviate both intra- and inter-stream (conflicts among multiple streams) bank conflicts. As an example, the Impulse memory system [Carter et al. 1999] improves memory performance by dynamically remapping physical addresses, but it requires modifications to the applications and the operating system.

Hardware techniques that do not require operating system support include the various heuristics that have been proposed to reorder memory commands, including a memory reordering technique [Valero et al. 1992; Peiron et al. 1995] 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. This approach can only eliminate bank conflicts if the requests are fairly uniformly distributed among banks.

Because of their technical context—bank conflicts were the only issue for older memory systems—all of the preceding studies focus exclusively on bank conflicts. Many of these approaches are complementary to ours, so an AHB scheduler could use these methods as another optimization criteria. For example, to break ties when there are multiple commands in the reorder queues to choose from, an AHB scheduler could select the command that matches a predetermined sequence of bank accesses rather than choosing the oldest command.

McKee et al. [McKee 1993, 1995; McKee et al. 1998, 2000], Mathew et al. [Mathew et al. 2000a, 2000b; Mathew 2000], and Rixner et al. [Rixner et al. 2000; Khailany et al. 2001; Rixner 2004] were the first to recognize the importance of moving beyond bank conflict avoidance.

McKee et al.'s Stream Memory Controller (SMC) [McKee 1993, 1995; McKee et al. 1998, 2000] maximizes bandwidth for streaming applications. The SMC design includes three main components: stream buffers, caches, and a memory command scheduler. The compiler detects streams in the code and generates noncacheable memory requests that bypass caches at run time and go directly to the stream buffers. The memory scheduler dynamically selects commands from either the stream buffers or from the caches. The SMC system addresses two issues in reordering commands: selecting the memory bank to which the next access to schedule, and selecting the buffer which has a command for that particular bank. In the SMC work, McKee et al. focus on three aspects of memory system performance: hitting the hot row, avoiding bank conflicts, and avoiding switching between Reads and Writes. They examine and evaluate various dynamic ordering heuristics, but they do not propose a specific algorithm.

Mathew et al. [Mathew et al. 2000a, 2000b; Mathew 2000] present a heuristic, which they evaluate using streaming workloads, to maximize row hits in open-page memory systems. Rixner et al. [Rixner et al. 2000; Khailany et al. 2001; Rixner 2004] explore the benefits of various heuristics for reordering accesses for both streaming workloads [Rixner et al. 2000; Khailany et al. 2001] and Web applications [Rixner 2004]. Each of these heuristics reorders 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.

3. BACKGROUND

This section describes the technical context for our work. We first describe the IBM Power5 chip, including its memory controller; we then describe the Power5 memory system; and finally, we argue why other modern memory controllers are likely to have organizations that are similar to that of the Power5.

3.1 The IBM Power5

The IBM Power5 [Clabes et al. 2004; Kalla et al. 2004] is the successor to the Power4 [Tendler et al. 2002]. 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 the L2 cache, and from the L2 cache to the L1 cache.

Each chip has a single memory controller with two reorder queues: a Read Reorder Queue and a Write Reorder Queue (see Figure 1). 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. A scheduler selects an appropriate command from these queues to place in the Centralized 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 scheduler 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 DDR2-533 SDRAM chips [Micron 2004] (our earlier work [Hur and Lin 2004] uses the older DDR2-266 chips), which have 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 5.3 for details), and the costs of these delays depends on whether the operations are Reads or Writes. In this system, bank conflict delays are an order of magnitude greater than the delays introduced by rank or port conflicts.

3.3 Modern Memory Controller Design

To understand the relevance of our work to other systems, it is worth asking whether the Power5's memory controller design is representative of other modern memory controllers. Thus, this section takes a closer look at the Power5's memory controller in an attempt to understand whether it is likely to be representative of other modern memory controllers.

We begin by examining the decision to use separate Read and Write reorder queues instead of a single unified reorder queue. A single reorder queue allows for a simpler scheduler, one that does not need to balance the fullness of the two queues. However, there are two advantages of segregated queues. (1) The decoding logic required to select a command from the queue depends on the length of the queue, so for a given capacity, a single longer queue requires more logic and greater decoding time than two shorter queues. For example, for a particular implementation of the Power5, it was not possible to increase the number of entries in the Read reorder queue by 20% because the increased logic would prevent the decoding from completing in a single clock cycle. In other words, the Read reorder queue was already as large as it could be. (2) The elements of the Read and Write reorder queues are processed differently, so it takes considerably less hardware to implement two separate queues, each specialized to its own use, than to implement a single queue in which each entry can perform either duty.

Most vendors do not provide details of their memory controller designs, but since 2000, there have been a number of patents [Foster 2000; Harriman et al. 2000; Larson 2001; Kessler et al. 2003; Jenne and Olarig 2003; Harris 2003; McGee and Chau 2005; Sah et al. 2006] by vendors such as Sun Microsystems, Intel, Hewlett-Packard, Compaq, and Micron, that refer to segregated Read and Write reorder queues, so we conclude that other vendors have made the same design choices as IBM to use segregated reorder queues.

As we look to the future, we expect segregated reorder queues to become more common. With multiple threads and multiple cores on a chip, the demand for memory bandwidth will continue to increase. To satisfy this demand, there is pressure to increase reorder queue capacity, and because of constraints on decoding latency, the only way to increase capacity is to use multiple queues. In fact, we believe that future memory controllers will see more than two reorder queues, perhaps including separate queues for prefetches, and perhaps further segregating the Read reorder queues and the Write reorder queues.

We now turn to the third queue, the CAQ. The primary benefit of this FIFO queue is to allow a greater number of commands to reside in the memory controller so that the CPU does not have to stall as often when DRAM bandwidth cannot keep up with memory command bandwidth. Of course, this queue does not have to be a FIFO queue, but the Power5 designers chose such a queue to reduce hardware complexity.

We conclude that the Power5's memory controller is likely to be representative of other modern memory controllers in its use of multiple internal queues, and we believe that future memory controllers will become increasingly complex as more queues are added. This trend will complicate the memory scheduler's task, as it will need to avoid multiple internal bottlenecks.

4. 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 a 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; the AHB scheduler then observes the recent command pattern and periodically chooses the most appropriate history-based scheduler.

4.1 History-Based Schedulers

This section describes the basic structure of history-based schedulers. Similar to branch predictors, which use the history of the previous branches to make predictions [Cragon 1996], 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 a scheduler that uses a history length of three. Assuming that each command is either a Read or a Write operation to either port number 0 or 1, 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 $4 \times 4 \times 4 = 64$. In Figure 2 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.

4.2 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 a scheduler 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 DRAM. 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.

4.2.1 *Optimizing for the Command Pattern.* The command-pattern-scheduler algorithm generates state transitions for a scheduler that selects 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

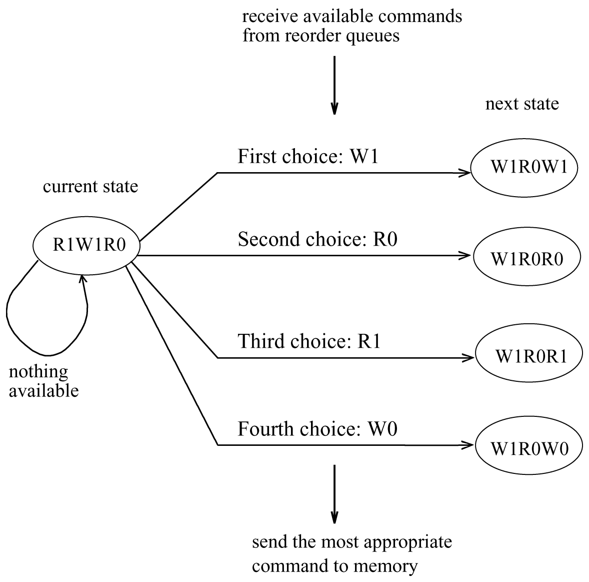

Fig. 2. Transition diagram for the current state $R1W1R0$. Each available command type has a different selection priority.

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 a 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.

COMMAND-PATTERN-SCHEDULER (n)

```
1 n is the history length
2 for all command sequences of size n
3   do r_old ← Read/Write ratio of the command sequence
4      for each possible next command
5        do r_new ← Read/Write ratio
6
7      if r_old < ratio of the scheduler (x/y)
8        then Read commands have higher priority.
9        else Write commands have higher priority.
10
11     if there are commands with equal r_new
12       then Sort them with respect to expected latency.
13            Pick the command with the minimum delay.
14
15     for each possible next command
16       do Output the next state in the FSM.
```

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.

4.2.2 *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 c_{new} and the previous n commands that were sent to memory. We refer to the previous n commands as $c_1, c_2, \ldots, c_n$, where c_1 is the most recently issued command.

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, $f\,cost_{i..n}$, such that

$$f\,cost_i(c_{new}) = max(f_j(c_i, c_{new})) - (i-1),$$

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 separate the issue

EXPECTED-LATENCY-SCHEDULER (n)

```
1 n is the history length
2 for all command sequences of size n
3    do for each possible next command
4          do Calculate the expected latency, $T_{delay}$.
5
6       Sort possible commands with respect to $T_{delay}$.
7
8       for commands with equal expected latency value
9          do Use Read/Write ratios to make decisions.
10
11      for each possible next command
12         do Output the next state in the FSM.
```

of command c_i and c_{new}. Thus, the expected latency that will be introduced by sending c_{new} is

$$T_{delay}(c_{new}) = max(f\,cost_{1..n}(c_{new})).$$

The expected-latency-scheduler algorithm generates an 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.

4.2.3 *A Probabilistic Scheduler Design Algorithm.* To combine our two optimization criteria, the probabilistic-scheduler algorithm 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:

```
PROBABILISTIC-SCHEDULER ( )
1 if random_number < threshold
2    then command-pattern-scheduler
3    else expected-latency-scheduler
```

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 and program dependent and should be determined experimentally. (See Section 6.2 for more details.)

4.3 Adaptive Selection of Schedulers

Our adaptive history-based scheduler is schematically shown in Figure 3. Each of the *n* schedulers is a history-based scheduler that is tuned for a specific mixture of Read and Write commands. The memory controller tracks the mixture of Reads and Writes that it receives from the processors and periodically switches among the schedulers depending on this mixture.

4.3.1 *Detecting Memory Command Pattern.* To select one of the history-based schedulers, the AHB scheduler assumes the availability of three counters: *Rcnt* and *Wcnt* count the number of Reads and Writes received from the processor, and *Pcnt* provides the period of adaptivity. Every *Pcnt* 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 *Pcnt* 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 values of *Rcnt* and *Wcnt*. The values of *Rcnt* and $Wcnt$ are set to zero when *Pcnt* becomes zero.

5. METHODOLOGY AND EXPERIMENTAL SETUP

In this section, we describe the benchmarks, simulation methodology, and simulations parameters that we use in evaluating the memory schedulers.

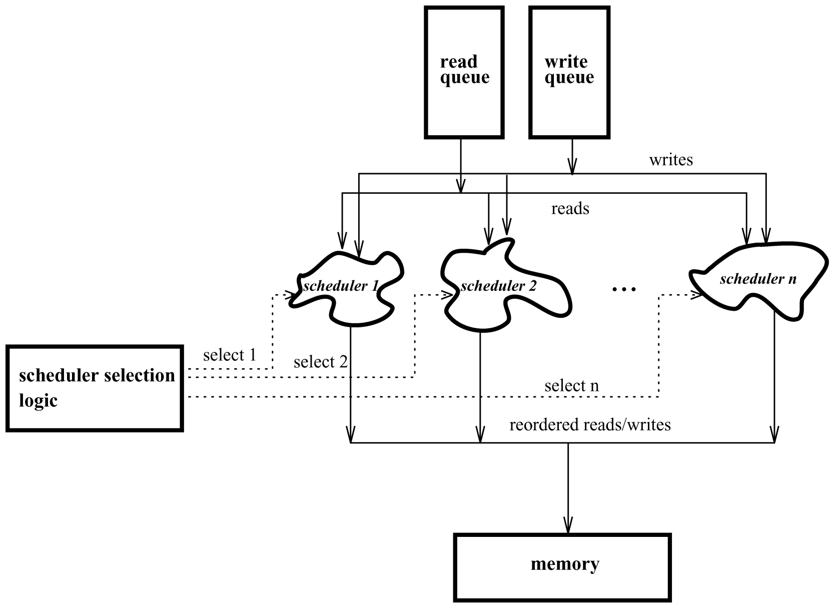

Fig. 3. Overview of adaptive selection of schedulers in memory controller.

5.1 Benchmarks and Microbenchmarks

We evaluate the performance of the memory schedulers using both technical and non-technical benchmarks. For technical benchmarks, we use the Stream [McCalpin 1995] and NAS [Bailey et al. 1994] benchmarks. For non-technical benchmarks, we use a set of internal IBM commercial workloads. We also create a set of microbenchmarks for detailed analysis of the memory controller.

The first set of benchmarks measure streaming behavior. The Stream benchmarks, which others have used to measure the sustainable memory bandwidth of systems [Cvetanovic 2003; Scott 1996], 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 Stream2 benchmarks to create the extended Stream benchmarks, which consist of seven vector kernels. We list these kernels in Table I and, for simplicity, we refer to them collectively as the Stream benchmarks in the rest of this article.

The second set of benchmarks, the NAS (Numerical Aerodynamic Simulation) benchmarks (see Table II), is a group of eight programs developed by NASA. 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 exist parallel and serial implementations of the various sizes of the NAS benchmarks. In this study, we use a serialized version of class B.

Table I. The Extended Set of Stream Benchmarks

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 II. The NAS Benchmarks

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

The third set of benchmarks consists of four commercial server applications, namely, *tpcc*, *trade2*, *cpw2*, and *sap*. 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; and sap is a database workload.

Finally, we use a set of microbenchmarks, which allows us to study the performance of dual-processor systems (see Section 5.2) and which allows us to explore the detailed behavior of the various memory schedulers. Each of the microbenchmarks uses a different Read/Write ratio, and each is named $xRyW$, indicating that it has x Read streams and y Write streams. Table III shows the 14 microbenchmarks that we use. These microbenchmarks represent most of the data streaming patterns that we expect to see in any given epoch.

5.2 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 [Hur 2006]. This simulator attempts to model every detail of the hardware, as evidenced by the size of the simulator, which is on the order of one million lines of high-level language code. This simulator, which is one of several used by the Power5 design team, simulates both the processor and the memory system in one of three distinct modes. The first mode can simulate arbitrary applications using execution traces, and it can simulate a single-processor system with one or two threads. We use this mode to simulate the Stream and the NAS benchmarks. The second mode can simulate dual-processor systems that share

Table III. Microbenchmarks

Benchmark	Description	Benchmark	Description
4r0w	4 Read Streams	3r2w	3 Read and 2 Write Streams
2r0w	2 Read Streams	1r1w	1 Read and 1 Write Streams
1r0w	1 Read Streams	1r2w	1 Read and 2 Write Streams
8r1w	8 Read and 1 Write Streams	1r4w	1 Read and 4 Write Streams
4r1w	4 Read and 1 Write Streams	0r1w	1 Write Streams
3r1w	3 Read and 1 Write Streams	0r2w	2 Write Streams
2r1w	2 Read and 1 Write Streams	0r4w	4 Write Streams

a memory controller, but it can only simulate microbenchmarks whose behavior can be concisely described without execution traces, and it cannot simulate multiple threads. We use this mode to gather dual-processor results for our microbenchmarks. The third mode uses statistical information collected from trace based runs to simulate an arbitrary number of processors with single or multiple threads. The rate of the commands coming from the processors to the memory controller matches the actual run, but the low-order bits of addresses are randomized. We use this third mode to simulate commercial benchmarks.

We simulate the microbenchmarks in their entirety. To simulate Stream and NAS benchmarks, which have billions of dynamic instructions, we use uniform sampling, taking 50 uniformly chosen samples that each consist of 2 million instructions. We simulate commercial benchmarks for 10 million processor cycles, which we observed to be sufficient to reach steady-state.

We choose to not use SimPoint [Sherwood et al. 2001] for two reasons. First, SimPoint has been designed to identify representative samples of long execution traces where behavior is defined by CPI. We have evidence that SimPoint requires modification to produce representative traces if performance is defined by memory behavior. Second, as a practical matter, it would take an enormous amount of time to run SimPoint on our benchmarks to produce the representative traces.

5.3 Simulation Parameters

We simulate a Power5 running at 2.132 GHz. Our simulator models all three levels of the cache. The L1D and L1I caches are 64 KB with 2-way set associativity, and the L2 cache is a 3×640KB shared cache with 10-way set associativity and a line size of 128B. The optional off-chip L3 cache is a 36 MB victim cache. We simulate the DDR2-533 SDRAM chips running at 533 MHz. This DRAM is organized as 4 ranks, where each rank consists of 4 banks.

We use three types of history-based schedulers. 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 schedulers use history lengths of 2 and consider commands that read or write from either of two ports, so each scheduler uses a 16-state FSM. The adaptive history-based scheduler combines these three history-based schedulers by using the $2R1W$ scheduler when the Read/Write ratio is greater than 1.2, by using the $1R1W$ scheduler when the Read/Write ratio is between 0.8 and 1.2, and by otherwise using the $1R2W$ scheduler. The selection of these schedulers is performed every 10,000 processor cycles. We assign 30% priority to the command pattern criterion and 70% to the expected latency criterion.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the AHB scheduler and compare its performance against two previous scheduling approaches. The first is the in-order scheduler, which we believe is used in most current processors. The second represents the state of the art as defined by the literature, which we empirically identify in the next subsection. We then compare the performance of these two schedulers and the AHB scheduler using the Stream, NAS, and commercial benchmarks. 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.

6.1 Evaluating Previous Approaches

To assess the prior state-of-the-art, we identify three main features of previously proposed memory controllers that are relevant to DDR2 SDRAM chips, namely, (1) the handling of bank conflicts, (2) the bank scheduling method, and (3) the priorities for Reads and Writes. We then compare the space of these three design features using the Stream benchmarks. We now describe each of these three features in more detail.

The first feature specifies the scheduler's behavior when the selected command has a bank conflict, of which two choices have been proposed: the scheduler can hold the conflicting command in the reorder queues until the bank conflict is resolved, or 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: FIFO, LRU, and Round-Robin. The first, FIFO, implements the simple first-in first-out 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, FIFO 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 are more than one such commands, the scheduler will switch to the FIFO 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

Table IV. Performance (in CPI) of Previous Scheduling Approaches as Measured on the Stream Benchmarks

Bank Hold, Scheduler, Priority	daxpy	copy	scale	vsum	triad	fill	sum	mean
No Hold, FIFO, Equal (in-order)	1.987	3.142	2.131	2.001	2.005	2.265	0.851	1.938
No Hold, FIFO, Read	1.260	2.164	1.474	1.542	1.561	2.121	0.650	1.448
No Hold, LRU, Equal	0.895	1.557	1.072	1.060	1.061	1.783	0.527	1.067
No Hold, LRU, Read	0.856	1.467	1.006	1.003	1.004	1.825	0.864	1.105
No Hold, Round-Robin, Equal	1.118	1.812	1.242	1.244	1.246	2.007	0.555	1.233
No Hold, Round-Robin, Read	1.119	1.776	1.211	1.213	1.219	2.018	0.555	1.219
Hold, FIFO, Equal	0.866	1.475	1.014	1.028	1.032	1.798	0.515	1.035
Hold, FIFO, Read (historyless)	0.825	1.487	1.020	0.978	0.977	1.775	0.517	1.014
Hold, LRU, Equal	0.855	1.507	1.038	1.017	1.017	1.782	0.560	1.047
Hold, LRU, Read	0.846	1.463	0.999	0.982	0.980	1.800	0.515	1.014
Hold, Round-Robin, Equal	0.808	1.463	1.001	0.956	0.957	1.786	0.569	1.014
Hold, Round-Robin, Read (best)	0.824	1.478	1.013	0.973	0.969	1.783	0.521	1.011

LRU approach, if there are multiple commands to the bank, the scheduler uses a FIFO policy and selects the oldest such command.

The third design feature describes how commands are selected from the 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.

To translate these design features into a concrete scheduler, we prioritize 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 approaches for prioritizing Reads and Writes, and two choices for handling bank conflicts, we evaluate a total of twelve points in the design space. Table IV illustrates that out of the three criteria, the bank hold policy has the greatest effect—up to 46%—on performance. We see that any method that does not holds commands with bank conflicts is better than its counterpart that does not hold the commands. Among the six approaches that hold for bank conflicts, the Read/Write priority is generally more important than the bank scheduling method, as the performance gains from holding banks obviate the need for a complicated bank scheduling method. Among the six approaches that do not hold for bank conflicts, the effect of the bank scheduling policy can be significant, as high as 45%, with LRU being the best. In terms of implementation complexity, FIFO bank scheduling is the simplest. Therefore, we identify "Hold, FIFO, Read Priority" approach, which we call *historyless*, as the first baseline for our study. Note that in our previous work [Hur and Lin 2004, 2006], we used the term *memoryless* for "Hold, FIFO, Equal Priority" method, which is a slightly inferior method.

6.1.1 *The In-order and AHB Schedulers in Context.* We can now describe the in-order and AHB schedulers in relation to these three design features. The in-order scheduler is equivalent to the "No Hold, FIFO, Equal Priority" scheduler. The AHB scheduler requires a bit more explanation: It holds the commands in the reorder queues if there is a bank conflict; it then uses 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, the adaptive history-based approach is used to handle rank and port conflicts but not bank conflicts. The adaptive history-based approach also combines bank scheduling with and Read/Write priorities, so it eliminates the need to determine thresholds for priority selection. In short, the AHB scheduler uses a simple mechanism for deciding how to deal with bank conflicts, and it introduces a single mechanism for implementing the second and third design features.

6.2 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 the daxpy benchmark and assuming that there are two active threads on one processor.

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

6.2.2 *Epoch Length.* We vary epoch length from 100 to 10,000 processor cycles. Table V(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.

6.2.3 *Ratio for Optimization Criteria.* The AHB scheduler optimizes for two criteria, namely the expected latency and the command pattern. As we describe in Section 4, our approach combines two criteria probabilistically by giving weights to each criterion. Table V(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%.

6.3 Benchmark Results

We now present simulation results for the AHB, in-order, and historyless schedulers using the Stream, NAS, and commercial benchmarks. For the Stream and

Table V. Tuning of the AHB Scheduler

(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 (processor cycles)	CPI
100	0.712
500	0.703
1,000	0.694
5,000	0.696
10,000	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

NAS benchmarks, our simulator allows us to simulate one or two threads on one processor. For the commercial benchmarks, we can simulate one or two threads on either single or dual core systems.

We first compare the single thread performance of the three schedulers for the Stream benchmarks (see Table VI). The geometric means of the performance benefit of the AHB scheduler over the in-order and the historyless schedulers is 52.8% and 11.8%, respectively. For two threads on a processor, adaptive history-based scheduler improves execution time by an average of 55.9% over the in-order scheduler and 15.6% over the historyless scheduler.

Our second set of results are for the NAS benchmarks, which provide a more comprehensive evaluation of overall performance. Table VII shows that for the single thread experiments, the average improvement of the AHB scheduler over the in-order method is 17.6%, and the average improvement over the historyless method is 6.1%. In the SMT experiments, we use two threads of the same application, and the AHB scheduler improves performance by 26.3% and 9.9% over the in-order and historyless schedulers, respectively.

Finally, in Table VIII, 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

Table VI. Performance Comparison (in CPI) of the In-Order, Historyless, and AHB Schedulers as Measured on the Stream Benchmarks

Benchmark	in-order	historyless	AHB	gain over in-order (%)	gain over historyless (%)
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
geometric mean	2.040	1.090	0.962	52.8	11.8
Two Threads on One Processor					
daxpy	1.987	0.825	0.696	65.0	15.6
copy	3.142	1.487	1.212	61.4	18.5
scale	2.131	1.020	0.833	60.9	18.3
vsum	2.001	0.978	0.837	58.2	14.4
triad	2.005	0.977	0.838	58.2	14.2
fill	2.265	1.775	1.518	33.0	14.5
sum	0.851	0.517	0.447	47.5	13.5
geometric mean	1.939	1.013	0.855	55.9	15.6

single threaded case on a single processor, the AHB scheduler has, on the average, a 13.4% performance advantage over the in-order scheduler and a 3.1% advantage over the historyless scheduler. When there are two total threads executing, the AHB scheduler's advantage increases to 28.2% and 4.5% on a single core system, and to 33.5% and 5.6% on a dual core system. For two threads running on each of two processors, the gain from the AHB scheduler is 51.8% over the in-order scheduler and 7.6% over the historyless scheduler.

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

6.4 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 4 shows that for these microbenchmarks, the adaptive history-based method improves performance by 20–70% compared to the in-order scheduler and by 17–20% compared to the historyless scheduler.

The most direct measure of the quality of a memory controller is its impact on memory system utilization. Figure 5 shows a histogram of the number of

Table VII. Performance Comparison (in CPI) of the In-Order, Historyless, and AHB Schedulers as Measured on the NAS Benchmarks

Benchmark	in-order	historyless	AHB	gain over in-order (%)	gain over historyless (%)
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	14.1	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
geometric mean	1.852	1.626	1.526	17.6	6.1
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
geometric mean	1.694	1.385	1.248	26.3	9.9

operations that are active in the memory system on each cycle. We see that when compared against the historyless scheduler, the AHB 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.

Memory system utilization is also important when interpreting our other results, because it is easier for an 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 is about 65% for the Stream benchmarks and about 13%, on average, for the NAS benchmarks. We conclude that the memory system is not saturated for our workloads.

6.4.1 *Bottlenecks in the System.*

To better understand why the AHB scheduler 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

Table VIII. Performance Comparison (in CPI) of the In-Order, Historyless, and AHB Schedulers as Measured on the Commercial Benchmarks

Benchmark	in-order	historyless	AHB	gain over in-order (%)	gain over historyless (%)
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
geometric mean	13.995	12.514	12.124	13.4	3.1
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
geometric mean	10.516	7.907	7.554	28.2	4.5
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
geometric mean	9.805	6.906	6.520	33.5	5.6
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
geometric mean	9.048	4.715	4.359	51.8	7.6

demands. Figure 6 shows that the adaptive history-based method, compared to the historyless approach, sometimes increases and sometimes decreases retry rates. Thus, we see that there is no clear correlation between the retry rate and performance.

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 scheduler performance, because a large number of such cases suggests that the scheduler has done a poor job of scheduling memory operations. Figure 7 compares the total number of such blocked commands for the AHB and historyless schedulers. This graph considers only cases where the reorder queues are the bottleneck; that is, all operations in the reorder queues are blocked even though the CAQ has empty slots. We see that except for four microbenchmarks, the AHB scheduler 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

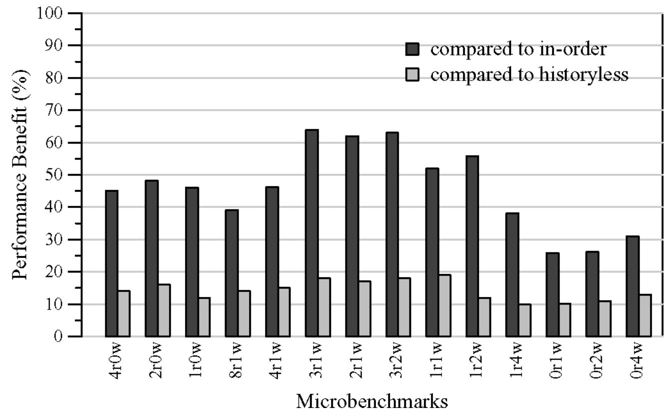

Fig. 4. Performance benefit, for the microbenchmarks, of the AHB scheduler over the in-order and historyless schedulers.

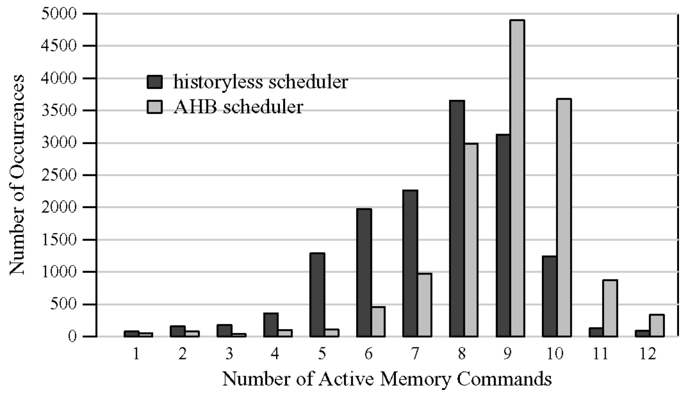

Fig. 5. DRAM Utilization for the daxpy kernel.

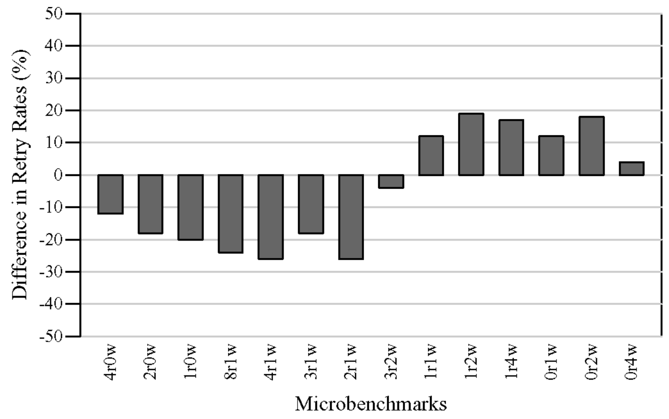

Fig. 6. Percent difference in retry rates between the historyless scheduler and the AHB scheduler ($100 \times \frac{h-a}{h}$, where h is the retry rate of the historyless scheduler and a is the retry rate of the AHB scheduler). The AHB scheduler leads to more retries for the first 8 microbenchmarks.

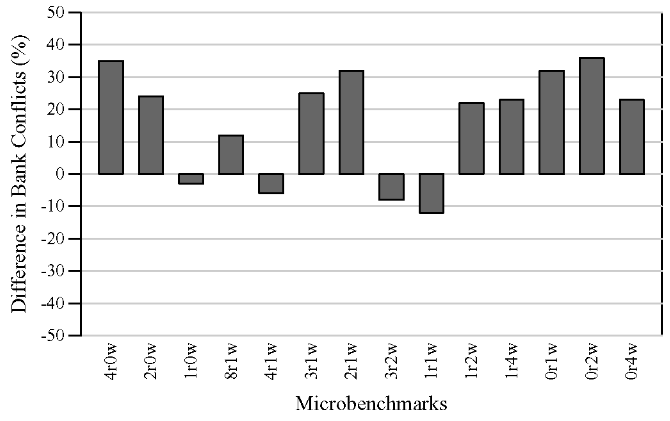

Fig. 7. Percent difference in number of bank conflicts that exist for commands in the reorder queues between the historyless scheduler and the AHB scheduler ($100 \times \frac{h-a}{h}$, where h is the number of bank conflicts of the historyless scheduler and a is the number for the AHB scheduler). The AHB scheduler causes fewer bank conflicts in 10 out of 14 microbenchmarks.

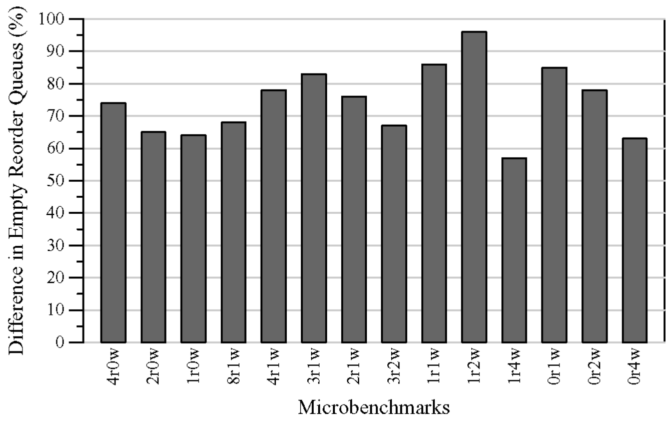

Fig. 8. Reduction in the occurrences of empty reorder queues, which is a measure of the occupancy of the reorder queues. The y-axis represents the percent difference between the historyless scheduler and the AHB scheduler ($100 \times \frac{h-a}{h}$, where h is the number of occurrences of empty reorder queues for the historyless scheduler and a is the number of occurrences for the AHB scheduler). With the AHB scheduler, substantially fewer empty reorder queues occur.

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 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 8 shows that the AHB scheduler 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 9 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.

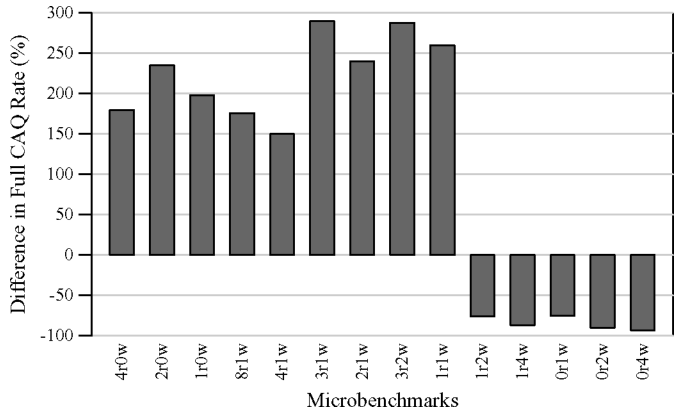

Fig. 9. Increases in the number of occurrences where the CAQ is a bottleneck. The y-axis represents the percent difference between the AHB scheduler and the historyless scheduler ($100 \times \frac{a-h}{h}$, where h is the number of occurrences where the CAQ is a bottleneck for the historyless scheduler and a is the number of occurrences for the AHB scheduler). For the first 9 microbenchmarks, the CAQ is a bottleneck more frequently for the AHB scheduler.

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, the AHB scheduler 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, the scheduler 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.

6.4.2 *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 10 compares the standard deviations of the adaptive history-based and historyless schedulers when data are aligned on 16 different address offsets. We see that the adaptive history-based solution reduces sensitivity to alignment.

6.4.3 *Perfect DRAM Results.* We have so far evaluated the AHB scheduler by comparing against previous solutions. To see how much room there is for further improvement, we compare the performance of the AHB scheduler against a perfect DRAM in which there are no conflict hazards. We find that for

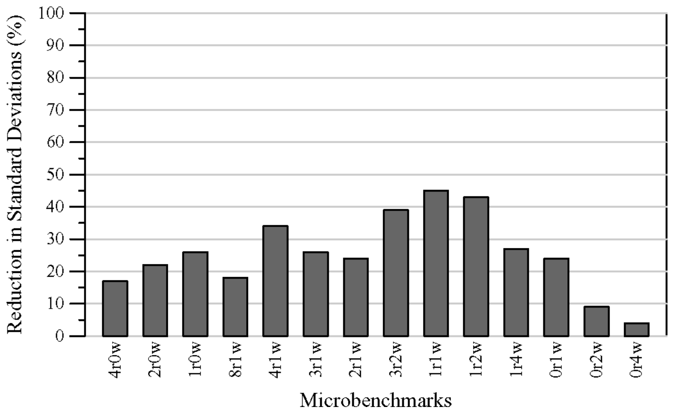

Fig. 10. Reduction in the standard deviations for 16-different address offsets. The y-axis represents percent difference between the standard deviation for the historyless scheduler and the AHB scheduler ($100 \times \frac{h-a}{h}$, where h is the standard deviation for the historyless scheduler and a is the standard deviation for the AHB scheduler).

the benchmarks that we evaluate, the AHB scheduler achieves 95–98% of the performance of the perfect DRAM.

7. SENSITIVITY ANALYSIS

In the previous section we analyzed the performance of the AHB scheduler in the context of the IBM Power5. In this section we explore the broader utility of this scheduler by analyzing its performance in the context of various derivatives of the Power5. In particular, this section has three goals:

—We analyze the sensitivity and robustness of the AHB scheduler to various micro-architectural features. We show that the AHB scheduler yields performance that is robust across a variety of micro-architectural parameters. We also show that the other schedulers cannot achieve the performance of the AHB scheduler even if given additional hardware resources.
—We identify optimal values for parameters related to the memory scheduler.
—We evaluate the AHB scheduler for possible future architectural trends.

The next three subsections address these goals by varying a set of memory controller parameters, DRAM parameters, and system parameters, and studying the effects of these changes on performance. We consider both single and multiple-thread performance, and we use the historyless scheduler as our baseline. To reduce the amount of data to present, we limit our experiments to the daxpy benchmark.

7.1 Modifying Memory Controller Parameters

In this section, we see how the AHB and historyless schedulers are affected by modifying various memory controller design features. Since the design space is large, we limit ourselves to variations of the Power5 memory controller, and we focus on three important parameters: 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 have the greatest impact on performance.

7.1.1 *CAQ Length.* The Centralized 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 historyless 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 6.4, 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 11, 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 a queue length of 4 compared to a length of 3. This performance difference goes up to 4.4% when the queue has 16 slots.

Figure 11 also shows that for the historyless scheduler, longer CAQs always yield better performance, most likely because the historyless scheduler has no way to exploit larger scheduling windows. For example, in the ST case, the performance of the historyless scheduler improves by 7.1% as the CAQ length increases from 3 to 16. However, even with this queue length, the AHB scheduler is still superior to the historyless scheduler. In the SMT experiments with the historyless scheduler, we find that the performance gain from increasing the queue length to 16 is much smaller than for the ST case.

In summary, the performance of the historyless method improves as the CAQ gets longer, but it cannot achieve the performance of the AHB scheduler

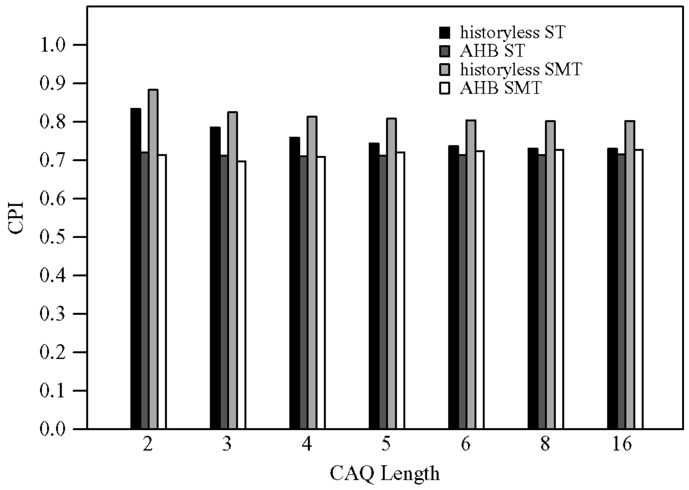

Fig. 11. ST and SMT performance results for the historyless and the AHB schedulers when the length of the CAQ is varied.

even if given a much longer CAQ. We also conclude that the length of the CAQ significantly affects performance.

7.1.2 *Reorder Queue Lengths.* As we show in Figure 1, the Power5 has separate Read and Write reorder queues inside the memory controller. In the current design of the Power5, each of these queues has an equal length of 8. Here, we analyze the effect of the reorder queue lengths on the scheduling approaches.

The length of the reorder queues 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 to 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 12, we present the effects of the reorder queue lengths on performance for both the AHB and the historyless 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 historyless scheduler loses 25.3%. The same reduction in the reorder queue lengths for the SMT experiments degrades performance 27.3% for the AHB scheduler and 19.9% for the historyless schedulers. 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 negligible performance improvements.

To summarize, the AHB method can better utilize a larger reorder queue, so its advantage over the historyless method increases as the reorder queue

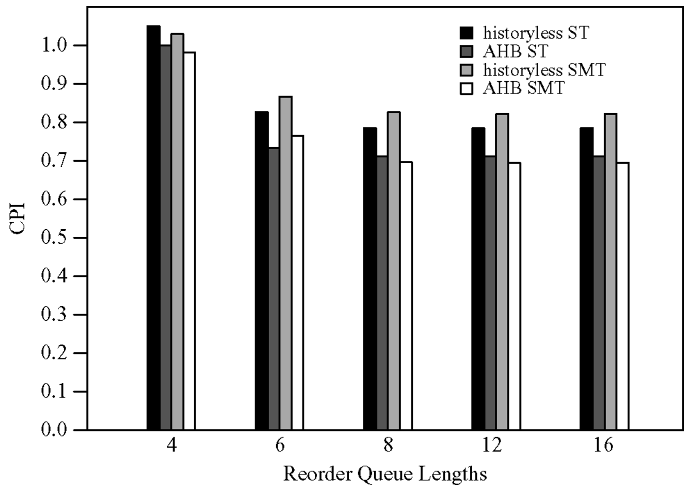

Fig. 12. ST and SMT performance results for the historyless and AHB scheduler when the reorder queue length is varied.

length grows. 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.

7.1.3 *Wait Times for Commands with Bank Conflicts.* This section analyzes the effect of the third memory controller design parameter, which we now describe. We have pointed out that the Power5 memory controller holds in the reorder queues any command that has a bank conflict. Such a strategy is rational because bank conflicts prohibit the entrance of new commands to DRAM, and since the CAQ is a FIFO queue, if the command in front of the CAQ conflicts with a command in DRAM, then all commands in the CAQ are blocked until the conflict is cleared. However, such a strategy is also conservative, because even with an empty CAQ, there is in the current Power5 implementation a latency of 32 processor cycles before a command in a reorder queue is issued to DRAM. Thus, the third parameter, which we refer to as the *wait time,* is the duration of time, measured in cycles, that a command must be held in the reorder queues if it has a bank conflict with a previously issued command.

The 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 destined for 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 this wait time, we conduct experiments for the AHB and the historyless schedulers for both ST and SMT environments. As we see in Figure 13, 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 the ST and SMT experiments. If a command waits until the bank conflict is

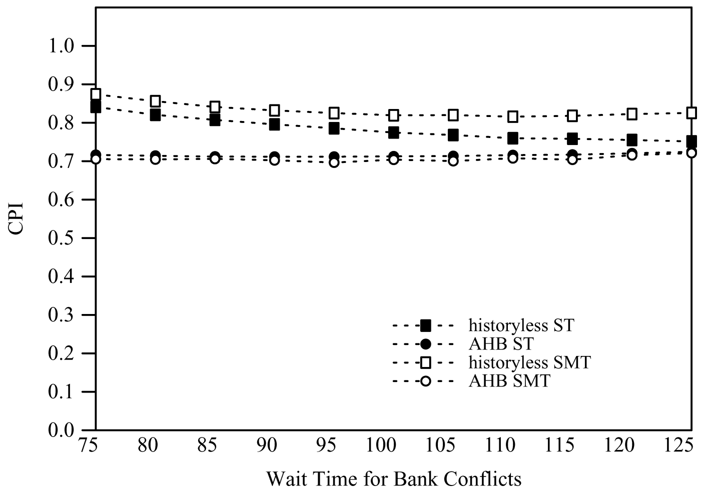

Fig. 13. ST and SMT performance results for the historyless and the AHB schedulers when the wait time for bank conflicts is varied.

cleared, performance will degrade by 1.8% for the ST case and 3.5% for the SMT case. For the historyless approach, 125 and 110 cycles are the optimal wait times for ST and SMT, respectively. The historyless method with SMT has a 1.2% performance advantage when it uses 110 cycle wait time rather than 125 cycles.

In summary, we conclude that to account for internal memory controller latency, the scheduler should issue commands from the reorder queues before the bank conflict is actually cleared. We also find that for the ST case, the AHB approach is less sensitive to this parameter than the historyless approach. For the SMT case, both scheduling approaches show similar sensitivity.

7.2 Modifying DRAM Parameters

In this section we investigate the effect of varying DRAM system parameters. In particular, we evaluate the performance of the AHB and the historyless 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.

7.2.1 *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 14 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

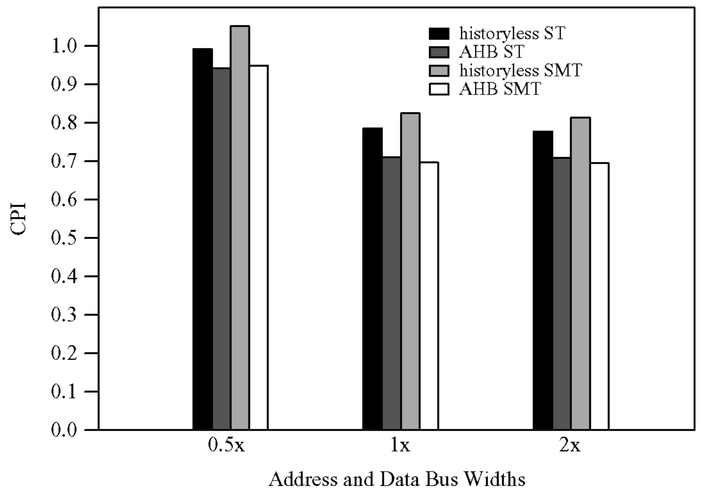

Fig. 14. ST and SMT performance results for the historyless and AHB schedulers when the memory address and data bus widths are varied.

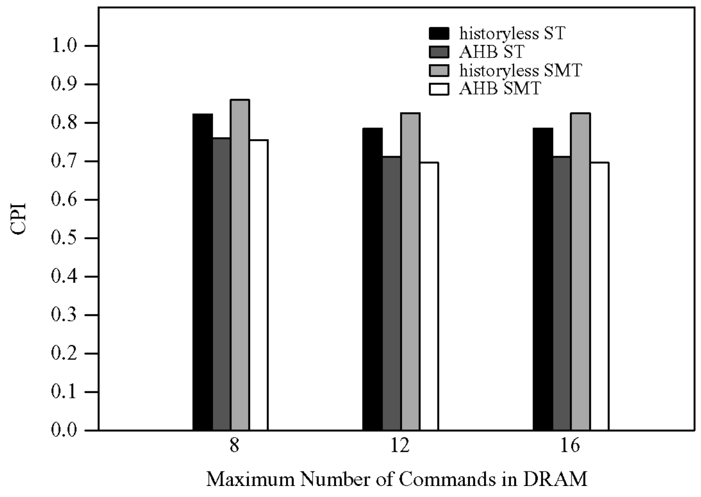

Fig. 15. ST and SMT performance results for the historyless and AHB schedulers when the maximum number of DRAM commands is varied.

by 50% significantly degrades performance (20.9–26.6%) for both the AHB and historyless schedulers. We also observe that increasing bus widths beyond the current values of the Power5 has little effect on performance.

7.2.2 *Maximum Number of Commands in DRAM.* In the systems that 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 15, we show results for both ST and SMT workloads. We find that increasing beyond

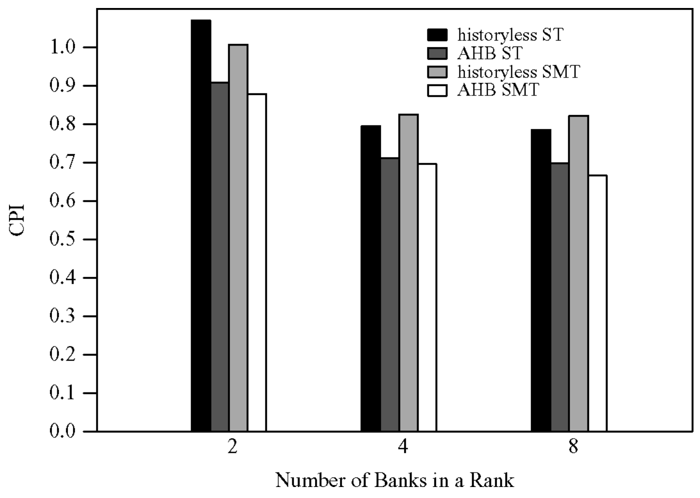

Fig. 16. ST and SMT performance results for the historyless and the AHB schedulers when the number of banks in a rank is varied.

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%.

7.2.3 *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 16 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 ST and SMT experiments. The performance gain is 20.8%–21.7% and 18.1%–26.6% for the AHB and historyless schedulers, respectively. On the other hand, further increasing the number of banks to eight has a much smaller effect: It does not improve the performance of the historyless scheduler, and the performance gain for the AHB scheduler is 1.9% for the ST experiments and 4.6% for the SMT experiments. In summary, our experiments confirm our intuition that the benefit of AHB scheduling increases as the number of banks in a rank increases, that is, as the memory system admits more parallelism.

7.3 Modifying System Parameters

7.3.1 *Processor Frequency.* In addition to memory controller and DRAM parameters, we also explore the impact of higher processor clock rates. While increases in clock rate have slowed, processor frequency continues to increase. In Figure 17, we present the differences between the AHB and the historyless schedulers for systems with 1.5, 2, 3, and 4 times the processor frequency of our base Power5 system. As the ratio of the processor frequency to the DRAM frequency grows, we find that advantage of the AHB scheduler over the historyless method also increases. For example, for the ST case, with the current processor frequency, the AHB scheduler is superior to the historyless scheduler by 9.3%, but the advantage grows to 15.6% when the processor frequency doubles. Similarly, for the SMT case, AHB method's advantage increases from

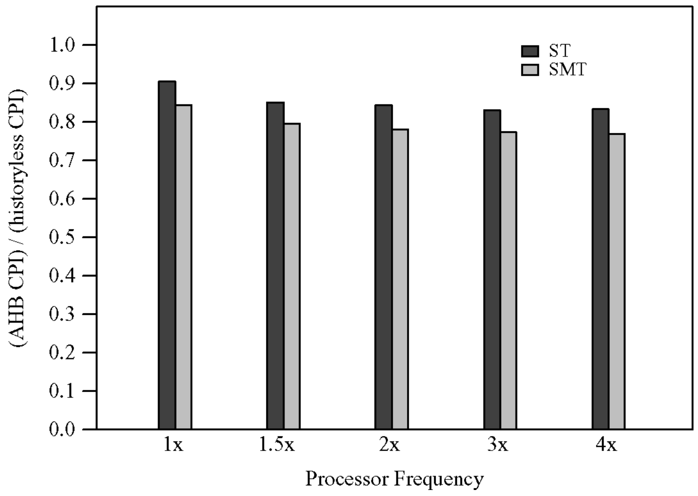

Fig. 17. ST and SMT performance results for the historyless and AHB schedulers when the processor frequency is increased by 1.5×, 2×, 3×, and 4×.

16.4% to 22.0% with 2× processor frequency. We see that as the ratio of the processor to memory speeds increases, the significance of AHB scheduling also increases because the value of memory bandwidth grows.

7.3.2 *Data Prefetching.* Finally, we 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 historyless method 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 AHB scheduler over the historyless scheduler is reduced from 16.4% to 7.3% when the hardware prefetching unit is turned off.

8. HARDWARE COSTS

To evaluate the cost of the AHB scheduler, 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 the adaptive history-based scheduler. To quantify these costs, we use transistor counts provided by the designers of the Power5 to provide detailed estimates of the transistor budget of the modified memory controller. We find that the memory controller utilizes 1.58% of the Power5's total transistors. The size of one historyless scheduler is in turn 1.19% of the memory controller. The adaptive history-based scheduler 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 the AHB scheduler has only negligible effects on power.

9. CONCLUSIONS

This article 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 scheduler that incorporates several techniques. First, the scheduler uses the command history—in conjunction with a cost model—to select commands that will have low latency. The scheduler also uses 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, the AHB scheduler implements three history-based schedulers—each tailored to a different command pattern—and dynamically selects from among these three schedulers based on the observed ratio of Reads and Writes.

To place our work in 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 historyless scheduler. We use this historyless 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 the AHB scheduler using three benchmark suites. For SMT workloads consisting of the Stream benchmarks, this scheduler improves CPI by 55.9% over in-order scheduling and 15.6% over historyless scheduling. For the NAS benchmarks, again with SMT workloads, the improvements are 26.3% over in-order scheduling and 9.9% over historyless scheduling. For a set of commercial SMT workloads, the improvements are 51.8% over in-order scheduling and 7.6% over historyless scheduling.

To explain our results, we have looked inside the memory system to provide insights about how the AHB scheduler 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.

To better understand the broader applicability of our ideas, we have explored the effects of varying characteristics 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 multithreaded workloads. We find that the AHB scheduler is more robust than historyless scheduling in the sense that its performance is less sensitive to changes in design parameters. We also find that the AHB scheduler is typically superior to the historyless scheduler even when the latter is given additional hardware resources.

ACKNOWLEDGMENTS

We thank the entire IBM Power5 team, in particular, Men-Chow Chiang, Cheryl Chunco, Steve Dodson, Douglas Logan, John McCalpin, Gary Morrison, Stephen J. Powell, Ram Raghavan, and Joel Tendler. We also thank Bill Mark, E Lewis, and the anonymous referees for their valuable comments on previous drafts of this article.

REFERENCES

Bailey, D., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrishnan, V., and Weeratunga, S. 1994. The NAS parallel benchmarks (94). Tech. rep. RNR-94-007, NASA Ames Research Center.

Carter, J., Hsieh, W., Stoller, L., Swanson, M., Zhang, L., Brunvand, E., Davis, A., Kuo, C.-C., Kuramkote, R., Parker, M., Schaelicke, L., and Tateyama, T. 1999. Impulse: Building a smarter memory controller. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*. 70–79.

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

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

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

Foster, J. E. 2000. Memory controller and method for dynamic page management. U.S. Patent 6,052,134.

Gao, Q. S. 1993. The Chinese remainder theorem and the prime memory system. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 337–340.

Harper, III, D. T. and Jump, J. R. 1986. Performance evaluation of vector accesses in parallel memories using a skewed storage scheme. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*. 324–328.

Harriman, D. J., Langendorf, B. K., and Ajanovic, J. 2000. Method and apparatus for improving system performance when reordering commands. U.S. Patent 6,088,772.

Harris, J. G. 2003. Apparatus and method for handling memory access requests in a data processing system. U.S. Patent 6,601,151.

Hur, I. 2006. Enhancing memory controllers to improve DRAM power and performance. Ph.D. thesis, The University of Texas at Austin.

Hur, I. 2007. Method and system for creating and dynamically selecting an arbiter design in a data processing system. US patent 7,287,111.

Hur, I. and Lin, C. 2004. Adaptive history-based memory schedulers. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*. 343–354.

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

Jenne, J. E. and Olarig, S. P. 2003. Method and apparatus for scheduling memory calibrations based on transactions. U.S. Patent 6,631,440.

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

Kessler, R. E., Bertone, M. S., Braganza, M. C., Bouchard, G. A., and Steinman, M. B. 2003. System for minimizing memory bank conflicts in a computer system. U.S. Patent 6,622,225.

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

Larson, D. A. 2001. Apparatus for controlling pipelined memory access requests. U.S. Patent 6,321,233.

MATHEW, B. 2000. Parallel vector access: A technique for improving memory system performance. M.S. thesis, University of Utah.

MATHEW, B., MCKEE, S. A., CARTER, J. B., AND DAVIS, A. 2000a. Algorithmic foundations for a parallel vector access memory system. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 156–165.

MATHEW, B., MCKEE, S. A., CARTER, J. B., AND DAVIS, A. 2000b. Design of a parallel vector access unit for SDRAM memory systems. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA-6)*. 39–48.

MCCALPIN, J. D. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*.

MCGEE, B. J. AND CHAU, J. B. 2005. Memory controller and method using read and write queues and an ordering queue for dispatching read and write memory requests out of order to reduce memory latency. U.S. Patent 6,877,077.

MCKEE, S. A. 1993. Hardware support for dynamic access ordering: Performance of some design options. Tech. rep. CS-93-08, University of Virginia.

MCKEE, S. A. 1995. Maximizing memory bandwidth for streamed computations. Ph.D. thesis, University of Virginia.

MCKEE, S. A., KLENKE, R. H., WRIGHT, K. L., WULF, W. A., SALINAS, M. H., AYLOR, J. H., AND BATSON, A. P. 1998. Smarter memory: Improving bandwidth for streamed references. *IEEE Comput.*, *31*, 54–63.

MCKEE, S. A., WULF, W. A., AYLOR, J. H., SALINAS, M. H., KLENKE, R. H., HONG, S. I., AND WEIKLE, D. A. B. 2000. Dynamic access ordering for streamed computations. *IEEE Trans. Comput. 49,* 11, 1255–1271.

MICRON. 2004. http://download.micron.com/pdf/datasheets/dram/ddr2/512MbDDR2.pdf.

MOYER, S. A. 1993. Access ordering and effective memory bandwidth. Ph.D. thesis, University of Virginia.

PEIRON, M., VALERO, M., AYGUADE, E., AND LANG, T. 1995. Vector multiprocessors with arbitrated memory access. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. 243–252.

RAGHAVAN, R. AND HAYES, J. P. 1990. On randomly interleaved memories. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. 49–58.

RAU, B. R. 1991. Pseudo-randomly interleaved memory. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*. 74–83.

RIXNER, S. 2004. Memory controller optimizations for web servers. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. 355–366.

RIXNER, S., DALLY, W. J., KAPASI, U. J., MATTSON, P., AND OWENS, J. D. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*. 128–138.

SAH, S., KULICK, S. S., UDOMPANYANAN, V., NATARAJAN, C., AND PAI, H. S. 2006. Memory read/write reordering. U.S. Patent 7,047,374.

SCOTT, S. L. 1996. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. 26–36.

SHERWOOD, T., PERELMAN, E., AND CALDER, B. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques*. 3–14.

TENDLER, J. M., DODSON, J. S., FIELDS JR., J. S., LEE, H., AND SINHAROY, B. 2002. Power4 system microarchitecture. *IBM J. Resear. Develop. 46,* 1, 5–26.

VALERO, M., LANG, T., LLABER, J. M., PEIRON, M., AYGUADE, E., AND NAVARRA, J. J. 1992. Increasing the number of strides for conflict-free vector access. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*. 372–381.

Received May 2006; revised August 2007; accepted October 2007