Adaptive History-Based Memory Schedulers for Modern Processors

Careful memory scheduling can increase memory bandwidth and overall system performance. We present a new memory scheduler that makes decisions based on the history of recently scheduled operations, providing two advantages: it can better reason about the delays associated with complex DRAM structure, and it can adapt to different observed workload.

Ibrahim Hur
IBM Corp. and University of Texas at Austin

Calvin Lin
University of Texas at Austin

The gap between processor and memory speeds has led to aggressive use of techniques such as prefetching, speculation, and multithreading, which consume bandwidth to hide or reduce memory latency. Architectural trends toward multicore chips and application trends toward multimedia workloads further increase the demands on memory systems, which make memory bandwidth an increasingly important resource.

To increase bandwidth, modern DRAMs have hierarchical structures that allow multiple memory commands to concurrently access different substructures. However, if two commands are scheduled for the same substructure, the second command might need to stall for a certain number of cycles. Rixner et al.[1] have shown that simple schemes for reordering memory commands can avoid such hardware conflicts and significantly improve memory bandwidth. However, modern DRAMs with 5D structure and many different types of internal latencies make more sophisticated memory scheduling schemes necessary.

To complicate matters, as DRAMs have become more complex, so have their accompanying memory controllers, which use internal buffering and parallelism to increase bandwidth. For example, Figure 1 shows the structure of the IBM Power5's memory controller. Memory commands from the L2 and L3 cache are placed in the two reorder queues. The memory arbiter moves commands from these queues to the central arbiter queue (CAQ), from which they go to memory in first-in, first-out order. If the memory arbiter is not careful, bottlenecks in the various queues can limit bandwidth. For example, if the application requests commands in a ratio of two reads for every write, and if the arbiter selects commands in the ratio of one read for one write, the read queue will quickly become full, preventing the memory controller from accepting any more reads. Thus, a second goal of a memory scheduler is to avoid bottlenecks within the memory controller itself.

This article describes a new memory scheduling approach that makes decisions based on

Published by the IEEE Computer Society

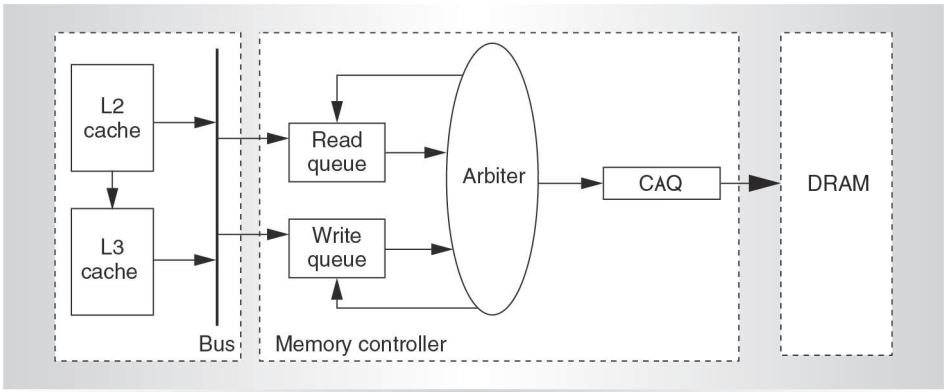

Figure 1. Power5 memory controller.

the history of recently scheduled operations. When compared with an in-order scheduler, our solution improves instruction per cycle (IPC) on the NASA Advanced Supercomputing (NAS) benchmarks by a geometric mean of 10.9 percent, and it improves IPC on the Stream benchmarks by more than 63 percent. Compared to one of Rixner et al.'s solutions[1], our approach sees improvements of 5.1 percent for the NAS benchmarks and more than 18 percent for the Stream benchmarks. Our solution also has minimal hardware cost. We observe that a history length of two is effective for the Power5 system, which increases the chip's transistor count by approximately 0.04 percent.

Related work

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

Rixner et al.[1] explore several simple policies for reordering accesses on the Imagine stream processor.[2] These policies reorder memory operations by considering the characteristics of modern DRAM systems. For example, one policy gives row accesses priority over column accesses, and another gives column accesses priority over row accesses. None of these simple policies prove best in all situations. Furthermore, these policies do not easily extend to more complex memory systems with many different types of hardware constraints.

Moyer avoids bank conflicts by using compiler transformations to change the order of the memory requests that the processor generates.[3] Moyer's technique applies specifically to stream-oriented workloads in cacheless systems. McKee et al. use a runtime approach to order the accesses to streams in a stream memory controller.[4-5] This approach uses a simple ordering mechanism: The memory controller considers each stream buffer in round-robin fashion, streaming as much data as possible to the current buffer before going to the next buffer. This approach can reduce conflicts among streams, but it does not reorder references within a single stream.

Valero et al. reduce bank conflicts on vector processors by reordering memory commands so that memory banks are accessed strictly in round robin order.[6] The Impulse memory system by Carter et al. improves memory system performance by dynamically remapping physical addresses.[7] This approach requires modifications to the applications and operating system.

A modern architecture: The IBM Power5

The Power5[8,9] is IBM's successor to the Power4.[10] The Power5 has two processors per chip, where each processor has split L1 data and instruction caches. Each chip has a unified L2 cache that the two processors share, along with an optional L3 cache. The Power5 has hardware data prefetching units that prefetch from memory to L2, and from L2 to L1.

Each Power5 chip contains a single memory controller—shown schematically in Figure 1—that the two processors share. Each reorder queue can hold eight memory commands, where each memory reference is an entire L2 cache line or a portion of an L3 cache line. The CAQ can hold four commands.

The Power5 does not allow dependent mem-

ory operations to enter the memory controller at the same time, so the arbiter can arbitrarily reorder memory operations. 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.

Power5 memory system

The Power5 systems that we consider use DDR2-266 (double data rate 2, 266 MHz) SDRAM chips, which have a 5D structure. Two ports connect the memory controller to the DRAM, which has four ranks, where each rank is an organizational unit consisting of four banks. Each bank, in turn, is organized as a set of rows and columns. This structure imposes many different constraints. For example, port, rank, and bank conflicts each incur their own delay, and the costs of these delays depend on whether the operations are reads or writes. In this system, bank conflict delays are an order of magnitude longer than the delays introduced by rank or port conflicts.

Our solution

A history-based scheduler selects the next memory command using the history of recently scheduled memory commands and the set of available commands from the reorder queues. Thus, the scheduler's goal is to prioritize the set of available commands using some optimization criterion. In our work, we use two optimization criteria. The first criterion is to minimize the latency of the scheduled command. The second is to match some desired balance of reads and writes to avoid bottlenecks in the reorder queues. Because both goals are important, we probabilistically combine two arbiters FSMs to produce a scheduler that encodes both goals. The result is a history-based scheduler optimized for one particular command pattern. To accommodate a wide range of command patterns, we introduce adaptivity by using three history-based schedulers, one tuned for a heavy mix of reads, one tuned for a heavy mix of writes, and one tuned for a balanced mix of reads and writes. Our adaptive scheduler then observes the recent command pattern and periodically chooses the most appropriate history-based scheduler.

History-based arbiters

We can implement a history-based arbiter as a finite-state machine (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—read-read, read-write, write-read, and write-write—leading to four possible FSM states. Here, history string *x*-*y* means that the last command transmitted to memory was *y* and the one before that was *x*.

Each state of the FSM encodes the history of recent commands, and the FSM checks for possible next commands in some particular order, effectively prioritizing the desired next command. When the arbiter selects a new command, it changes state to represent the new history string. If the reorder queues are empty, there is no state change in the FSM.

An FSM for an arbiter that uses a history length of three illustrates this process. Assume that each command is either a read or a write operation to either port number 0 or 1. Therefore, there are four possible commands: 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 command types. In this example, since the arbiter keeps the history of the last three commands and there are four possible command types, the total number of states in the FSM is $4 \times 4 \times 4 = 64$. Figure 2 shows transitions from one particular state in this sample FSM. In this hypothetical example, 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.

Design details of history-based arbiters

We have identified two optimization criteria for prioritization: the amount of deviation from the command pattern and the expected latency of the scheduled command. The first criterion allows an arbiter to schedule commands to match some expected mixture of reads and writes. The second criterion represents the mandatory delay between the new memory command and the commands that

are currently being processed by the DRAM. We have developed algorithms to generate FSMs for each optimization criteria.[11,12]

The first algorithm generates state transitions for an arbiter that schedules commands to match a certain ratio of reads and writes. In situations where multiple available commands have the same effect on the deviation from the read-write ratio of the arbiter, the algorithm uses some secondary criterion, such as expected latency, to make final decisions.

The second algorithm minimizes the expected latency of its scheduled operations, using a cost model for the mandatory delays between various memory operations. The model captures 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 define cost functions to represent the mandatory delays between any two memory commands that cause a hardware hazard. For our system, we have many cost functions, each representing delays such as the following:

- a write to a different rank after a read,
- a read to the same port after a write,
- a read to the same port but to a different rank after a read.

As with the first algorithm, if multiple available commands have the same expected latency, we use the secondary criterion—in this case, the deviation from the command pattern—to break ties.

To combine the two optimization criterion, our approach weights each criterion and probabilistically chooses between the two. This technique essentially interleaves two state machines into one, periodically switching between the two. We currently give equal weight to each criterion, but future work will study whether some other weighting produces better results.

Adaptive selection of arbiters

Figure 3 shows a schematic of our adaptive history-based arbiter. The memory controller tracks the command pattern that it receives from the processors and periodically switches among the arbiters depending on this pattern.

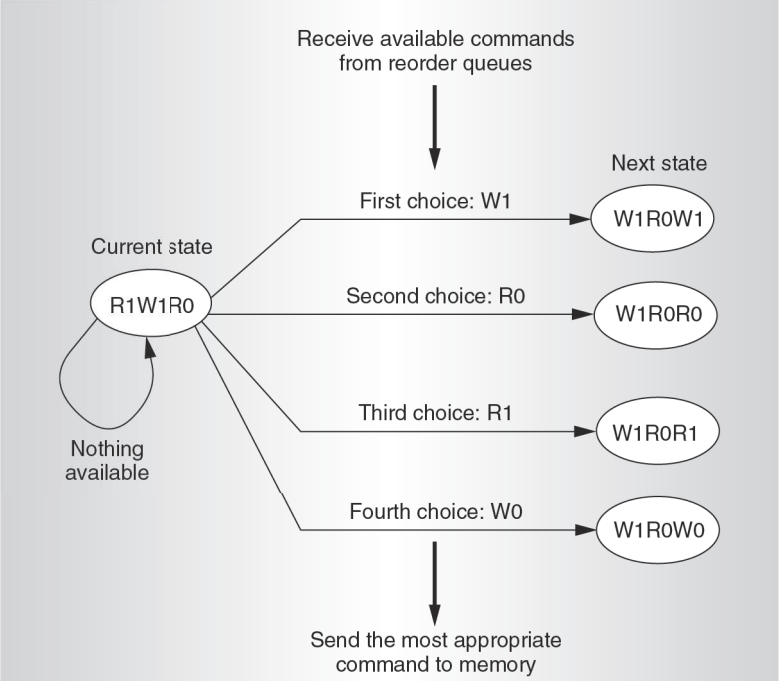

Figure 2. Transition diagram for the current state, R1W1R0. Each available command type has a different selection priority.

Evaluation

We evaluate the performance of our memory scheduler using the Stream (http://www.cs.virginia.edu/stream/) and NAS benchmarks.[13] The Stream benchmarks, which have been used to measure the sustainable memory bandwidth of systems,[14] consist of four simple vector kernels: Copy, Scale, Sum, and Triad. The NAS benchmarks are well-known, fairly data-intensive, scientific benchmarks.

Schedulers studied

We compare three memory arbiters. The first, in-order, implements the simple FIFO policy that most general-purpose memory controllers use. In a Power5 system, this arbiter would transmit memory commands from the reorder queues to the CAQ in the order in which they arrive from the processors.

The second arbiter implements one of the policies that Rixner et al.[1] propose. We refer to this as the memoryless arbiter because it does not use command history information. This arbiter avoids long bank conflict delays by selecting commands from the reorder queues that do not conflict with commands in DRAM. When there are multiple eligible

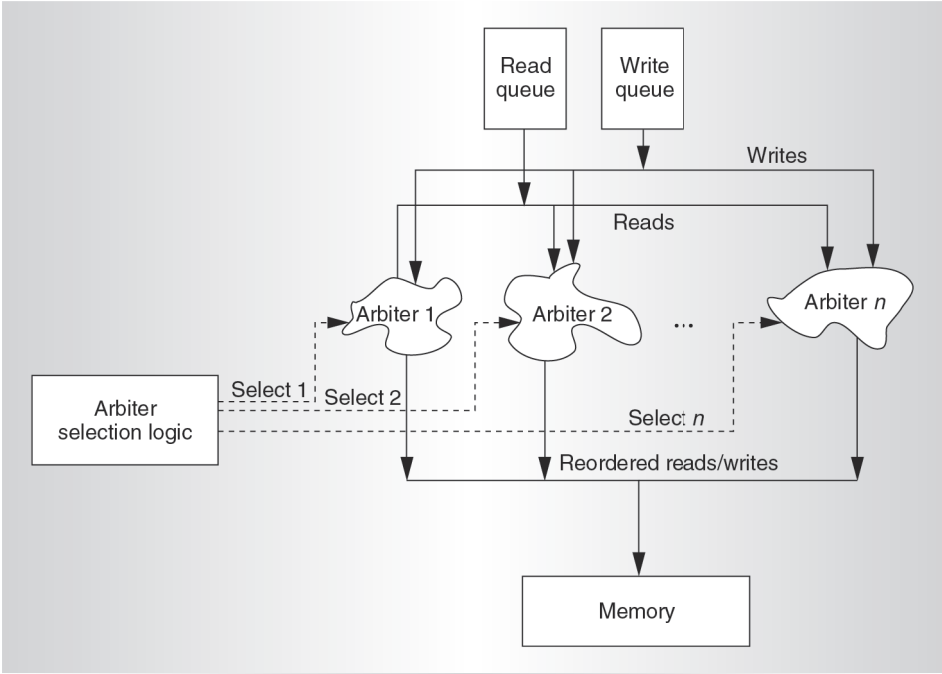

Figure 3. Overview of dynamic selection of arbiters in memory controller.

commands, it chooses the oldest command.

The third arbiter is our adaptive history-based scheduler. Because bank conflicts cause much longer delays than rank or port conflicts, our scheduler avoids bank conflicts using a simple approach similar to the memoryless arbiter. To select the most appropriate command from among the remaining commands in the reorder queues, our scheduler then implements our adaptive history-based technique. Thus, our adaptive history-based approach handles rank and port conflicts, but not bank conflicts. Our history-based approach could also be used to prioritize bank conflicts, at the cost of increased hardware complexity.

Simulation methodology

To evaluate performance, we use a cycle-accurate simulator for the IBM Power5, one verifiable to within 1 percent of the actual hardware's performance. This simulator, one of several simulators used by the Power5 design team, uses trace-based simulation to simulate both the processor and the memory system.

Simulation parameters

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

The adaptive history-based arbiter combines these three history-based arbiters by using the 2R1W arbiter when the read-write ratio is greater than 1.2, the 1R1W arbiter when the read-write ratio is between 0.8 and 1.2, and the 1R2W arbiter, otherwise. The adaptive scheduler performs this arbiter selection every 10,000 processor cycles. (Our results are not very sensitive to this period, as long as it is greater than about 100 cycles.)

Results

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

The NAS benchmarks provide a more comprehensive evaluation of overall performance; they include

- embarrassingly parallel (EP),
- multigrid (MG),
- conjugate gradient (CG),
- 3D fast Fourier transfer (FT) partial differential equation,
- integer sort (IS),
- lower-upper (LU) matrix solver,
- pentadiagonal solver (SP), and
- block tridiagonal solver (BT).

The two graphs in Figure 5 show that improvements over the in-order method are

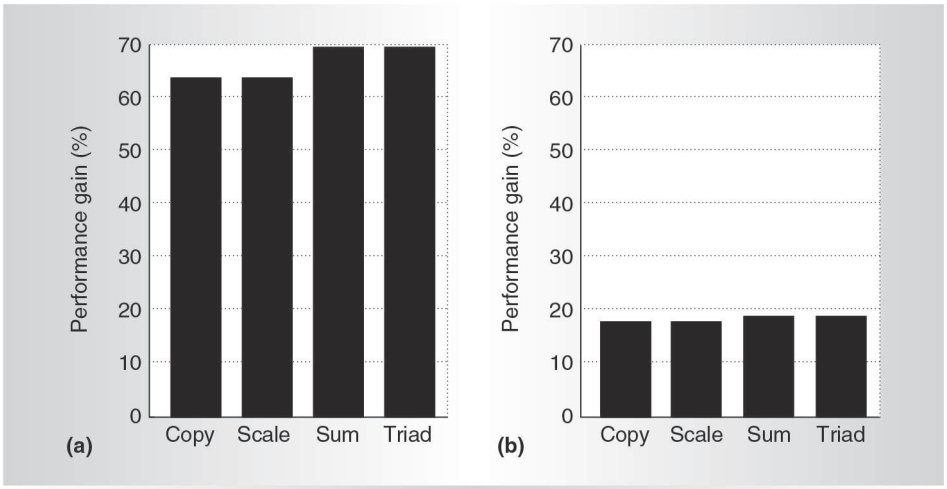

Figure 4. Stream benchmark performance for the adaptive history-based approach against the in-order (a) and memoryless (b) arbiters.

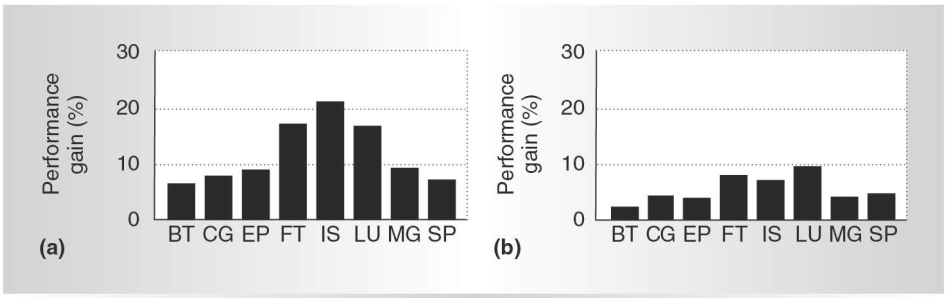

Figure 5. NAS benchmark performance of the adaptive history-based approach against the in-order (a) and memoryless (b) arbiters.

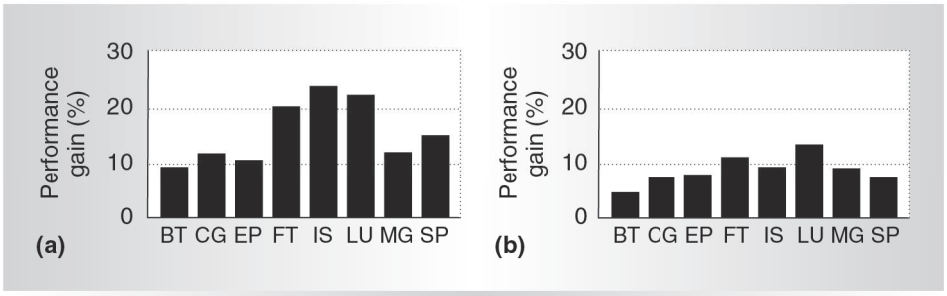

Figure 6. NAS benchmark performance of the adaptive history-based approach against the in-order (a) and memoryless (b) arbiters running on a processor that is four times faster than the current IBM Power5.

between 6.6 and 21.4 percent, with a geometric mean of 10.9 percent. Improvements over the memoryless method are between 2.4 and 9.7 percent, with a geometric mean of 5.1 percent. Figure 6 presents results when the CPU has four times the clock rate, showing that future systems will benefit more from these intelligent arbiters. Here, the geometric mean improvement is 14.9 percent over the in-order arbiter and 8.4 percent over the memoryless arbiter.

Understanding the results

To better understand our results, we conduct

experiments to examine various potential bottlenecks within the memory controller.[12] Based on those experiments, our solution improves bandwidth by moving bottlenecks from outside the memory controller, where the arbiters cannot help, to inside the memory controller. More specifically, the bottlenecks tend to appear at the end of the pipeline—at the CAQ—where there is no more ability to reorder memory commands. By shifting the bottleneck, our solution tends to increase the occupancy of the reorder queues, which gives the arbiter more memory operations to choose from. The result is fewer hardware conflicts and increased bandwidth.

To see how much room there is for further improvement, we compare the performance of our new arbiter against a perfect DRAM that has no hardware hazards. We find that for our benchmarks, our solution achieves 95 to 98 percent of the performance of the perfect DRAM.

Other benefits of our solution[12] include a reduced sensitivity to data alignment. With a poor scheduler, data alignment can cause significant performance differences. The largest effect occurs when a data structure fits into one cache line when aligned fortuitously but straddles two cache lines when aligned differently. In such cases, the poor alignment doubles the number of memory commands. By improving bandwidth, our adaptive history-based solution reduces this sensitivity to alignment.

To quantify the cost of our solution in terms of transistors, we use the Power5 implementation to provide detailed transistor count estimates. We find that the memory controller consumes 1.58 percent of the Power5's total transistors. The size of one memoryless arbiter is in turn 1.19 percent of the memory controller. Our adaptive history-based arbiter increases the size of the memory controller by 2.38 percent, which increases the overall chip's transistor count by 0.038 percent.

To satisfy the increasing demand on memory bandwidth for general-purpose processors, our new arbiter incorporates several techniques. We use the command history—in conjunction with a cost model—to select commands that will have low latency. We also use the command history to schedule commands that match some expected command pattern, because this tends to avoid bottlenecks in our memory controller's reorder queues. Both of these techniques can be implemented using FSMs, but because the goals of the two techniques might conflict, we probabilistically combine these FSMs to produce a single history-based arbiter that partially satisfies both goals. Finally, because we cannot know the actual command-pattern beforehand, we implement three history-based arbiters—each tailored to a different command pattern—and our adaptive scheduler dynamically selects from among these three arbiters based on the observed ratio of reads and writes.

As future work, we plan to evaluate the effectiveness of our techniques in a broader variety of contexts. First, we plan to understand the sensitivity of our results to various microarchitectural features. In the memory controller itself, we will study the effect of varying the reorder queue and CAQ sizes. In the DRAM system, we will vary the number of ranks, the number of banks, and the memory address and the data bus widths. In the larger system, we will explore the impact of higher clock rates in both the processor and the memory system. Second, we plan to explore the robustness of our approach in the face of multiple threads running on the same processor. Our intuition is that multiple threads will require a finer period of adaptivity. Finally, we plan to explore the applicability of our work to power management. In particular, we believe that our scheduler can simultaneously manage both performance and power in memory systems. MICRO

Acknowledgments

We thank Steve Dodson and the entire Power5 team. We also thank Bill Mark, E. Lewis, and the anonymous referees for their valuable comments on earlier drafts of this article. This work was supported by NSF grants ACI-9984660 and ACI-0313263, an IBM Faculty Partnership Award, and DARPA contract F33615-03-C-4106.

References

1. S. Rixner et al., "Memory Access Scheduling," *Proc. 27th Ann. Int'l Symp. Computer Architecture* (ISCA 00), ACM Press, 2000, pp. 128-138.
2. B. Khailany et al., "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no.

2, Mar.-Apr. 2001, pp. 35-46.
3. S.A. Moyer, *Access Ordering and Effective Memory Bandwidth*, doctoral thesis, Dept. of Computer Science, Univ. of Virginia, 1993.
4. S.A. McKee, et al., *Dynamic Access Ordering for Streamed Computations*, *IEEE Trans. Computers*, vol. 49, no. 11, Nov. 2000, pp. 1255-1271.
5. C. Zhang and S.A. McKee, "Hardware-Only Stream Prefetching and Dynamic Access Ordering," *Proc. 14th Int'l Conf. Supercomputing* (ICS 00), ACM Press, 2000, pp. 167-175.
6. M. Valero et al., "Increasing the Number of Strides for Conflict-Free Vector Access," *Proc. 19th Ann. Int'l Symp. Computer Architecture* (ISCA 92), ACM Press, 1992, pp. 372-381.
7. J. Carter et al., "Impulse: Building a Smarter Memory Controller," *Proc. 5th Int'l Symp. High-Performance Computer Architecture* (HPCA 99), IEEE CS Press, 1999, pp. 70-79.
8. J. Clabes et al., "Design and Implementation of the Power5 Microprocessor," *Proc. 41st Ann. Design Automation Conf.* (DAC 04), ACM Press, 2004, pp. 670-672.
9. R. Kalla et al. "IBM Power5 chip: A Dual-Core Multithreaded Processor," *IEEE Micro*, vol. 24, no. 2, Mar.-Apr. 2004, pp. 40-47.
10. J.M. Tendler et al., "Power4 System Microarchitecture," *IBM J. Research and Development*, vol. 46, no. 1, Jan. 2002, pp. 5-26.
11. I. Hur, *Method and System for Creating and Dynamically Selecting an Arbiter Design in a Data Processing System*, patent pending, 2004.
12. I. Hur and C. Lin, "Adaptive History-Based Memory Schedulers," *37th Ann. IEEE/ACM Int'l Symp. Microarchitecture* (Micro-37), IEEE CS Press, 2004, pp. 343-354.
13. D. Bailey et al., *The NAS Parallel Benchmarks (94)*, tech. report, RNR-94-007, Nat'l Aeronautics and Space Administration, Mar. 1994; http://www.nas.nasa.gov/News/Techreports/1994/PDF/RNR-94-007.pdf.
14. Z. Cvetanovic, "Performance Analysis of the Alpha 21364-Based HP GS1280 Multiprocessor," *Proc. 30th Ann. Int'l Symp. Computer Architecture* (ISCA 03), IEEE CS Press, pp. 218-229.

Ibrahim Hur is a staff engineer in the Future Processor Performance Department of IBM Corp. and a PhD candidate at the University of Texas at Austin. His research interests include novel microprocessor architectures, memory systems, performance analysis, and performance modeling. Hur has a MSc in computer science from Southern Methodist University. He is a member of the ACM and the IEEE.

Calvin Lin is an associate professor in the Department of Computer Sciences at the University of Texas at Austin. His research interests include languages, compilers, and microarchitecture. Lin has a PhD in computer science from the University of Washington. He is a member of the ACM and the IEEE.

Direct questions and comments about this article to Ibrahim Hur, ihur@cs.utexas.edu.