Designing a Modern Memory Hierarchy with Hardware Prefetching

Wei-fen Lin, *Student Member*, *IEEE*, Steven K. Reinhardt, *Member*, *IEEE*, and Doug Burger, *Member*, *IEEE*

Abstract—In this paper, we address the severe performance gap caused by high processor clock rates and slow DRAM accesses. We show that, even with an aggressive, next-generation memory system using four Direct Rambus channels and an integrated one-megabyte level-two cache, a processor still spends over half its time stalling for L2 misses. Our experimental analysis begins with an effort to tune our baseline memory system aggressively: incorporating optimizations to reduce DRAM row buffer misses, reordering miss accesses to reduce queuing delay, and adjusting the L2 block size to match each channel organization. We show that there is a large gap between the block sizes at which performance is best and at which miss rate is minimized. Using those results, we evaluate a hardware prefetch unit integrated with the L2 cache and memory controllers. By issuing prefetches only when the Rambus channels are idle, prioritizing them to maximize DRAM row buffer hits, and giving them low replacement priority, we achieve a 65 percent speedup across 10 of the 26 SPEC2000 benchmarks, without degrading the performance of the others. With eight Rambus channels, these 10 benchmarks improve to within 10 percent of the performance of a perfect L2 cache.

Index Terms—Prefetching, caches, memory bandwidth, spatial locality, memory system design, Rambus DRAM.

1 Introduction

Continued improvements in processor performance and, in particular, sharp increases in clock frequencies are placing increasing pressure on the memory hierarchy. Modern system designers employ a wide range of techniques to reduce or tolerate memory-system delays, including dynamic scheduling, speculation, and multithreading in the processing core; multiple levels of caches, nonblocking accesses, and prefetching in the cache hierarchy; and banking, interleaving, access scheduling, and high-speed interconnects in main memory.

In spite of these optimizations, the time spent in the memory system remains substantial. In Fig. 1, we depict the performance of the SPEC CPU2000 benchmarks for a simulated 1.6GHz, 4-way issue, out-of-order core with 64KB split level-one caches; a four-way, 1MB on-chip level-two cache; and a Direct Rambus memory system with four 1.6GB/s channels. (We describe our target system in more detail in Section 3.) Let I_{Real}, $I_{PerfectL2}$, and $I_{PerfectMem}$ be the instructions per cycle of each benchmark assuming the described memory system, the described L1 caches with a perfect L2 cache, and a perfect memory system (perfect L1 cache), respectively. The three sections of each bar, from bottom to top, represent I_{Real}, $I_{PerfectL2}$, and $I_{PerfectMem}$. By taking the harmonic mean of these values across our benchmarks and computing $I_{PerfectMem} - I_{Real}/I_{PerfectMem}$, we obtain the fraction of performance lost due to an imperfect memory system.[1] Similarly, the fraction of performance lost due to an imperfect L2 cache—the fraction of time spent waiting for L2 cache misses—is given by $I_{PerfectL2} - I_{Real}/I_{PerfectL2}$. (In Fig. 1, the benchmarks are ordered according to this metric.) The difference between these values is the fraction of time spent waiting for data to be fetched into the L1 caches from the L2. For the SPEC CPU2000 benchmarks, our system spends 61 percent of its time servicing L2 misses, 11 percent of its time servicing L1 misses, and only 28 percent of its time performing useful computation.

Since over half of our system's execution time is spent servicing L2 cache misses, the interface between the L2 cache and DRAM is a prime candidate for optimization. Unfortunately, diverse applications have highly variable memory system behaviors. For example, *art* has a high L2 stall fraction (73 percent) because it suffers almost 15 million L2 misses during the 200-million-instruction sample we ran, saturating the memory controller request bandwidth. At the other extreme, our 200M-instruction sample of *facerec* is latency bound, not bandwidth bound: It spends 64 percent of its time waiting for only 1.2 million DRAM accesses.

These varied behaviors imply that memory-system optimizations that improve performance for some applications may penalize others. For example, prefetching may improve the performance of a latency-bound application, but will decrease the performance of a bandwidth-bound application by consuming scarce bandwidth and increasing queuing delays [4]. Conversely, reordering memory references to increase DRAM bandwidth [5], [13], [20], [21], [24] may not help latency-bound applications, which rarely

- *W.-f. Lin and S.K. Reinhardt are with the Electrical Engineering and Computer Science Department, University of Michigan, Ann Arbor, MI 48109. E-mail: wflin@umich.edu, stever@eecs.umich.edu.*
- *D. Burger is with the Department of Computer Sciences, University of Texas at Austin, Taylor Hall 2.124, Austin, TX 78712-1188. E-mail: dburger@cs.utexas.edu.*

Manuscript received 5 Dec. 2000; revised 25 May 2001; accepted 31 May 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114258.

1. This equation is equivalent to $(CPI_{Real} - CPI_{PerfectMem,})/CPI_{Real}$, where CPI_X is the cycles per instruction for system X.

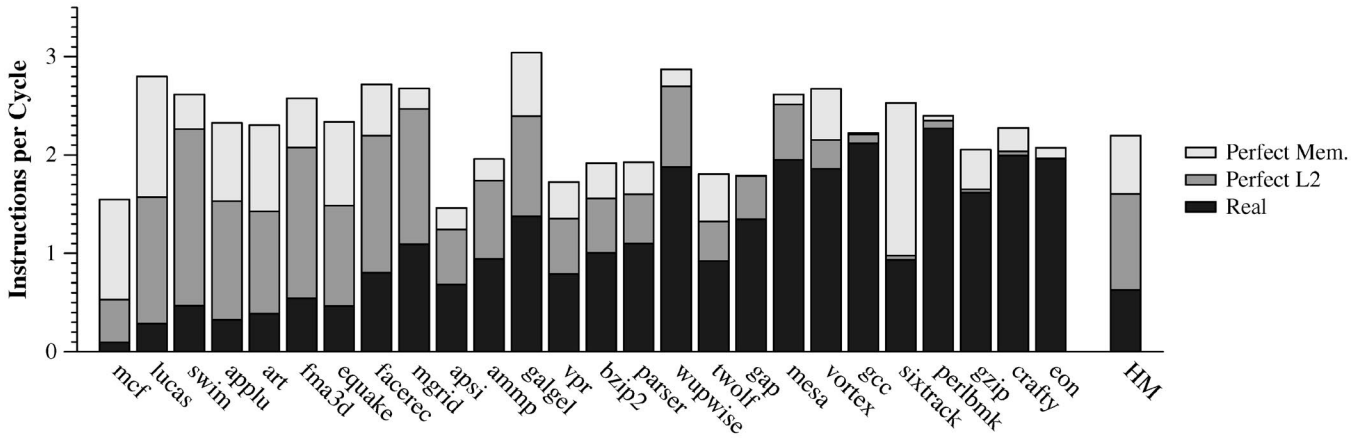

Fig. 1. Processor performance for SPEC2000.

issue concurrent memory accesses—and may even hurt performance by increasing latency.

In this paper, we describe techniques to reduce level-two miss latencies for memory-intensive applications that are not bandwidth bound. These techniques complement the current trend in newer DRAM architectures, which provide increased bandwidth without corresponding reductions in latency [7]. The techniques that we evaluate, in addition to improving the performance of latency-bound applications, avoid significant performance degradation for bandwidth-intensive applications.

Our primary contribution is a proposed prefetching engine specifically designed for level-two cache prefetching on a Direct Rambus memory system. The prefetch engine utilizes *scheduled region prefetching*, in which blocks spatially near the addresses of recent demand misses are prefetched into the L2 cache only when the memory channel would otherwise be idle. We show that the prefetch engine improves memory system performance substantially (11 percent to 145 percent) for 10 of the 26 benchmarks we study. We see smaller improvements for the remaining benchmarks, limited by lower prefetch accuracies, a lack of available memory bandwidth, or few L2 misses. Our prefetch engine is unintrusive: With four memory channels, all benchmarks show improvements with the prefetching. With eight memory channels and larger cache blocks, some benchmarks show slight performance drops, but the mean eight-channel improvement across the suite is 9 percent overall and 29 percent for the 10 benchmarks with high spatial locality.

Three mechanisms minimize the potential negative aspects of aggressive prefetching: prefetching data only on idle memory-channel cycles, scheduling prefetches to maximize hit rates in both the L2 cache and the DRAM row buffers, and placing the prefetches in a low-priority position in the cache sets, reducing the impact of cache pollution.

The remainder of the paper begins with a brief description of near-future memory systems in Section 2. In Section 3, we show how address mappings and miss scheduling may be tuned to improve the memory system. In Section 4, we study the impact of block size, memory bandwidth, and address mapping on performance. In Section 5, using the results from Section 3 and Section 4 to set our baseline, we describe and evaluate our scheduled region prefetching engine. We discuss related work in Section 6 and draw our conclusions in Section 7.

2 High-Performance Memory Systems

The two most important trends affecting the design of high-performance memory systems are integration and direct DRAM interfaces. Imminent transistor budgets permit both megabyte-plus level-two caches and DRAM memory controllers on the same die as the processor core, leaving only the actual DRAM devices off chip. Highly banked DRAM systems, such as double-data-rate synchronous DRAM (DDR SDRAM) and Direct Rambus DRAM (DRDRAM), allow heavy pipelining of bank accesses and data transmission. While the system we simulate in this work models DRDRAM channels and devices, the techniques we describe herein are applicable to other aggressive memory systems, such as DDR SDRAM, as well.

2.1 On-Chip Memory Hierarchy

Since level-one cache sizes are constrained primarily by cycle times and are unlikely to exceed 64KB [1], level-two caches are coming to dominate on-chip real estate. These caches tend to favor capacity over access time, so their size is constrained only by chip area. As a result, on-chip L2 caches of over a megabyte have been announced and multimegabyte caches will follow. These larger caches, with more sets, are less susceptible to pollution, making more aggressive prefetching feasible.

The coupling of high-performance CPUs and high-bandwidth memory devices (such as Direct Rambus) makes the system bus interconnecting the CPU and the memory controller both a bandwidth and a latency bottleneck [7]. With sufficient area available, high-performance systems will benefit from integrating the memory controller with the processor die, in addition to the L2 cache. That integration eliminates the system-bus bottleneck and enables high-performance systems built from an integrated CPU and a handful of directly connected DRAM devices. At least two high-performance chips—the Sun UltraSPARC-III and

2. Intel CPUs currently maintain their memory controllers on a separate chip. This organization allows greater product differentiation among multiple system vendors—an issue of less concern to Sun and Compaq.

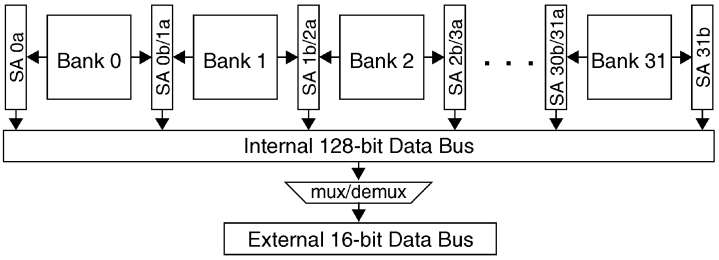

Fig. 2. Rambus shared sense-amp organization.

Compaq 21364—are following this route.[2] In this study, we are exploiting that integration in two ways. First, the higher available bandwidth again allows more aggressive prefetching. Second, we can consider closer communication between the L2 cache and memory controller so that L2 prefetching can be influenced by the state of the memory system—such as which DRAM rows are open and which channels are idle—contained in the controller.

2.2 Direct Rambus Architecture

Direct Rambus (DRDRAM) [6] systems obtain high bandwidth from a single DRAM device using aggressive signaling technology. Data are transferred across a 16-bit data bus on both edges of a 400-MHz clock, providing a peak transfer rate of 1.6 Gbytes per second. DRDRAMs employ two techniques to maximize the actual transfer rate that can be sustained on the data bus. First, each DRDRAM device has multiple banks, allowing pipelining and interleaving of accesses to different banks. Second, commands are sent to the DRAM devices over two independent control buses (a 3-bit row bus and a 5-bit column bus). Splitting the control buses allows the memory controller to send commands to independent banks concurrently, facilitating greater overlap of operations than would be possible with a single control bus. In this paper, we focus on the 256-Mbit Rambus device, the most recent for which specifications are available. This device contains 32 banks of 1 megabyte each. Each bank contains 512 rows of 2 kilobytes per row. The smallest addressable unit in a row is a *dualoct*, which is 16 bytes.

A full Direct Rambus access involves up to three commands on the command buses: precharge (PRER), activate (ACT), and, finally, read (RD) or write (WR). The PRER command, sent on the row bus, precharges the bank to be accessed, as well as releasing the bank's sense amplifiers and clearing their data. Once the bank is precharged, an ACT command on the row bus reads the desired row into the sense-amp array (also called the *row buffer* or *open page*). Once the needed row is in the row buffer, the bank can accept RD or WR commands on the column bus for each dualoct that must be read or written.[3]

RD and WR commands can be issued immediately if the correct row is held open in the row buffers. Open-row policies hold the most recently accessed row in the row buffer. If the next request falls within that row, then only RD or WR commands need be sent on the column bus. If a row buffer miss occurs, then the full PRER, ACT, and RD/WR sequence must be issued. Closed-page policies, which are better for access patterns with little spatial locality, release the row buffer after an access, requiring only the ACT-RD/WR sequence upon the next access.

A single, contentionless dualoct access that misses in the row buffer will incur 77.5 ns on the 800-40 256-Mbit DRDRAM device. PRER requires 20 ns, ACT requires 17.5 ns, RD or WR requires 30 ns, and data transfer requires 10 ns (eight 16-bit transfers at 1.25 ns per transfer). An access to a precharged bank therefore requires 57.5 ns and a page hit requires only 40 ns.

A row miss occurs when the last and current requests access different rows within a bank. The DRDRAM architecture incurs additional misses due to sense-amp sharing among banks. As shown in Fig. 2, row buffers are split in two and each half-row buffer is shared by two banks; the upper half of bank n's row buffer is the same as the lower half of bank $n + 1$'s row buffer. This organization permits twice the banks for the same number of sense-amps, but imposes the restriction that only one of a pair of adjacent banks may be active at any time. An access to bank 1 will thus flush the row buffers of banks 0 and 2 if they are active, even if the previous access to bank 1 involved the same row.

3 Basic Memory System Parameters

In this section, we measure the effects of address mappings and demand miss ordering, exploring the extent to which we can improve memory system performance without changing the cache or channel parameters. We show that, by remapping the DRAM bank ordering, we can increase the number of row buffer hits by 51 percent on average, causing a 16 percent overall increase in performance across our benchmark suite. We also show that a DRAM-aware miss scheduling policy improves overall performance an additional 6 percent. These optimizations form the baseline that we use in the subsequent sections.

3.1 Experimental Methodology

We simulated our target systems with an Alpha-ISA derivative of the SimpleScalar tools [3]. We extended the tools with a memory system simulator that models contention at all buses, finite numbers of MSHRs (Miss Status Holding Registers [16]), and Direct Rambus memory channels and devices in detail [6].

3. Most DRAM device protocols transfer write data along with the column address, but defer the read data transfer to accommodate the access latency. In contrast, DRDRAM data transfer timing is similar for both reads and writes, simplifying control of the bus pipeline and leading to higher bus utilization.

TABLE 1
Microarchitectural Simulation Parameters

Front end	
L1 Instruction cache	64KB, 2-way, 64B blocks, 1-cycle hit
Branch predictor	16K-entry local/global hybrid
Branch target buffer	2-way, 256 entry
Execution core	
Clock, issue width	1.6 GHz, 4-way issue
Issue window/reorder buffer (RUU)	64 entries
Load/store queue	64 entries
Functional units	Same latencies as 21264
Memory system	
L1 Data cache	64KB, 2-way, 64B blocks, 3-cycle hit
L2 cache	1MB, 4-way, 64B blocks, 12-cycle hit
MSHRS	8 per cache
L1/L2 bus	128 bits
DRDRAM	256MB with 800MHz channels

Although the SimpleScalar microarchitecture is based on the Register Update Unit [27]—a microarchitectural structure that merges the reorder buffer, physical register file, and issue window—we chose the rest of the parameters to match the Compaq Alpha 21364 [12] as closely as possible, shown in Table 1. In addition, our target system uses multiple DRDRAM channels in a simply interleaved fashion, i.e., n physical channels are treated as a single logical channel of n times the width. We selected the 1.6GHz clock rate as it is both near the maximum clock rate announced for current and near-future products and because it is exactly twice the effective frequency of the DRDRAM channels.

We evaluated our simulated systems using the 26 SPEC CPU2000 benchmarks compiled with recent Compaq compilers (C V5.9-008 and Fortran V5.3-915).[4] We simulated a 200-million-instruction sample of each benchmark running the reference data set after 20, 40, or 60 billion instructions of execution. We verified that cold-start misses did not impact our results significantly by simulating our baseline configuration assuming that all cold-start accesses are hits. This assumption changed IPCs by 1 percent or less on each benchmark.

The memory system behavior of the SPEC2000 benchmarks varies significantly across the suite, so presenting mean results of all benchmarks would obscure important behavioral differences. We divide the 26 benchmarks into four categories: `C-cpu`, `C-local`, `C-nolocal`, and `C-bw`. The `C-cpu` class contains those benchmarks that miss infrequently in the level-two cache and are thus bounded by CPU throughput or L1 misses. We place all the benchmarks that incur fewer than 0.75 L2 misses per thousand instructions with a 1MB L2 cache into `C-cpu`. Those benchmarks include *crafty*, *eon*, *gzip*, *gcc*, *perlbmk*, *sixtrack*, and *vortex*. Note that these correspond to the rightmost seven benchmarks in Fig. 1. In `C-bw`, we include the benchmarks that have such high L2 miss rates that the Rambus channels have little free bandwidth. Only *art* falls into that category. In `C-local`, we list the benchmarks that show high spatial locality, which includes any benchmark not in the previous two classes that has an optimal block size (for our base system to be described later) of at least 512 bytes: *applu*, *equake*, *facerec*, *gap*, *mesa*, *mgrid*, *parser*, *swim*, and *wupwise*. We also include *fma3d* in this category, although its performance-optimal block size is 256 bytes. The rest of the benchmarks, which exhibit frequent L2 misses, are not bandwidth-bound, and have performance-optimal block sizes of 256 bytes or less, fall into the `C-nolocal` category. Those are *ammp*, *apsi*, *bzip2*, *galgel*, *lucas*, *mcf*, and *twolf*.

In Table 2, we summarize the memory system behavior of the four benchmark classes on our base system (including the optimizations described in Sections 3.2 and 3.3). The first two columns show the number of L2 misses per thousand instructions and their mean latency. The next four columns characterize the behavior of the DRDRAM subsystem. The DRDRAM service time reflects the column access time plus precharge and row activation time, if required. These initial measurements reflect our base 64-byte block size. The final two columns indicate, as the block size is increased up to 8K bytes, the block sizes that maximize performance and minimize miss rate, respectively (referred to as the performance point and pollution point in Section 4.1). The `C-cpu` benchmarks, naturally, incur a low number of L2 misses. These infrequent misses experience little queuing delay at the DRAM controller, but have a higher service time since many of them are row-buffer misses. The `C-local` benchmarks have high miss rates due to the 64-byte blocks in this simulated base case, but show substantially reduced miss rates at larger block sizes. Their DRAM service time—and, hence, L2 miss latency—is low because the spatial locality in these benchmarks translates into a high row-buffer hit rate. The `C-nolocal` benchmarks show the highest miss latencies due to both queuing delay and the poorest locality in the DRAM row buffers. The `C-bw` benchmark shows the highest miss rate, leading to the largest queuing delays, though with the best row-buffer hit rate and lowest DRAM

4. We used the "peak" compiler options from the Compaq-submitted SPEC results, except that we omitted the profile-feedback step. Furthermore, we did not use the "-xtaso-short" option that defaults to 32-bit (rather than 64-bit) pointers.

TABLE 2
Memory System Comparison of Four Benchmark Classes

Category	L2 misses		DRDRAM measurements				Block size	
	per thousand instructions	average latency	queuing delay	service time	data bus utilization	row buffer hit rate	best perf.	fewest misses
C-cpu	0.32	110.1	3.3	82.9	1.0%	0.684	128B	512B
C-local	11.61	104.9	12.0	72.0	21.4%	0.869	1KB	8KB
C-nolocal	22.71	150.8	30.1	99.7	18.0%	0.415	128B	1KB
C-bw	73.59	138.5	57.6	67.9	61.4%	0.938	64B	8KB
Overall	14.37	121.7	17.0	83.3	16.4%	0.682	128B	2KB

service time. However, due to limited bandwidth, this locality does not translate into better performance at larger block sizes as it does for the `C-local` benchmarks.

3.2 Address Mapping

In all DRAM architectures, the best performance is obtained by maximizing the number of row-buffer hits while minimizing the number of bank conflicts. Both these numbers are strongly influenced by the manner in which physical processor addresses are mapped to the channel, device, bank, and row coordinates of the Rambus memory space. Optimizing this mapping improves performance on our benchmarks by 16 percent on average, with improvements for several benchmarks above 40 percent.

In Fig. 3a, we depict a standard address mapping. The horizontal bar represents the physical address, with the high-order bits to the left. The bar is segmented to indicate how fields of the address determine the corresponding Rambus device, bank, and row.

Starting at the right end, the low-order four bits of the physical address are unused since they correspond to offsets within a dualoct. In our simply interleaved memory system, the memory controller treats the physical channels as a single wide logical channel, so an n-channel system contains n times wider rows and fetches n dualocts per access. Thus, the next least-significant bits correspond to the channel index. In our base system with four channels and 64-byte blocks, these channel bits are part of the cache block offset.

The remainder of the address mapping is designed to leverage spatial locality across cache-block accesses. As physical addresses increase, adjacent blocks are first mapped contiguously into a single DRAM row (to increase the probability of a row-buffer hit), then are striped across devices and banks (to reduce the probability of a bank conflict). Finally, the highest-order bits are used as the row index.

In Table 3, we show the row buffer hit and miss rates for our scheduling policies. The standard address mapping provides a reasonable row-buffer hit rate on read accesses (50 percent on average), but achieves a mere 26 percent hit rate on writebacks. This difference is due to an anomalous interaction between the cache indexing function and the address mapping scheme. For a 1MB cache, the set index is formed from the lower 18 bits ($\log_2(1\mathrm{MB}/4)$) of the address. Each of the blocks that map to a given cache set will be identical in these low-order bits and will vary only in the upper bits. With the mapping shown in Fig. 3a, these blocks will map to different rows of the same bank in a system with only one device per channel, guaranteeing a bank conflict between a miss and its associated writeback. With two devices per channel, the blocks are interleaved across a pair of banks (as indicated by the vertical line in the figure), giving a 50 percent conflict probability.

One previously described solution is to exchange some of the row and column index bits in the mapping [35], [31]. If the bank and row are largely determined by the cache index, then the writeback will go from being a likely bank conflict to a likely row-buffer hit. However, by placing noncontiguous addresses in a single row, spatial locality is reduced.

Our solution, shown in Fig. 3b, XORs the initial device and bank index values with the lower bits of the row address to generate the final device and bank indices. This mapping retains the contiguous-address striping properties of the base mapping, but "randomizes" the bank ordering, distributing the blocks that map to a given cache set evenly across the banks. Zhang et al. [33] independently developed, and thoroughly analyzed, a similar scheme. As a final Rambus-specific twist, we move the low-order bank index bit to the most-significant position. This change stripes addresses across all the even banks successively, then

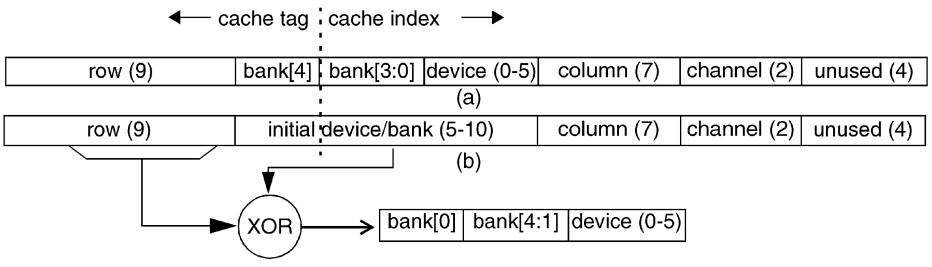

Fig. 3. Mapping physical addresses to Rambus coordinates.

TABLE 3
Effect of Mapping Change on Row Buffer Hit Rates and Performance

	Simple mapping				XOR mapping			
	IPC	Row Buffer Hit Rate			IPC	Row Buffer Hit Rate		
		Reads	Writes	Overall		Reads	Writes	Overall
C-cpu	1.686	0.585	0.213	0.540	1.695	0.705	0.418	0.684
C-local	0.712	0.516	0.339	0.467	0.941	0.898	0.799	0.869
C-nolocal	0.390	0.358	0.139	0.293	0.430	0.477	0.271	0.414
C-bw	0.388	0.888	0.627	0.866	0.468	0.958	0.757	0.940
Mean	0.630	0.500	0.255	0.448	0.733	0.719	0.532	0.682

TABLE 4
Effect of Scheduling on Performance (IPC)

	in-order open-page	in-order closed-page	row scheduling	row+col scheduling
C-cpu	1.695	1.672	1.695	1.697
C-local	0.941	0.514	0.943	0.969
C-nolocal	0.430	0.268	0.453	0.475
C-bw	0.468	0.115	0.469	0.477
Mean	0.733	0.418	0.753	0.779

across all the odd banks, reducing the likelihood of an adjacent buffer-sharing conflict (see Section 2.2).

As a result, we achieve a row-buffer hit rate of 72 percent for read accesses and 53 percent for writebacks. This final address mapping, which will be used for the remainder of our studies, improves performance by 16 percent on average and considerably more for four benchmarks: 66 percent for *applu*, 49 percent for *fma3d*, 43 percent for *swim*, and 39 percent for *facerec*.

3.3 Row Buffering and Scheduling Policies

In advanced DRAM architectures, accesses to independent banks may be carried out concurrently to provide increased bandwidth. Direct Rambus provides a large number of banks per device to maximize the potential for concurrency. The extent to which a system can exploit this feature for higher performance is a function of both the number of concurrent memory accesses that can be scheduled and the memory controller's ability to overlap these accesses. In this section, we examine the effect of the memory controller's scheduling policy on application performance in our baseline system. The scheduling policy represents a trade-off between implementation complexity and performance. DRAM access scheduling has been examined previously in the context of vector and streaming applications [5], [13], [20] ,[21], [24], but not on the broader class of benchmarks studied in this work.

With an open-page policy, a memory access that hits in an open row buffer requires no row-bus commands, while a row-buffer miss requires two: a precharge and an activate. In a closed-page policy, every access requires a single activate command on the row buffer. In any case, one column-bus command (RD or WR) is needed for each dualoct transferred.[5]

5. The precharge required by the closed-page policy can be piggybacked on the RD or WR.

The simplest scheduler we examine, used for the address mapping study in the previous section, handles requests in strict first-come, first-served order. Command packets are issued as quickly as possible without allowing packets from a later request to bypass those of an earlier request. This simple algorithm achieves overlap between requests due to the inherent pipelining of the Direct Rambus interface: Row commands from one request can overlap the column commands of the previous request, which in turn can overlap the data transfer of an earlier access.

Using this scheduler, we investigated the performance impact of open-page vs. closed-page policies (Section 2.2). As shown in the first two columns of Table 4, the benefit of the row-buffer hits afforded by the open-page policy more than compensates for the added latency of precharging a bank on a row-buffer miss. The baseline experiments used for the remainder of the paper assume an open-page policy.

The efficiency of this simple scheduler is limited by Rambus interpacket timing constraints. For example, precharge and activate commands for a given row must be separated by four bus cycles; row-bus commands to the same device must also be separated by four cycles. There is also a fixed delay (dependent on the device speed) between an activate command and a column-bus RD or WR to the activated row. Because the scheduler only considers a single packet per bus, the bus idle time induced by these constraints cannot be hidden.

To increase bus efficiency, a more advanced scheduler would attempt to issue a packet associated with a later request if the next packet needed by the oldest request cannot issue. Our more aggressive scheduler implementation maintains a queue of pending memory requests in the order they are received. The size of this queue is bounded by the maximum number of outstanding L2 accesses plus the writeback buffer depth. Each queue entry tracks the next command packet required to advance the corresponding request. When a command bus becomes idle, the

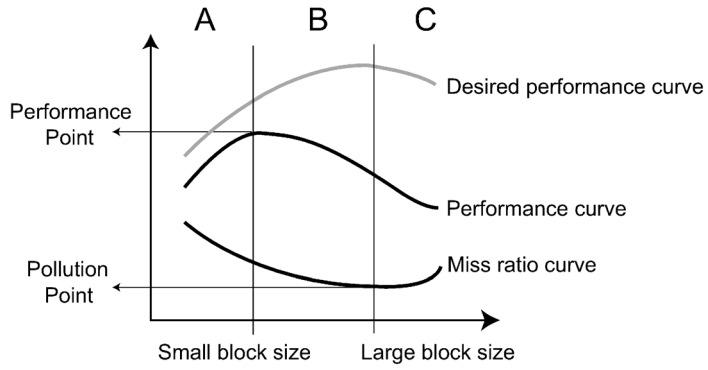

Fig. 4. Illustration of performance and pollution points.

scheduler scans the queue, issuing the first packet it finds for that bus that meets the Rambus interpacket timing constraints. To prevent a row buffer from thrashing among multiple rows, the scheduler will only consider the first request that requires a particular buffer.

We consider two aggressive schedulers which use this basic strategy; their performance is depicted in Table 4. The first schedules only the row bus; column-bus commands (dualoct reads and writes) are issued in order as in the simple scheduler. This technique allows interleaving of precharge and activate commands, but prevents a request's column data access from being delayed by that of a later request. This policy increases the harmonic-mean IPC by 2.7 percent over the simple scheduler. Most of the benefit is concentrated in a few benchmarks: *galgel*, *lucas*, and *mcf* improve by over 6 percent; *apsi*, *vortex*, and bzip improve between 2 percent and 4 percent. No other benchmark achieves more than 0.6 percent improvement and the median speedup is a mere 0.2 percent. The key factor that limits the benefit of scheduling is that many of the benchmarks do not have enough concurrent DRAM accesses to permit significant interleaving.

Our second and most aggressive scheduler reorders packets on both the row and column buses, trading the potential for an occasional increase in request latency for more efficient column-bus utilization. The harmonic-mean IPC improves an additional 3.5 percent, a net 6.3 percent improvement over the simple scheduler. The benefits are somewhat more widespread, with 11 benchmarks seeing an improvement of 3 percent or more. Three benchmarks (*apsi*, *lucas*, and *mcf*) improve 5 percent or more relative to the row-bus scheduler. Unlike the row-bus-only scheduler, it is possible for this aggressive scheduler to degrade performance relative to the in-order scheduler; however, only *eon* suffers a negligible (0.1 percent) slowdown. This aggressive scheduling policy will be used as the baseline policy for the remainder of this paper.

4 Trade-Offs in Exploiting Spatial Locality

In this section, we examine the L2 cache and memory channel organization, exploring the trade-offs between spatial locality and pollution versus bandwidth and memory channel contention. Many of the SPEC2000 benchmarks have significant spatial locality that they are unable to exploit due to memory channel contention. We show in this section that the block size that minimizes their miss rates is much larger than the block size that maximizes their performance. We use those results to motivate the prefetching engine described in Section 5.

4.1 Block Size, Contention, and Pollution

Increasing a cache's block size—generating large, contiguous transfers between the cache and DRAM—is a simple way to exploit additional memory system bandwidth. If an application has sufficient spatial locality, larger blocks will reduce the miss rate as well. Of course, large cache blocks can also degrade performance. For a given memory bandwidth, larger fetches can cause bandwidth contention, i.e., increased queuing delays. Larger blocks may also cause cache pollution because a cache of fixed size holds fewer unique blocks.

As L2 capacities grow, the corresponding growth in the number of blocks will reduce the effects of cache pollution. Larger L2 caches may also reduce bandwidth contention since the overall miss rate will be lower. Large L2 caches may thus benefit from larger block sizes, given sufficient memory bandwidth and spatial locality.

For any cache, as the block size is increased, the effects of bandwidth contention will eventually overwhelm any reduction in miss rate. We define this transition—at which bandwidth contention overcomes the benefits from reduced miss rates—as the *performance point*: the block size at which performance is highest. As the block size is increased further, cache pollution will eventually overwhelm spatial locality. We define this transition as the *pollution point*: the block size at which the miss rate is lowest. While the general effects of pollution and contention are well-known, this work is the first, to our knowledge, to formalize and name these inflection points. Furthermore, these points have not previously been quantified for systems in which a substantial fraction of the memory hierarchy is on chip.

Fig. 4 illustrates these points abstractly. In region A of the figure, larger blocks (farther to the right on the x-axis) improve the miss rate, which provides a performance gain that overcomes the performance drop due to increased channel contention, causing overall performance to rise. In region B, the miss ratio continues to drop with larger blocks, but performance also deteriorates due to increased contention. The performance point resides at the boundary between regions A and B. Finally, in region C, pollution causes the miss rate to begin to increase with larger blocks,

causing a sharper drop in performance than in region B. The pollution point resides at the boundary between regions B and C. The top gray line indicates the potential performance gain if spatial locality could be exploited without incurring bandwidth contention—the effect we seek to achieve with the scheduled region prefetching technique described in Section 5.

In Table 5, we show the pollution and performance points for our benchmarks assuming four DRDRAM channels, providing 6.4 GB/s peak bandwidth. The benchmarks are arranged by category, as discussed in Section 3.1. For each benchmark, the table gives the IPC and L2 misses per thousand instructions across a range of block sizes. The performance and pollution points are in boldface.

The most important result in Table 5 is that the pollution points are at block sizes much larger than typical L2 block sizes (e.g., 64 bytes in the Alpha 21264), averaging 2KB. Nearly half of the benchmarks show pollution points at 8KB, which was the maximum block size we measured (larger blocks would have exceeded the virtual page size of our target machine). After computing the harmonic mean of the IPCs at each block size, we find that the mean performance point resides at 128-byte blocks. While the performance points all reside in the smaller block sizes (generally 64 to 256 bytes), the pollution points are widely scattered across the benchmarks, depending on the amount of spatial locality in the application. The goal of our prefetching scheme is to exploit the spatial locality in the benchmarks that have that locality without degrading the applications whose pollution points reside at 64 or 128 byte blocks. For example, five of the `C-local` benchmarks have performance points at block sizes larger than 256 bytes, but all of the benchmarks in `C-bw` and `C-nolocal` have performance points between 64 and 256 bytes.

Furthermore, the miss rates at the pollution points are significantly lower than at the performance points: more than a factor of two for half of the benchmarks, and more than tenfold for seven of them. The differences in performance (IPC) at the pollution and performance points are significant, but less pronounced than the miss rate differences at those same points.

For benchmarks that have low L2 miss rates, the gap between the pollution and performance points makes little difference to overall performance since misses are infrequent. For the rest of the benchmarks, however, an opportunity clearly exists to improve performance beyond the performance point since there is additional spatial locality that can be exploited before reaching the pollution point. The key to improving performance is to exploit this locality without incurring the bandwidth contention induced by larger fetch sizes.

4.2 Channel Width

Emerging systems contain a varied number of Rambus channels. Systems based on the Intel Pentium 4 processor currently contain one or two channels, depending on the price/performance target. The Alpha 21364, however, will contain up to a maximum of eight channels, managed by two controllers.

Higher-bandwidth systems reduce contention, allowing larger blocks to be fetched with overhead similar to smaller blocks on a narrower channel. In Table 6, we show the effect of the number of physical channels on performance at various block sizes. In these experiments, we held the total number of DRDRAM devices in the memory system constant, resulting in fewer devices per channel as the number of channels was increased. The numbers shown in the table are the harmonic mean of the IPC measured for all of the SPEC2000 benchmarks at each block size and channel width.

As the number of channels increases, the performance point (shown in bold) also increases; the reduced impact of contention means that larger blocks can be fetched. The performance point across channel widths is surprisingly regular: For two or more channels, the best block size always occurs at 32 bytes (two dualocts) per channel. Thus, for a four-channel system, the performance point resides at 128-byte blocks.

Our data show that the best overall performance is obtained using a block size of 1 KB—given a 32-channel (51.2 GB/s) memory system. This result indicates that our 1 MB cache is sufficiently large to mitigate the impact of pollution on average (though some individual benchmarks do suffer). Given sufficient bandwidth, then, large block sizes can effectively exploit spatial locality. However, achieving this bandwidth is prohibitively expensive for next-generation systems. The scheduled region prefetching technique described in the following section exploits this locality without inducing bandwidth contention, requiring less aggregate memory bandwidth and thus providing a more cost-effective solution.

5 Improving Performance with Scheduled Region Prefetching

The four-channel, 64-byte block baseline with the XORed bank mapping recoups some of the performance lost due to off-chip memory accesses. In this section, we propose improving memory system performance further using *scheduled region prefetching*. On a demand miss, blocks in an aligned region surrounding the miss that are not already in the cache are prefetched [28]. For example, a cache with 64-byte blocks and 4KB regions would fetch the 64-byte block upon a miss and then prefetch any of the 63 other blocks in the surrounding 4KB region not already resident in the cache. A key feature of our scheme is that, to avoid contention with demand misses, we issue prefetches only when the Rambus channels are otherwise idle.

We depict our prefetch controller in Fig. 5. The prefetch queue maintains a list of n region entries. Each entry corresponds to an aligned memory region and contains a bit vector representing the cache blocks within the region. A bit in the vector is set if the corresponding block is not in the cache and thus is a candidate for prefetching.

When a demand miss occurs in a region that does not match an entry in the prefetch queue, a new entry corresponding to the accessed region replaces an existing queue entry. The prefetch prioritizer selects a queue entry for prefetching; within a region, candidate blocks are fetched in linear order starting with the block after the demand miss and wrapping around. The selected prefetch

TABLE 5
Pollution vs. Performance Points

		Block size							
Benchmark	**Metric**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**
crafty	IPC	2.00	**2.00**	2.00	1.99	1.95	1.82	1.53	0.89
	Miss/Kinst	*0.12*	***0.10***	*0.10*	*0.12*	*0.16*	*0.23*	*0.27*	*0.38*
eon	IPC	1.96	1.96	1.96	1.96	**1.96**	1.96	1.96	1.96
	Miss/Kinst	*0.01*	*0.00*	*0.00*	*0.00*	***0.00***	*0.00*	*0.00*	*0.00*
gcc	IPC	0.94	0.94	**0.95**	0.94	0.90	0.80	0.64	0.43
	Miss/Kinst	*0.69*	*0.47*	*0.33*	***0.27***	*0.32*	*0.37*	*0.43*	*0.49*
gzip	IPC	1.63	1.64	1.64	1.64	**1.64**	1.64	1.64	1.64
	Miss/Kinst	*0.19*	*0.09*	*0.05*	*0.02*	*0.01*	*0.01*	*0.00*	***0.00***
perlbmk	IPC	2.29	2.30	**2.31**	2.30	2.27	2.16	1.84	1.27
	Miss/Kinst	*0.18*	*0.12*	*0.09*	*0.07*	***0.06***	*0.08*	*0.10*	*0.14*
sixtrack	IPC	2.14	2.16	2.16	2.15	2.16	**2.16**	2.16	2.16
	Miss/Kinst	*0.35*	*0.24*	*0.17*	*0.11*	*0.06*	*0.03*	*0.02*	***0.01***
vortex	IPC	1.86	**1.87**	1.86	1.81	1.64	1.17	0.63	0.32
	Miss/Kinst	*0.71*	***0.65***	*0.65*	*0.72*	*0.84*	*1.04*	*1.22*	*1.29*
C-cpu mean	IPC	1.70	**1.70**	1.70	1.69	1.64	1.49	1.18	0.78
	Miss/Kinst	*0.32*	*0.24*	*0.20*	***0.19***	*0.21*	*0.25*	*0.29*	*0.33*
applu	IPC	0.57	0.80	0.92	0.97	1.06	1.14	**1.11**	1.09
	Miss/Kinst	*16.49*	*8.30*	*4.19*	*2.12*	*1.06*	*0.53*	*0.27*	***0.13***
equake	IPC	0.53	0.71	0.80	0.87	**0.90**	0.90	0.89	0.88
	Miss/Kinst	*27.43*	*13.84*	*6.99*	*3.53*	*1.79*	*0.91*	*0.46*	***0.24***
facerec	IPC	1.15	1.34	1.36	1.35	1.39	1.41	**1.42**	1.41
	Miss/Kinst	*6.03*	*4.50*	*2.62*	*1.31*	*0.66*	*0.33*	*0.17*	***0.08***
fma3d	IPC	0.84	0.89	**0.92**	0.86	0.84	0.84	0.82	0.80
	Miss/Kinst	*27.05*	*15.15*	*8.24*	*4.50*	*2.26*	*1.14*	*0.59*	***0.31***
gap	IPC	1.53	1.65	1.71	1.75	1.76	**1.76**	1.75	1.73
	Miss/Kinst	*1.45*	*0.72*	*0.36*	*0.18*	*0.09*	*0.05*	*0.02*	***0.01***
mesa	IPC	2.17	2.32	2.38	**2.38**	2.23	1.40	0.80	0.53
	Miss/Kinst	*0.80*	*0.41*	*0.23*	***0.18***	*0.31*	*0.63*	*0.67*	*0.52*
mgrid	IPC	1.30	1.61	1.81	**1.82**	1.76	1.66	1.58	1.57
	Miss/Kinst	*7.06*	*3.63*	*1.93*	*1.08*	*0.60*	*0.31*	*0.17*	***0.08***
parser	IPC	1.24	1.38	1.44	**1.44**	1.41	1.31	1.11	0.77
	Miss/Kinst	*2.75*	*1.46*	*0.82*	*0.50*	*0.36*	***0.30***	*0.30*	*0.33*
swim	IPC	0.70	0.94	1.05	**1.12**	1.12	1.06	0.93	0.78
	Miss/Kinst	*24.69*	*12.38*	*6.23*	*3.16*	*1.63*	*0.89*	*0.52*	***0.33***
wupwise	IPC	1.97	2.16	2.28	**2.28**	2.19	2.18	2.18	2.18
	Miss/Kinst	*2.37*	*1.31*	*0.78*	*0.51*	*0.35*	*0.18*	*0.09*	***0.05***
C-local mean	IPC	0.97	1.19	1.28	1.31	**1.31**	1.26	1.13	0.98
	Miss/Kinst	*11.61*	*6.17*	*3.24*	*1.71*	*0.91*	*0.53*	*0.33*	***0.21***
ammp	IPC	**1.08**	1.03	0.94	0.74	0.44	0.21	0.16	0.11
	Miss/Kinst	*9.22*	*7.19*	*5.63*	*5.06*	*5.13*	*5.84*	*3.80*	***2.78***
apsi	IPC	0.87	0.90	**0.91**	0.81	0.64	0.45	0.07	0.02
	Miss/Kinst	*10.23*	*6.08*	*3.95*	*2.92*	*2.41*	***2.15***	*9.58*	*16.34*
bzip2	IPC	**1.06**	1.05	0.96	0.68	0.39	0.22	0.11	0.05
	Miss/Kinst	*3.42*	***3.21***	*3.63*	*4.40*	*5.03*	*4.93*	*5.10*	*5.51*
galgel	IPC	1.51	1.71	**1.77**	1.74	1.56	0.85	0.10	0.04
	Miss/Kinst	*3.59*	*2.10*	*1.35*	*0.96*	***0.78***	*1.24*	*7.44*	*9.70*
lucas	IPC	0.35	0.39	**0.42**	0.31	0.17	0.09	0.04	0.02
	Miss/Kinst	*27.81*	*19.07*	*14.70*	*12.47*	***11.51***	*11.70*	*12.24*	*12.95*
mcf	IPC	0.12	**0.13**	0.12	0.08	0.04	0.02	0.01	0.01
	Miss/Kinst	*115.67*	*95.18*	*71.74*	*61.29*	***61.13***	*65.48*	*65.85*	*64.39*
twolf	IPC	0.95	0.97	**0.98**	0.94	0.79	0.51	0.24	0.11
	Miss/Kinst	*3.66*	*3.13*	*2.77*	*2.42*	***2.20***	*2.21*	*2.58*	*2.84*
vpr	IPC	**0.83**	0.80	0.71	0.51	0.29	0.14	0.06	0.03
	Miss/Kinst	***8.09***	*8.33*	*8.89*	*9.47*	*9.92*	*10.61*	*11.52*	*12.80*
C-nolocal mean	IPC	0.47	**0.49**	0.46	0.34	0.20	0.10	0.05	0.02
	Miss/Kinst	*22.71*	*18.04*	*14.08*	*12.37*	***12.26***	*13.02*	*14.76*	*15.91*
art	IPC	**0.48**	0.36	0.30	0.30	0.30	0.28	0.27	0.27
	Miss/Kinst	*73.59*	*65.11*	*43.04*	*21.54*	*10.78*	*5.40*	*2.71*	***1.36***
C-bw mean	IPC	**0.48**	0.36	0.30	0.30	0.30	0.28	0.27	0.27
	Miss/Kinst	*73.59*	*65.11*	*43.04*	*21.54*	*10.78*	*5.40*	*2.71*	***1.36***
Overall mean	IPC	0.78	**0.82**	0.80	0.68	0.47	0.27	0.14	0.07
	Miss/Kinst	*14.37*	*10.49*	*7.29*	*5.34*	*4.59*	***4.48***	*4.85*	*5.12*

TABLE 6
Block Size vs. Channel Numbers

	# channels					
Block size	**1**	**2**	**4**	**8**	**16**	**32**
64	**0.499**	**0.671**	0.779	0.803	0.803	0.803
128	0.426	0.655	**0.821**	0.861	0.881	0.881
256	0.327	0.554	0.801	**0.929**	0.963	0.976
512	0.231	0.416	0.677	0.919	**1.015**	1.035
1024	0.140	0.264	0.468	0.742	0.976	**1.059**

block is provided to the access prioritizer, which forwards it to the Rambus controller only when there are no demand misses or writebacks pending. Channel contention thus occurs only when a demand miss arrives while a prefetch is in progress.

Sections 5.1-5.3 describe enhancing a scheduled region prefetching scheme by controlling the replacement priority of prefetches, improving scheduling in the prefetch prioritizer, and optimizing the region size and the depth of the prefetch queue. Section 5.4 summarizes the performance of the enhanced scheme. Sections 5.5-5.8 analyze the scheme's bandwidth utilization, its performance with varying cache sizes and DRAM latencies, and its interaction with software prefetching.

5.1 Insertion Policy

When prefetching directly into the L2 cache, the likelihood of pollution is high if the prefetch accuracy is low. In this section, we describe how to mitigate that pollution for low prefetch accuracies by assigning a lower replacement priority to prefetched data than to demand-miss blocks.

Our simulated 4-way set-associative cache uses the common least-recently-used (LRU) replacement policy. A block may be loaded into the cache with one of four priorities: *most-recently-used* (MRU), *second-most-recently-used* (SMRU), *second-least-recently-used* (SLRU), and LRU. Normally, blocks are loaded into the MRU position. By loading prefetches into a lower-priority slot, we restrict the amount of referenced data that prefetches can displace. For example, if prefetches are loaded with LRU priority, they can displace at most one quarter of the referenced data in the cache.

In the left half of Table 7, we depict the arithmetic mean of the prefetch accuracies for our four classes of benchmarks, shown as the region prefetches are loaded into differing points on the replacement priority chain. In the right half of Table 7, we show the harmonic mean of IPC values for each benchmark class. In these experiments, we simulated 4KB prefetch regions with two entries per queue, 64-byte blocks, and four DRDRAM channels.

In general, the prefetch accuracy diminishes as prefetches are placed lower in the LRU chain because a prefetch is more likely to be evicted before it is referenced. However, many of the benchmarks display higher prefetch accuracies at SECMRU than MRU. This anomaly is due to the varying number of prefetches for the different insertion priorities. SECMRU placement incurs less pollution and the reduced number of misses generates significantly fewer total prefetches, which may in turn improve accuracy. However, most of the benchmarks show consistent drops in accuracy when prefetches are reduced in priority to SECLRU and all but two of the benchmarks show further accuracy reductions when prefetches are loaded at LRU priority.

For the `C-local` benchmarks, the prefetch accuracy decrease from SECLRU to LRU causes a drop in performance. However, since most of the `C-local` benchmarks quickly reference their prefetches, the impact is minor. For the `C-nolocal` benchmarks, relative performance is raised significantly (67 percent) as the prefetches are moved to LRU priority from MRU since the pollution effects are mitigated—the `C-nolocal` benchmarks display prefetch accuracies of only a few percent.

While replacement prioritization does not help benchmarks with high prefetch accuracies (`C-local`) significantly, it mitigates the adverse pollution impact of prefetching on the other benchmarks, just as scheduling mitigates the bandwidth impact. We assume LRU placement for the rest of the experiments in this paper.

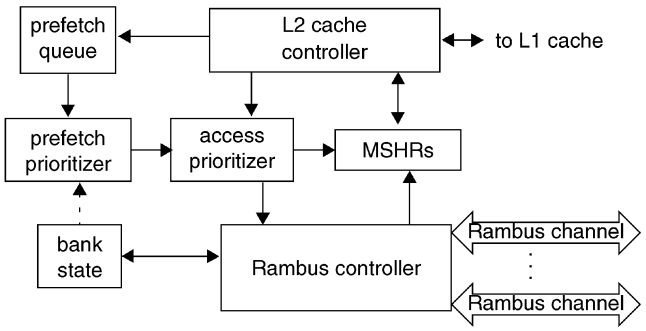

Fig. 5. Prefetching memory controller.

5.2 Prefetch Scheduling

Although the prefetch insertion policy diminishes the effects of cache pollution, simple aggressive prefetching can consume copious amounts of bandwidth, interfering with the handling of latency-critical misses. Table 8 shows the effects of various prefetch scheduling policies on L2 miss rate, L2 miss latency, and IPC. Miss latency is measured as the time spent returning an L2 miss, not the load-use delay for a load that misses in the L2 cache. The first column of results shows our base system (1MB, 4-way L2 cache with 64-byte blocks, XOR mapping, aggressive miss scheduling, and four DRDRAM channels) with no prefetching, labeled

TABLE 7
Effect of Prefetch Insertion Policy on Performance

	Accuracy				IPC			
	MRU	SECMRU	SECLRU	LRU	MRU	SECMRU	SECLRU	LRU
C-cpu	33.1%	32.2%	30.1%	25.2%	1.28	1.45	1.53	1.57
C-local	61.6%	62.6%	62.8%	58.1%	1.37	1.45	1.47	1.43
C-nolocal	3.1%	3.2%	3.1%	2.6%	0.06	0.08	0.09	0.10
C-bw	48.4%	48.0%	47.4%	46.7%	0.29	0.30	0.30	0.32
Mean	35.4%	35.6%	35.0%	31.7%	0.18	0.23	0.26	0.27

"base." The next five columns correspond to region prefetching with various prefetch scheduling policies.

In the second column, labeled "none," we add prefetching of 4KB regions, but do not schedule prefetches specially—all the prefetches generated by a miss will be issued before any subsequent demand miss. Unscheduled region prefetching avoids a substantial number of misses: the L2 miss rate is reduced from 36.4 percent to just 10.9 percent. Despite the sharp reduction in miss rate, contention increases the miss latencies dramatically. The (arithmetic) mean L2 miss latency, across all benchmarks, rises more than sevenfold, from 122 cycles to 890 cycles, due to contention caused by the substantial number of prefetches. Although the `C-local` benchmarks benefit from unscheduled prefetching in spite of the latency increase, overall performance drops significantly.

This large increase in channel contention can be avoided by issuing prefetch accesses only when the Rambus channel would otherwise be idle. When the Rambus controller is ready for an access, it signals an access prioritizer circuit, which forwards any pending L2 demand misses before it will forward a prefetch request from the prefetch queue, as depicted in Fig. 5.

Our simplest prefetch prioritizer uses a FIFO policy for issuing prefetches and for replacing regions. The oldest prefetch region in the queue has the highest priority for issuing requests to the Rambus channels and is also the region that is replaced when a demand miss adds another region to the queue. Results for this scheme are shown in the "fifo" column of Table 8.

Comparing scheduled to unscheduled prefetching, we see miss rates increase slightly because some prefetches will be displaced from the prefetch queue before they can be issued. However, most of the miss-rate reduction relative to the nonprefetching base case remains. Furthermore, in stark contrast to unscheduled prefetching, scheduled prefetching incurs only a small increase in mean L2 miss latency. The result is an across-the-board performance improvement. The `C-local` benchmarks show a mean 62 percent performance improvement—higher than that of unscheduled prefetching, despite a larger demand miss rate. More importantly, this prefetch scheme is unintrusive: Due to the combination of LRU priority insertion and channel scheduling, no benchmark shows a performance drop when prefetching is added to the base system. Across the entire SPEC2000 suite, performance shows a mean 24 percent increase with the addition of FIFO region prefetching.

We can further improve our prefetching scheme by taking into account not only the idle/busy status of the Rambus channel, but also the expected utility of the

TABLE 8
Comparison of Prefetch Policies

		base	none	fifo	lifo	lifo+lru	lifo+lru +bank
C-cpu	Miss rate	12.9%	2.6%	3.1%	2.9%	2.9%	2.9%
	Miss lat.	110.1	596.3	119.0	119.6	119.6	117.9
	IPC	1.697	1.566	1.715	1.718	1.719	1.720
C-nolocal	Miss rate	35.0%	23.6%	26.0%	25.8%	25.6%	25.5%
	Miss lat.	150.8	1718.9	159.7	159.5	160.3	158.5
	IPC	0.475	0.096	0.544	0.551	0.553	0.555
C-local	Miss rate	52.5%	6.2%	12.0%	11.8%	11.9%	11.8%
	Miss lat.	104.9	442.4	109.7	109.3	107.5	108.7
	IPC	0.969	1.433	1.572	1.580	1.604	1.595
C-bw	Miss rate	50.3%	14.3%	47.6%	47.6%	47.9%	48.1%
	Miss lat.	138.4	780.4	139.1	139.6	138.2	139.1
	IPC	0.477	0.319	0.537	0.536	0.541	0.537
Mean	Miss rate	36.4%	10.9%	15.3%	15.1%	15.0%	15.0%
	Miss lat.	121.7	889.6	128.7	128.7	128.2	127.7
	IPC	0.779	0.265	0.963	0.971	0.977	0.977

TABLE 9
Performance with Varied Queue Entries

	1	2	4	8	16	32
C-cpu	1.718	**1.720**	1.718	1.718	1.716	1.717
C-local	1.587	1.595	**1.596**	1.595	1.595	1.595
C-nolocal	0.553	**0.555**	0.550	0.544	0.538	0.534
C-bw	0.536	0.537	**0.539**	0.538	0.539	0.538
Mean	0.974	**0.977**	0.972	0.966	0.960	0.956

prefetch request and the state of the Rambus banks. These optimizations fall into three categories: *prefetch region prioritization*, *prefetch region replacement*, and *bank-aware scheduling*.

When large prefetch regions are used on an application with limited available bandwidth, prefetch regions are typically replaced before all of the associated prefetches are completed. The FIFO policy can then cause the system to spend most of its time prefetching from "stale" regions, while regions associated with more recent misses languish at the tail of the queue. We address this issue by changing to a LIFO algorithm for prefetching in which the highest-priority region is the one that was added to the queue most recently. We couple this with an LRU prioritization algorithm that moves queued regions back to the highest-priority position when a demand miss occurs within that region and replaces regions from the tail of the queue when it is full.

Finally, the row-buffer hit rate of prefetches can be improved by giving highest priority to regions that map to open Rambus rows. Prefetch requests will generate precharge or activate commands only if there are no pending prefetches to open rows. This optimization makes the row-buffer hit rate for prefetch requests nearly 100 percent and reduces the total number of row-buffer misses by 9 percent.

These optimizations, labeled "lifo" in column six of Table 8, "lifo+lru" in column seven, and "lifo+lru+bank" in column eight, improve the performance across all the applications, reducing the average miss rate by 0.3 percent, with an associated one-cycle reduction in miss latency over the "fifo" policy. The mean performance improvement, across all applications, increases to 25.3 percent.

5.3 Region and Queue Sizes

In addition to prefetch scheduling and prioritization, we experimented with various region prefetch-queue sizes. In Table 9, we show the harmonic mean IPC for each of the benchmark classes as the number of prefetch queue entries (each corresponding to one region) is varied from 1 to 32. These experiments were run using 4KB regions and included all of the scheduling optimizations described in the previous subsection. We were surprised that the number of regions at which performance was highest was so consistent across the benchmark classes and so low, at two to four regions. The `C-local` benchmarks are insensitive to the number of queue entries past two since the larger queue sizes do not end up producing substantially more prefetches. On the other hand, the low prefetch accuracy of the `C-nolocal` benchmarks favors keeping the queue size to a minimum.

In Table 10, we show the changes in performance for region prefetching with two queue entries for each benchmark class as the region size is varied from 1KB to 4KB. Performance increases for all classes as the region size is increased. We did not measure regions beyond 4KB since a region size larger than the virtual page size is not meaningful when using physical addresses. Given the consistency of these two results, all subsequent performance results using prefetching assume 4KB regions and two entries per prefetch queue, LIFO prefetching, LRU region upgrades, and scheduled prefetches.

To compare against previously published solutions, we also experimented with an unaligned prefetching scheme, in which the N blocks following the miss block are considered as prefetch candidates (rather than the N blocks in the surrounding aligned region), as proposed by Dahlgren et al. [8]. While the two schemes show similar performance improvements for small values of N, the aligned region scheme is superior for sufficiently large values of N since the utility of prefetching the preceding block eventually exceeds the utility of prefetching the Nth succeeding block as more blocks are prefetched. The aligned scheme has the additional benefits of eliminating virtual page crossings (as long as the region size is less than or equal to the page size) and a slightly simpler representation in the prefetch queue.

5.4 Performance Summary

Though scheduled region prefetching provides a mean performance increase over the entire SPEC suite, the benefits are concentrated in a subset of the benchmarks. In Fig. 6, we show detailed performance results for the 10 benchmarks in `C-local`. The leftmost bar for each benchmark is stacked, showing the IPC values for three targets. The 64-byte block, four-channel experiments with

TABLE 10
Prefetch Region Size Effect on Performance

	1K	2K	4K
C-cpu	1.718	1.718	1.720
C-local	1.523	1.572	1.595
C-nolocal	0.550	0.554	0.555
C-bw	0.528	0.533	0.537
Mean	0.961	0.972	0.977

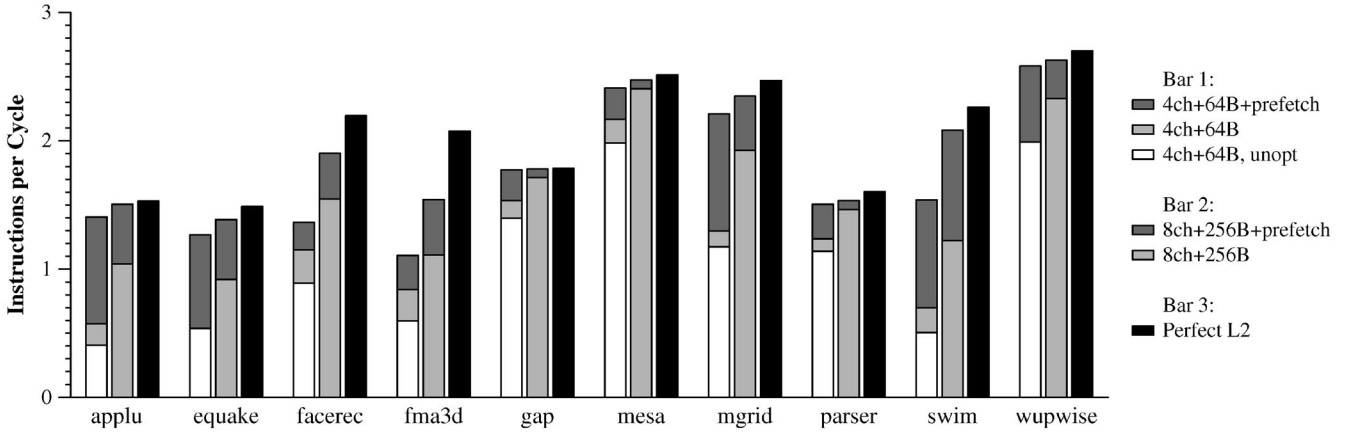

Fig. 6. Overall performance of tuned scheduled region prefetching.

the standard bank mapping is represented by the white bar, the XOR mapping improvement is represented by the middle, light gray bar, and region prefetching is represented by the top, dark gray bar. The second bar in each cluster shows the performance of 8-channel runs with a 256-byte block size (which is the performance point for eight channels) in light gray and the same system with region prefetching in dark gray. The rightmost bar in each cluster shows the IPC obtained by a perfect L2 cache.

On the 4-channel system, the address mapping tuning provides a mean 24 percent speedup for `C-local`. Adding prefetching results in an additional 65 percent speedup. For eight of the 10 benchmarks, the 4-channel prefetching experiments outperform the 8-channel system with no prefetching. The 8-channel, 256-byte block experiments with region prefetching show the highest attained performance, however, with a mean speedup of 90 percent over the tuned, 4-channel base case for the benchmarks depicted in Fig. 6 and a 30 percent speedup across all the benchmarks. The 8-channel system with 256-byte blocks and region prefetching comes within 10 percent of perfect L2 cache performance for eight of the 10 `C-local` benchmarks, as well as within 10 percent of a perfect L2 cache for the mean across these benchmarks.

5.5 Effect on Rambus Channel Utilization

The region prefetching scheme increases traffic on the memory channel for all of the benchmarks we studied. We quantify this effect by measuring the utilization of the data channels, which is simply the fraction of cycles during which data are transmitted. We show the channel utilizations for the four benchmark classes in Table 11, for a 4-channel system with 4KB, 2-entry region prefetching.

TABLE 11
Effect of Channel Policies on Utilization

	simple	aggressive	prefetch
C-cpu	1.0%	1.0%	12.6%
C-local	15.6%	21.4%	46.5%
C-nolocal	15.3%	18.0%	66.2%
C-bw	50.4%	61.4%	80.5%
Mean	12.9%	16.4%	44.7%

The first two columns of Table 11 show the channel utilization without prefetching for our simplest configuration (simple address mapping and in-order Rambus scheduling) and our more aggressive baseline (XOR address mapping and aggressive Rambus scheduling). The more aggressive system sees increased channel utilization: A more efficient system reduces execution time, resulting in an increased utilization as the same amount of data is moved across the channels in less time. Region prefetching, shown in the third column, increases the channel utilization substantially. The benchmarks with spatial locality (all in `C-local` and some in `C-bw`) show a two-fold increase in utilization, which is partially due to spurious prefetches and partially due to reduced execution time. The `C-cpu` benchmarks show a relatively large but absolutely small increase in utilization, from 1 percent to 12.6 percent. They do not incur sufficient misses for even the aggressive prefetching to keep the channels busy. Finally, the `C-nolocal` benchmarks bear the brunt of the useless prefetches. Since their prefetch accuracies are so low, 97 percent (on average) of the prefetches result in extra traffic, causing a large increase from 18.2 percent utilization to 66.0 percent.

Since the prefetches are only issued on otherwise idle cycles, the performance impact from these increases in channel contention are uniformly outweighed by the benefits of the prefetching. However, these increases could have negative implications in a multiprocessor system, where bandwidth is a more critical resource. The other major concern for these utilization increases is a corresponding increase in power consumption. If power consumption or other considerations require limiting useless bandwidth consumption, counters could measure prefetch accuracy online and throttle the prefetch engine if the accuracy is sufficiently low. Similar adaptive prefetch schemes have shown great effectiveness in previous work [8]. In related work, we have shown that simple hardware heuristics can greatly reduce superfluous prefetches and, thus, bandwidth consumed, without losing much of the benefits gained by region prefetching [19].

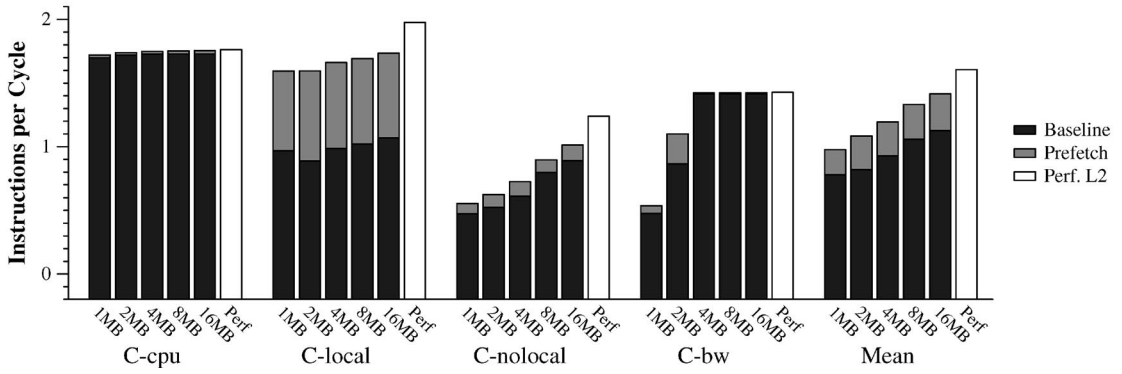

Fig. 7. Effect of cache size on region prefetching.

5.6 Implications of Multimegabyte Caches

Thus far we have simulated only 1MB level-two caches. On-chip L2 cache sizes will doubtless grow in subsequent generations. We simulated the tuned baseline organization and the best region prefetching policy with caches of two, four, eight, and 16 megabytes. We display the results graphically for each benchmark category in Fig. 7. For the baseline system, the resulting speedups over a 1MB cache were 5 percent, 19 percent, 36 percent, and 45 percent, respectively. The performance improvement from prefetching remains remarkably stable across these cache sizes: 25-28 percent at all sizes except 2MB, where an anomalous interaction with our address mapping (Section 3.2) on several benchmarks degrades their base performance, enabling a 32 percent improvement from prefetching.

The effect of larger caches varied substantially across the benchmarks, breaking roughly into three categories:

1. The benchmarks in `C-cpu` incur few L2 misses at 1MB and thus benefit neither from prefetching nor from larger caches.
2. Most of the benchmarks for which we see large improvements from prefetching benefit significantly less from increases in cache sizes. The 1MB cache is sufficiently large to capture the largest temporal working sets and the prefetching exploits the remaining spatial locality. For all of the `C-local` benchmarks except *facerec*, the performance of the 1MB cache with prefetching is higher than the 16MB cache without prefetching.
3. Eight of the SPEC applications have working sets larger than 1MB, but do not have sufficient spatial locality or available bandwidth for scheduled region prefetching to perform well. Some of these working sets reside at 2MB (*bzip2*, *galgel*), between 2MB and 4MB (*ammp*, *art*, *vpr*), and near 8MB (*ammp*, *facerec*, *mcf*). These eight benchmarks are the only ones for which increasing the cache size beyond 1MB provides greater improvement than scheduled region prefetching.

5.7 Sensitivity to DRAM Latencies

We performed further experiments to measure the effects of varied DRAM latencies on the effectiveness of region prefetching. In addition to the 40-800 DRDRAM part (40ns latency at 800 MHz data transfer rate) that we simulated throughout this paper, we also measured our prefetch performance on published 50-800 part parameters and a hypothetical 34-800 part (obtained using published 45-600 cycle latencies without adjusting the cycle time). If we were to hold the DRAM latencies constant, these latencies would correspond to processors running at 1.3 GHz and 2.0 GHz, respectively.

We show the effect of varied latencies upon the prefetching gains in Table 12, which contains the mean IPC across all benchmarks for our tuned baseline and the best 4-channel region prefetching policy. We find that, although the prefetching gains improve slightly as the DRAM latencies grow longer, they are relatively insensitive to the processor clock/DRAM speed ratio. For the slower 1.3 GHz clock (which is 18 percent slower than the base 1.6 GHz clock), the mean gain from prefetching, across all benchmarks, was reduced from 25 percent to 24 percent. A 25 percent longer DRAM latency translated into a half-percent improvement in the gain from prefetching.

Larger on-chip caches are a certainty over the next few generations and lower memory latencies are possible. Although this combination would help to reduce the impact of L2 stalls, we have shown that the scheduled region prefetching is relatively insensitive to both larger caches and faster DRAM parts. Region prefetching will thus likely reduce L2 stall time dramatically in future systems, without degrading the performance of applications that have poor spatial locality.

5.8 Interaction with Software Prefetching

It is possible that software prefetching could exploit much of the same regularity that the region prefetching engine exploits. To study the interaction of region prefetching with compiler-driven software prefetching, we modified our simulator to use the software prefetch instructions inserted

TABLE 12
Effect of DRAM Latencies on Prefetching

	800_34	800_40	800_50
Baseline	0.809	0.779	0.752
Prefetch	1.004	0.977	0.947
Speedup	1.241	1.254	1.259

TABLE 13
Interaction of Hardware and Software Prefetching

	base	HW	SW	SWovh	SW+HW
C-cpu	1.697	1.720	1.698	1.695	1.721
C-local	0.969	1.595	1.043	0.952	1.580
C-nolocal	0.475	0.555	0.474	0.471	0.552
C-bw	0.477	0.537	0.471	0.467	0.536
Mean	0.779	0.977	0.795	0.770	0.972

by the Compaq compiler. (In prior sections, we have ignored software prefetches by having the simulator discard these instructions as they are fetched.)

We show a summary of software prefetching efficacy in Table 13. The five columns of results, from left to right, contain mean IPCs for the 4-channel optimized baseline, the best region prefetching scheme, the harmonic mean IPC for software prefetching running on the baseline, the baseline plus the overhead of software prefetches (incurring the issue bandwidth but not the prefetches themselves), and the region prefetching and software prefetching together.

We found that, on the base system, only a few benchmarks benefit significantly from software prefetching: The performance of *mgrid*, *swim*, and *wupwise* improved by 30 percent, 43 percent, and 12 percent, respectively. The overhead of issuing prefetches decreased performance on *galgel* by 13 percent. For the other benchmarks, performance with software prefetching was within 2 percent of running without. We confirmed this behavior by running two versions of each executable natively on a 667-MHz Alpha 21264 system: one unmodified and one with all prefetches replaced by NOPs. Results were similar: *mgrid*, *swim*, and *wupwise* improved (by 36 percent, 23 percent, and 14 percent, respectively), and *galgel* declined slightly (by 1 percent). The native runs also showed small benefits on *apsi* (5 percent) and *lucas* (5 percent), but, otherwise, the performance gain was within 3 percent across the two versions.

As seen in Table 13, the benefits of software prefetching are largely subsumed by region prefetching. With region prefetching enabled, none of the benchmarks improved noticeably with the addition of software prefetching (2 percent at most). *Galgel* again dropped by 10 percent. Interestingly, software prefetching decreased performance on *mgrid* and *swim* by 6 percent and 3 percent, respectively, in spite of its benefits when no region prefetching was present. Not only does region prefetching subsume the benefits of software prefetching on these benchmarks, it makes them run so efficiently that the overhead of issuing software prefetch instructions has a detrimental impact.

These results represent only one specific compiler and prefetching algorithm, of course; in the long run, we anticipate synergy in being able to schedule compiler-generated prefetches along with hardware-generated region prefetches, or other prefetch types, on the memory channel.

6 Related Work

We first evaluated the region prefetching scheme in previously published work [18]. This paper extends that work with a more complete analysis of the data, the addition of aggressive scheduling of demand fetches on the DRDRAM channels, a comparison of scheduled region prefetching against software prefetching, and an evaluation of the performance impact of varied numbers of active regions in the prefetch queue. We were surprised to learn that two regions outperformed the larger numbers (16) simulated in previous work.

The ability of large cache blocks to decrease miss ratios and the associated bandwidth trade-off that causes performance to peak at much smaller block sizes are well-known [23], [25]. Using smaller blocks but prefetching additional sequential or neighboring blocks on a miss is a common approach to circumventing this trade-off. Gindele first proposed tagged prefetch, in which the next block is prefetched when the fetched block is accessed [11]. Smith analyzed Gindele's scheme and other simple sequential prefetching schemes [26]. Dahlgren et al. generalized the sequential prefetching scheme to prefetch N blocks upon a miss, in the context of fine-grained, cache-coherent shared-memory multiprocessors. They extended their scheme with an adaptive mechanism, which reduced N (to as low as zero) when prefetches were ineffective and increased N so long as the prefetch accuracy stayed above a predefined threshold. Without the adaptive scheme, they saw that the number of cache blocks at which performance was highest was prefetching one block. Their adaptive scheme played a role similar to the set of techniques that we evaluated to reduce the overhead of superfluous prefetches.

Several techniques seek to reduce both memory traffic and cache pollution by fetching multiple blocks only when the extra blocks are expected to be useful. This expectation may be based on profile information [10], [30], hardware detection of strided accesses [22] or spatial locality [14], [17], [30], compiler annotation of load instructions [28], or software binding of contiguous related lines into a prefetch group [34]. In the software binding scheme, the software (i.e., the compiler or programmer) marks groups of related, contiguous memory lines using additional memory state bits. A miss to any line in such a group causes the memory controller to prefetch all succeeding lines within the group. We note that our scheme requires neither software support nor additional memory bits; as a result, it is much less discriminating in what to prefetch.

Optimal offline algorithms for fetching a set of noncontiguous words [29] or a variable-sized aligned block [30] on each miss provide bounds on these techniques. Pollution may also be reduced by prefetching into separate buffers [15], [28].

Our work limits prefetching by prioritizing memory channel usage, reducing bandwidth contention directly and pollution indirectly. Driscoll et al. [9], [10] similarly cancel ongoing prefetches on a demand miss. However, their rationale appears to be that the miss indicates that the current prefetch candidates are useless and they discard them rather than resuming prefetching after the miss is handled. Przybylski [23] analyzed cancellations of ongoing demand fetches (after the critical word had returned) on a subsequent miss, but found that performance was reduced, most likely because the original block was not written into the cache. Our scheduling technique is independent of the scheme used to generate prefetch addresses; determining the combined benefit of scheduling and more conservative prefetching techniques [10], [14], [17], [22], [30] is an area of future research. Our results also show that, in a large secondary cache, controlling the replacement priority of prefetched data appears sufficient to limit the displacement of useful referenced data.

Prefetch reordering to exploit DRAM row buffers was previously explored by Zhang and McKee [32]. They interleave the demand miss stream and several strided prefetch streams—generated using a reference prediction table [2]—dynamically in the memory controller. They assume a nonintegrated memory controller and a single Direct Rambus channel, leading them to use a relatively conservative prefetch scheme. We show that near-future systems with large caches, integrated memory controllers, and multiple Rambus channels can profitably prefetch more aggressively. Zhang and McKee saw little benefit from prioritizing demand misses above prefetches. With our more aggressive prefetching, we found that allowing demand misses to bypass prefetches is critical for avoiding bandwidth contention.

Several researchers have proposed memory controllers for vector or vector-like systems that interleave access streams to better exploit row-buffer locality and hide precharge and activation latencies [5], [13], [20], [21], [24]. Vector/streaming memory accesses are typically bandwidth bound, may have little spatial locality, and expose numerous nonspeculative accesses to schedule, making aggressive reordering both possible and beneficial. In contrast, in a general-purpose environment, latency may be more critical than bandwidth, cache-block accesses provide inherent spatial locality, and there are fewer simultaneous nonspeculative accesses to schedule. A flexible system intended to run both types of codes well should incorporate mechanisms for both region prefetching and efficient streaming.

7 Conclusions

Even the integration of megabyte caches and fast Rambus channels on the processor die is insufficient to compensate for the penalties associated with going off-chip for data. Across the 26 SPEC2000 benchmarks, L2 misses account for 60 percent of overall performance on a system with four Direct Rambus channels. More aggressive processing cores will only serve to widen that gap.

We have measured several techniques for reducing the effect of L2 miss latencies. Tuning DRAM address mappings to reduce row-buffer misses and bank conflicts—considering both read and writeback accesses—provides significant benefits. Aggressive scheduling of demand misses provides incremental gains. These optimizations served as our baseline. Large block sizes can further improve performance on benchmarks with spatial locality, but fail to provide an overall performance gain unless much wider channels are used to provide higher DRAM bandwidth.

We proposed and evaluated a prefetch architecture, integrated with the on-chip L2 cache and memory controllers, that aggressively prefetches large regions of data on demand misses. By scheduling these prefetches only during idle cycles on the Rambus channel, inserting them into the cache with low replacement priority, and prioritizing them to take advantage of the DRAM organization, we improve performance significantly on 10 of the 26 SPEC benchmarks without adversely affecting the others.

To address the problem for the other benchmarks that stall frequently for off-chip accesses, we must discover other methods for driving the prefetch queue besides region prefetching, effectively making the prefetch controller programmable on a per-application basis. Other future work that will broaden the classes of applications that benefit from integrated prefetch controllers includes evaluating the effects of more sophisticated channel organizations: For example, treating single, wide channels as multiple narrow, complex interleaved channels, and/or dynamically switching between the two organizations.

References

[1] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock Rate versus ipc: The End of the Road for Conventional Microarchitectures," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 248-259, June 2000.

[2] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data access Penalty," *Proc. Supercomputing '91,* pp. 176-186, Nov. 1991.

[3] D. Burger and T.M. Austin, "The Simplescalar Tool Set Version 2.0," Technical Report 1342, Computer Sciences Dept., Univ. of Wisconsin, Madison, June 1997.

[4] D. Burger, J.R. Goodman, and A. Kägi, "Memory Bandwidth Limitations of Future Microprocessors," *Proc. 23rd Ann. Int'l Symp. Computer Architecture,* pp. 78-89, May 1996.

[5] J. Corbal, R. Espasa, and M. Valero, "Command Vector Memory Systems: High Performance at Low Cost," *Proc. 1998 Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 68-77, Oct. 1998.

[6] R. Crisp, "Direct Rambus Technology: The New Main Memory Standard," *IEEEM?* vol. 17, no. 6, pp. 18-27, Dec. 1997.

[7] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A Performance Comparison of Contemporary DRAM Architectures," *Proc. 26th Ann. Int'l Symp. Computer Architecture,* pp. 222-233, May 1999.

[8] F. Dahlgren, M. Dubois, and P. Stenström, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 7, pp. 733-746, July 1995.

[9] G.C. Driscoll, J.J. Losq, T.R. Puzak, G.S. Rao, H.E. Sachar, and R.D. Villani, "Cache Miss Directory—A Means of Prefetching Cache Missed Lines," *IBM Technical Disclosure Bulletin,* vol. 25, p. 1286, Aug. 1982, http://www.patents.ibm.com/tdbs/tdb?o=82A%2061161.

[10] G.C. Driscoll, T.R. Puzak, H.E. Sachar, and R.D. Villani, "Staging Length Table—A Means of Minimizing Cache Memory Misses Using Variable Length Cache Lines," *IBM Technical Disclosure Bulletin,* vol. 25, p. 1285, Aug. 1982, http://www.patents.ibm.com/tdbs/tdb?o=82A%2061160.
[11] J.D. Gindele, "Buffer Block Prefetching Method," *IBM Technical Disclosure Bulletin,* vol. 20, no. 2, pp. 696-697, July 1977.
[12] L. Gwennap, "Alpha 21364 to Ease Memory Bottleneck," *Microprocessor Review,* pp. 12-15, 26 Oct. 1998.
[13] S.I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor, and W.A. Wulf, "Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory," *Proc. Fifth Int'l Symp. High-Performance Computer Architecture,* pp. 80-89, Jan. 1999.
[14] T.L. Johnson and W.W. Hwu, "Run-Time Adaptive Cache Hierarchy Management via Reference Analysis," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 315-326, June 1997.
[15] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 364-373, 1990.
[16] D. Kroft, "Lockup-Free Instruction Fetch/Prefetch Cache Organization," *Proc. Eighth Int'l Symp. Computer Architecture,* pp. 81-87, May 1981.
[17] S. Kumar and C. Wilkerson, "Exploiting Spatial Locality in Data Caches Using Spatial Footprints," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* July 1998.
[18] W.-F. Lin, S.K. Reinhardt, and D. Burger, "Reducing DRAM Latencies with a Highly Integrated Memory Hierarchy Design," *Proc. Seventh Symp. High-Performance Computer Architecture,* pp. 301-312, Jan. 2001.
[19] W.-F. Lin, S.K. Reinhardt, D. Burger, and T.R. Puzak, "Filtering Superfluous Prefetches Using Density Vectors," *Proc. Int'l Conf. Computer Design,* Sept. 2001.
[20] B.K. Mathew, S.A. McKee, J.B. Carter, and A. Davis, "Design of a Parallel Vector Access Unit for SDRAM Memory Systems," *Proc. Sixth Int'l Symp. High-Performance Computer Architecture,* Jan. 2000.
[21] S.A. McKee nd W.A. Wulf, "Access Ordering and Memory-Conscious Cache Utilization," *Proc. First Int'l Symp. High-Performance Computer Architecture,* pp. 253-262, Jan. 1995.
[22] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 24-33, Apr. 1994.
[23] S. Przybylski, "The Performance Impact of Block Sizes and Fetch Strategies," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 160-169, May 1990.
[24] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens, "Memory Access Scheduling," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 128-138, June 2000.
[25] A.J. Smith, "Line (Block) Size Choice for CPU Cache Memories," *IEEE Trans. Computers,* vol. 36, no. 9, pp. 1063-1075, Sept. 1987.
[26] A.J. Smith, "Cache Memories," *Computing Surveys,* vol. 14, no. 3, pp. 473-530, Sept. 1982.
[27] G.S. Sohi, "Instruction Issue Logic for High-Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Trans. Computers,* vol. 39, no. 3, pp. 349-359, Mar. 1990.
[28] O. Temam and Y. Jegou, "Using Virtual Lines to Enhance Locality Exploitation," *Proc. 1994 Int'l Conf. Supercomputing,* pp. 344-353, July 1994.
[29] O. Temam, "Investigating Optimal Local Memory Performance," *Proc. Eighth Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 218-227, Oct. 1998.
[30] P. Van Vleet, E. Anderson, L. Brown, J.-L. Baer, and A. Karlin, "Pursuing the Performance Potential of Dynamic Cache Line Sizes," *Proc. 1999 Int'l Conf. Computer Design,* pp. 528-537, Oct. 1999.
[31] W.A. Wong and J.-L. Baer, "Dram Caching," Technical Report 97-03-04, Dept. of Computer Science and Eng., Univ. of Washington, 1997.
[32] C. Zhang and S.A. McKee, "Hardware-Only Stream Prefetching and Dynamic Access Ordering," *Proc. 14th Int'l Conf. Supercomputing,* May 2000.
[33] Z. Zhang, Z. Zhu, and X. Zhang, "A Permutation-Based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality," *Proc. 33rd Int'l Symp. Microarchitecture,* pp. 32-41, Dec. 2000.
[34] Z. Zhang and J. Torrellas, "Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching," *Proc. 22nd Ann. Int'l Symp. Computer Architecture,* pp. 188-199, June 1995.
[35] J.H. Zurawski, J.E. Murray, and P.J. Lemmon, "The Design and Verification of the Alphastation 600 5-Series Workstation," *Digital Technical J.,* vol. 7, no. 1, Aug. 1995.

Wei-Fen Lin is presently a PhD candidate in computer science and engineering in the Electrical Engineering and Computer Science Department at the University of Michigan in Ann Arbor. She received the BSE degree in electrical engineering from National Taiwan University, Taiwan, in 1996 and the MSE degree in system science engineering from University of Michigan in 1998. Her research interests include microarchitecture, caches, and memory systems. He is a student member of the IEEE and the IEEE Computer Society.

Steven K. Reinhardt received the BS degree in electrical engineering from Case Western Reserve University in 1987, the MS degree in electrical engineering from Stanford University in 1988, and the PhD degree in computer science from the University of Wisconsin-Madison in 1996. He is currently an assistant professor of electrical engineering and computer science at the University of Michigan in Ann Arbor. His primary research interest is in computer system architecture, focusing on uniprocessor and multiprocessor memory systems, multithreaded systems, and system simulation techniques. He is the recipient of a 2001 Sloan Research Fellowship and a 1998 US National Science Foundation CAREER award. Dr. Reinhardt is a member of the ACM, the IEEE, and the IEEE Computer Society.

Doug Burger received the BS degree in computer sciences from Yale University in 1991, the MS degree in computer sciences from the University of Wisconsin-Madison in 1993, and the PhD Degree in computer sciences from the University of Wisconsin-Madison in 1998. He is currently an assistant professor of computer sciences at the University of Texas at Austin. His primary research interests are in technology scaling, computer architecture, and low-level software for future high-performance systems. He is the recipient of a 1999 US National Science Foundation CAREER award. Dr. Burger is a member of the ACM, the IEEE, and the IEEE Computer Society.