Appears in the 7th International Symposium on High-Performance Computer Architecture, January 2001.

Reducing DRAM Latencies with an Integrated Memory Hierarchy Design

Wei-fen Lin and Steven K. Reinhardt
Electrical Engineering and Computer Science Dept.
University of Michigan
{wflin,stever}@eecs.umich.edu

Doug Burger
Department of Computer Sciences
University of Texas at Austin
dburger@cs.utexas.edu

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 of its time stalling for L2 misses. Large cache blocks can improve performance, but only when coupled with wide memory channels. DRAM address mappings also affect performance significantly.

We evaluate an aggressive 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 43% speedup across 10 of the 26 SPEC2000 benchmarks, without degrading performance on the others. With eight Rambus channels, these ten benchmarks improve to within 10% of the performance of a perfect L2 cache.

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, non-blocking 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 Figure 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 straightforward 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 Figure 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 57% of its time servicing L2 misses, 12% of its time servicing L1 misses, and only 31% 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, mcf has the highest L2 stall fraction (80%) because it suffers 23 million L2 misses during the 200-million-instruction sample we ran, saturating the memory controller request bandwidth. At the other extreme, a 200M-instruction sample of facerec spends 60% of its time waiting for only 1.2 million DRAM accesses.

These varying 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,

This work is supported in part by the National Science Foundation under Grant No. CCR-9734026, a gift from Intel, IBM University Partnership Program Awards, and an equipment grant from Compaq.

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

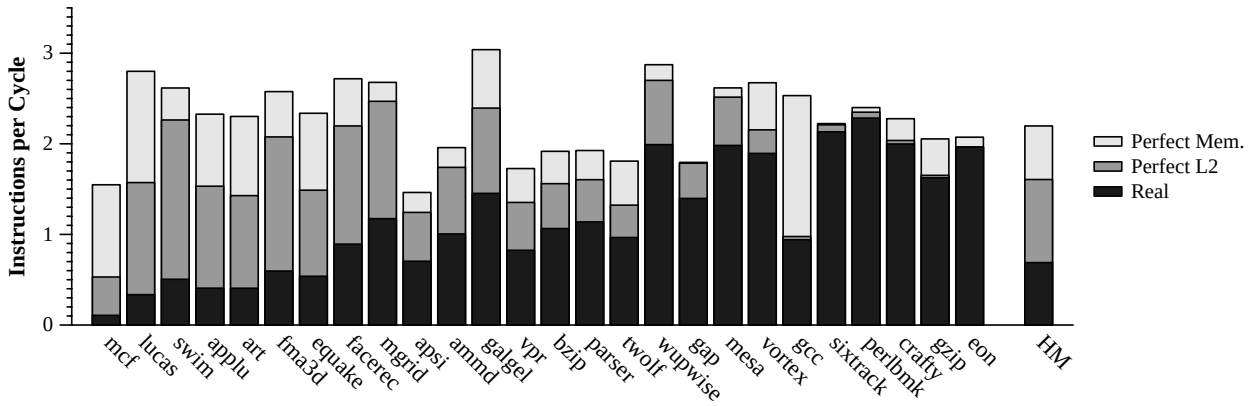

Figure 1. Processor performance for SPEC2000

but will decrease the performance of a bandwidth-bound application by consuming scarce bandwidth and increasing queueing delays [4]. Conversely, reordering memory references to increase DRAM bandwidth [5,11,15,16,19] may not help latency-bound applications, which rarely 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 (10% to 119%) 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, however, reducing performance for only one benchmark. Three mechanisms minimize the potential negative aspects of aggressive prefetching: prefetching data only on idle Rambus 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 study the impact of block size, memory bandwidth, and address mapping on performance. In Section 4, we describe and evaluate our scheduled region prefetching engine. We discuss related work in Section 5 and draw our conclusions in Section 6.

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 multi-megabyte caches will follow. These larger caches, with more numerous 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) make 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 Compaq 21364—are following this route.[1] 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 busses allows the memory controller to send commands to independent banks concurrently, facilitating greater overlap of operations that 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 one 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 a 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.[2]

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, than 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 Figure 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 effect of varying block sizes, channel widths, and DRAM bank mappings on the memory system and overall performance. Our results motivate our prefetching strategy, described in Section 4, and provide an optimized baseline for comparison.

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, and Direct Rambus memory channels and devices in detail [6].

Although the SimpleScalar microarchitecture is based on the Register Update Unit [22], we chose the rest of the parameters to match the Compaq Alpha 21364 [10] as closely as possible. These parameters include an aggres-

1. 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.

2. 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.

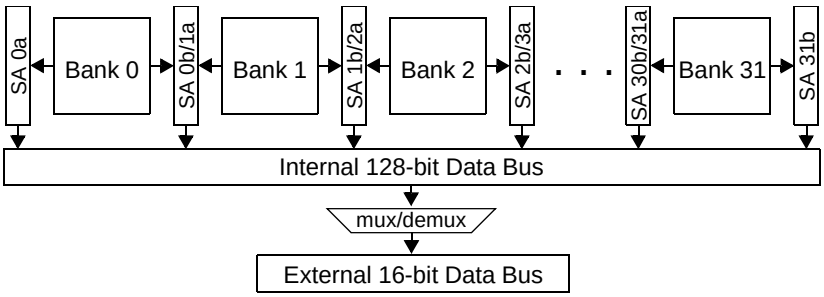

Figure 2. Rambus shared sense-amp organization.

sive 1.6GHz clock,[1] a 64-entry RUU (reorder buffer/issue window,) a 64-entry load/store queue, a four-wide issue core, 64KB 2-way associative first-level instruction and data caches, ALUs similar to the 21364 in quantities and delays, a 16K entry hybrid local/global branch predictor, a 2-way set associative, 256-entry BTB, a 128-bit L1/L2 on-chip cache bus, 8 MSHRs per data cache, a 1MB, 4-way set associative, on-chip level-two data cache accessible in 12 cycles, and a 256MB DRDRAM system transmitting data packets at 800MHz. Our systems use 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 evaluated our simulated systems using the 26 SPEC CPU2000 benchmarks compiled with recent Compaq compilers (C V5.9-008 and Fortran V5.3-915).[2] 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% or less on each benchmark.

3.2. 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 increase 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 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 that gives the minimum miss rate.

In Table 1, we show the pollution and performance points for our benchmarks assuming four DRDRAM channels, providing 6.4GB/s peak bandwidth. The pollution points are at block sizes much larger than typical L2 block sizes (e.g., 64 bytes in the 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). Taking the harmonic mean of the IPCs at each block size, we find that performance is highest at 128-byte blocks, with a negligible difference

Table 1: Pollution and performance points

BM	amm	app	aps	art	bzi	cra	eon	equ
Poll.	8K	8K	2K	4K	128	128	4K	8K
Perf.	64	2K	256	64	64	128	2K	2K
fac	**fma**	**gal**	**cap**	**gcc**	**gzi**	**luc**	**mcf**	**mes**
8K	8K	1K	8K	512	8K	1K	1K	512
8K	256	256	2K	256	1K	128	64	512
mgr	**par**	**per**	**six**	**swi**	**two**	**vor**	**vpr**	**wup**
8K	2K	1K	8K	8K	1K	128	64	8K
512	512	256	2K	1K	128	128	64	512

1. We selected this clock rate as it is both near the maximum clock rate announced for near-future products (1.5 GHz Pentium 4), and because it is exactly twice the effective frequency of the DRDRAM channels.

2. 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.

between 128- and 256-byte blocks. For eight of the benchmarks with high spatial locality, however, the performance point is at block sizes even larger than 256 bytes.

The miss rates at the pollution points (not shown due to space considerations) are significantly lower than at the performance points: more than a factor of two for half of the benchmarks, and more than ten-fold 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: a factor of ten for ammp, and two to three times for four others, but less than 50% for the rest.

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. We present a prefetching scheme that accomplishes this goal in Section 4.

3.3. Channel width

Emerging systems contain a varied number of Rambus channels. Intel's Willamette processor will contain between one and two RDRAM channels, depending on whether the part is used in medium- or high-end machines. The Alpha 21364, however, will contain up to a maximum of eight RDRAM 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 2, we show the effect of the number of physical channels on performance at various block sizes. The numbers shown in the table are the harmonic mean of IPC for all of the SPEC benchmarks at a given block size and channel width.

For a four-channel system, the performance point resides at 256-byte blocks. At eight channels, the best block size is 512 bytes. 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. This restriction favored larger blocks slightly, causing these results to differ from the performance point results described in Section 3.2.

As the channels grow wider, the performance point shifts to larger block sizes until it is eventually (for a sufficiently wide logical channel) equivalent to the pollution point. Past that point, larger blocks will pollute the cache and degrade performance.

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. Achieving this bandwidth is prohibitively expensive; our prefetching architecture provides a preferable solution, exploiting spatial locality while avoiding bandwidth contention on a smaller number of channels.

Table 2: Channel width vs. performance points

	Block size				
Channels	**64**	**128**	**256**	**512**	**1024**
1	**0.327**	0.275	0.219	0.159	0.099
2	**0.435**	0.422	0.369	0.286	0.186
4	0.502	0.529	**0.542**	0.468	0.329
8	0.478	0.545	0.638	**0.651**	0.525
16	0.456	0.555	0.665	**0.742**	0.710
32	0.424	0.521	0.656	0.730	**0.755**

3.4. 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% on average, with several benchmarks seeing speedups above 40%.

In Figure 3a, we depict the base address mapping used to this point. 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.

Although this address mapping provides a reasonable row-buffer hit rate on read accesses (51% on average), the

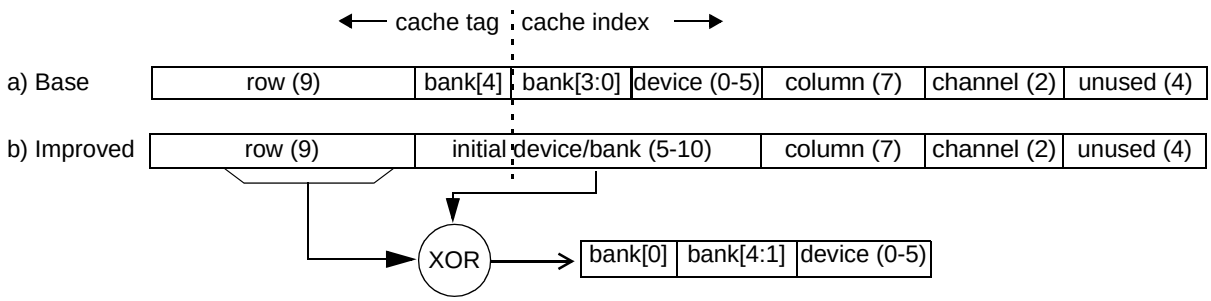

Figure 3. Mapping physical addresses to Rambus coordinates.

hit rate on writebacks is only 28%. 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 ($\log_2$(1MB/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 Figure 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% conflict probability.

One previously described solution is to exchange some of the row and column index bits in the mapping [28,26]. 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 discontiguous addresses in a single row, spatial locality is reduced.

Our solution, shown in Figure 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. 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 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% for read accesses and 55% for writebacks. This final address mapping, which will be used for the remainder of our studies, improves performance by 16% on average, and helps some benchmarks significantly (63% for applu and over 40% for swim, fma3d, and facerec).

4. Improving Rambus 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 to improve 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 [23]. 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.

We depict our prefetch controller in Figure 4. In our simulated implementation, region prefetches are *scheduled* to be issued only when the Rambus channels are otherwise idle. The prefetch queue maintains a list of *n* region entries not in the L2 cache, represented as bitmaps. The region entry spans multiple blocks over a region, with a bit vector representing each block in the region. A bit in the vector is set if a block is being prefetched or is in the cache. The number of bits is equal to the prefetch region size divided by the L2 block size.

When a demand miss occurs that does not match an entry in the prefetch queue, the oldest entry is overwritten with the new demand miss. The prefetch prioritizer uses the bank state and the region ages to determine which prefetch to issue next. The access prioritizer selects a prefetch when no demand misses or writebacks are pending. The prefetches thus add little additional channel contention, and only when a demand miss arrives while a prefetch is in progress. For the next two subsections, we assume that (1) prefetch regions are processed in FIFO order, (2) that a region's blocks are fetched in linear order starting with the block after the demand miss (and wrapped around), and (3) that a region is only retired when it is either overwritten by a new miss or all of its blocks have been processed.

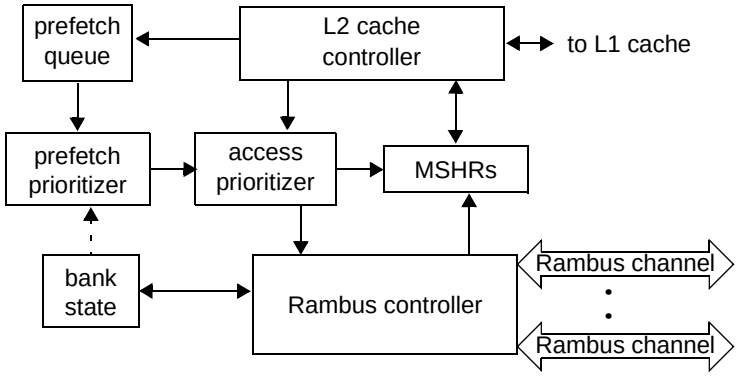

Figure 4. Prefetching memory controller

4.1. 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 the prefetch a lower replacement priority than 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 a prefetches are loaded with LRU priority, they can displace at most one quarter of the referenced data in the cache.

For this section, we divide the SPEC2000 benchmarks into two categories: those with prefetch accuracies of greater than 20% (applu, art, eon, equake, facerec, fma3d, gap, gcc, gzip, mgrid, parser, sixtrack, swim, and wupwise) and those with accuracies below 20% (ammp, apsi, bzip2, crafty, galgel, lucas, mcf, mesa, perlbmk, twolf, vortex, and vpr). In Table 3, we depict the arithmetic mean of the prefetch accuracies for the two classes of benchmarks, shown as the region prefetches are loaded into differing points on the replacement priority chain. We also show the speedups of the harmonic mean of IPC values over MRU prefetch insertion. In these experiments, we simulated 4KB prefetch regions, 64-byte blocks, and four DRDRAM channels.

Table 3: LRU chain prefetch priority insertion

		Priority			
Accuracy class	**Quantity**	**MRU**	**SMRU**	**SLRU**	**LRU**
High	Accuracy	63%	63%	62%	56%
	IPC	1.00	1.01	1.02	1.02
Low	Accuracy	4%	4%	4%	3%
	IPC	1.00	1.31	1.45	1.51

For the high-accuracy benchmarks, the prefetch accuracy decreases slightly as the prefetches are given lower priority in the set. With lower priority, a prefetch is more likely to be evicted before it is referenced. However, since many of the high-accuracy benchmarks quickly reference their prefetches, the impact on accuracy is minor. Performance drops by 12% and 17% on equake and facerec, respectively, as placement goes from MRU to LRU. These losses are counterbalanced by similar gains in other benchmarks (gcc, parser, art, and swim), where pollution is an issue despite relatively high accuracy.

For the low accuracy benchmarks, the prefetch accuracy drops negligibly from MRU (3.5%) to LRU (3.3%). The impact on IPC, however, is dramatic. Placing the prefetches in the cache with high priority causes significant pollution, lowering performance over that with MRU by 33%.

While replacement prioritization does not help high-accuracy benchmarks 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.

4.2. Prefetch scheduling

Unfortunately, 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.

With 4KB region prefetching, a substantial number of misses are avoided, as shown in column two of Table 4. The L2 miss rate is reduced from 36.4% in the base system (which includes the XOR bank mapping) to just 10.9%.

Table 4: Comparison of prefetch schemes

SPEC2000 average	Base (w/XOR)	FIFO prefetch	Sched. FIFO	Sched. LIFO
L2 miss rate	36.4%	10.9%	18.3%	17.0%
L2 miss latency (cycles)	134	980	140	141
Normalized IPC	1.00	0.33	1.12	1.16

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 134 cycles to 980 cycles.

This large increase in channel contention can be avoided by scheduling prefetch accesses only when the Rambus channel would otherwise be idle. When the Rambus controller is ready for another 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, depicted in Figure 4. Our baseline 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.

With this scheduling policy, the prefetching continues to achieve a significant reduction in misses, but with only a small increase in the mean L2 miss latency. While the unscheduled prefetching achieves a lower miss rate since every region prefetch issues, the miss penalty increase is far too high. The prefetch scheduling greatly improves the ten benchmarks for which region prefetching is most effective (applu, equake, facerec, fma3d, gap, mesa, mgrid, parser, swim, and wupwise), which show a mean 37% improvement in IPC. This prefetch scheme is also unintrusive; five of the other benchmarks (ammp, galgel, gcc, twolf, and vpr) show small performance drops (an average of 2% in IPC). Across the entire SPEC suite, performance shows a mean 12% increase.

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 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 on a demand miss 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%, and reduces the total number of row-buffer misses by 9%.

These optimizations, labeled “scheduled LIFO” in column four of Table 4, help all applications, reducing the average miss rate further to 17.0%, with only a one-cycle increase in miss latency. The mean performance improvement increases to 16%. With this scheme, only one benchmark (vpr) showed a performance drop (of 1.6%) due to prefetching.

We also experimented with varying the region size, and found that, with LIFO scheduling, 4KB provided the best overall performance. Improvement dropped off for regions of less than 2KB, while increasing the region size beyond 4KB had a negligible impact. Clearly using a region size larger than the virtual page size (8 KB in our system) is not likely to be useful when prefetching based on physical addresses.

4.3. 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. Figure 5 provides detailed performance results for the ten benchmarks whose performance improves by 10% or more with scheduled region prefetching. The left-most bar for each benchmark is stacked, showing the IPC values for three targets: the 64-byte block, four-channel experiments with the standard bank mapping represented by the white bar, the XOR mapping improvement represented by the middle, light grey bar, and LIFO, 4KB region prefetching represented by the top, dark grey bar. The second bar in each cluster shows the performance of 8-channel runs with 256-byte blocks in light grey, and the same system with LIFO, 4KB region prefetching in dark grey. The right-most bar in each cluster shows the IPC obtained by a perfect L2 cache.

On the four-channel system, the XOR mapping provides a mean 33% speedup for these benchmarks. Adding prefetching results in an additional 43% speedup. Note that for eight of the ten benchmarks, the 4-channel prefetching experiments outperform the 8-channel system

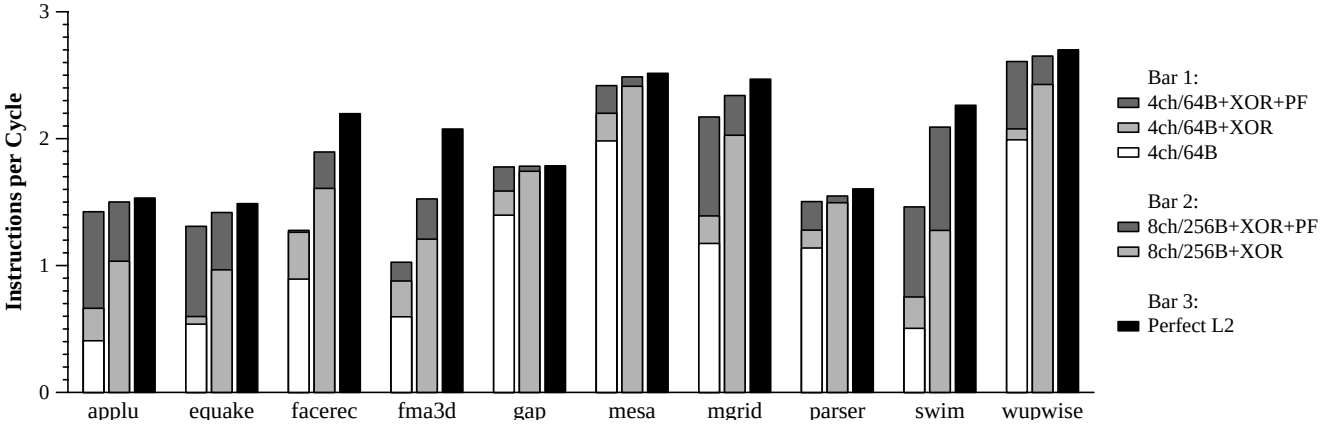

Figure 5. Overall performance of tuned scheduled region prefetching

with no prefetching. The 8-channel, 256-byte block experiments with region prefetching show the highest attainable performance, however, with a mean speedup of 118% over the 4-channel base case for the benchmarks depicted in Figure 5, and a 45% speedup across all the benchmarks. The 8-channel system with 256-byte blocks and 4KB region prefetching comes within 10% of perfect L2 cache performance for 8 of these 10 benchmarks (and thus on average for this set).

There are three effects that render scheduled region prefetching ineffective for the remaining benchmarks. The first, and most important, is low prefetch accuracies. Ammp, bzip2, crafty, mesa, twolf, vortex, and vpr all fall into that category, with prefetch accuracies of 10% or less. The second effect is a lack of available bandwidth to perform prefetching. Art achieves a prefetch accuracy of 45%, while mcf achieves 35%. However, both are bandwidth-bound, saturating the memory channel in the base case, leaving little opportunity to prefetch. Finally, the remaining benchmarks for which prefetching is ineffective typically have high accuracies and adequate available bandwidth but have too few L2 misses to matter.

4.4. Effect on Rambus channel utilization

The region prefetching scheme produces more traffic on the memory channel for all the benchmarks. We quantify this effect by measuring utilization of both the command and data channels. We derive command-channel utilization by calculating the number of cycles required to issue all of the program's memory requests in the original order but with no intervening delays (other than required inter-packet stalls) as a fraction of the total number of execution cycles. Data-channel utilization is simply the fraction of cycles during which data are transmitted.

For the base 4-channel case without prefetching, the mean command- and data-channel utilizations are 28% and 17%, respectively. Utilization on the command channel is always higher than on the data channel due to row-buffer precharge and activate commands, which count as busy time on the command channel but result in idle time on the data channel. Our memory controller pipelines requests, but does not reorder or interleave commands from multiple requests; a more aggressive design that performed this reordering would reduce this disparity.

With scheduled region prefetching, command- and data-channel utilizations are 54% and 42%, respectively—increases of 1.9 and 2.5 times over the non-prefetching case. The disparity between command- and data-channel utilizations is reduced because our bank-aware prefetch scheduling increases the fraction of accesses that do not require precharge or row-activation commands.

The increased utilizations are due partly to the increased number of fetched blocks and partly to decreased execution time. For many benchmarks, one or the other of these reasons dominates, depending on that benchmark's prefetch accuracy. At one extreme, swim's command-channel utilization increases from 58% to 96% with prefetching, thanks to a 99% prefetch accuracy giving a 49% execution-time reduction. On the other hand, twolf's command-channel utilization increases from 22% to 90% with only a 2% performance improvement due to its 7% prefetch accuracy. However, not all benchmarks consume bandwidth this heavily; half have command-channel utilization under 60% and data-channel utilization under 40%, including several that benefit significantly from prefetching (gap, mgrid, parser, and wupwise).

Even when prefetch accuracy is low, channel scheduling minimizes the adverse impact of prefetching: only one benchmark sees any performance degradation. However, if power consumption or other considerations require limiting this useless bandwidth consumption, counters could measure prefetch accuracy on-line and throttle the prefetch engine if the accuracy is sufficiently low.

4.5. Implications of multi-megabyte caches

Thus far we have simulated only 1MB level-two caches. On-chip L2 cache sizes will doubtless grow in subsequent generations. We simulated our baseline XOR-mapping organization and our best region prefetching policy with caches of two, four, eight, and sixteen megabytes. For the baseline system, the resulting speedups over a 1MB cache were 6%, 19%, 38%, and 47%, respectively. The performance improvement from prefetching remains stable across these cache sizes, growing from 16% at the 1MB cache to 20% at the 2MB cache, and remaining between 19% and 20% for all sizes up to 16MB. The effect of larger caches varied substantially across the benchmarks, breaking roughly into three categories:

1. Several benchmarks (perlbmk, eon, gap, gzip, vortex, and twolf) 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 applu, equake, fma3d, mesa, mgrid, parser, swim, and wupwise, 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 for the scheduled region prefetching to exploit 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 provides greater improvement than region prefetching at 1MB.

4.6. Sensitivity to DRAM latencies

We ran experiments to measure the effects of varying 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 find that the prefetching gains are relatively insensitive to the processor clock/DRAM speed ratio. For the slower 1.3 GHz clock (which is 18% slower than the base 1.6 GHz clock), the mean gain from prefetching, across all benchmarks, was reduced from 15.6% to 14.2%. Interestingly, the faster 2.0 GHz clock also caused a slight (less than 1%) drop in prefetch improvements.

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, scheduled region prefetching and DRAM bank mappings will still reduce L2 stall time dramatically in future systems, without degrading the performance of applications with poor spatial locality.

4.7. Interaction with software prefetching

To study the interaction of our region prefetching with compiler-driven software prefetching, we modified our simulator to use the software prefetch instructions inserted by the Compaq compiler. (In prior sections, we have ignored software prefetches by having the simulator discard these instructions as they are fetched.) We found that, on our base system, only a few benchmarks benefit significantly from software prefetching: performance on mgrid, swim, and wupwise improved by 23%, 39%, and 10%, respectively. The overhead of issuing prefetches decreased performance on galgel by 11%. For the other benchmarks, performance with software prefetching was within 3% 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%, 23%, and 14%, respectively), and galgel declined slightly (by 1%). The native runs also showed small benefits on apsi (5%) and lucas (5%) but otherwise performance was within 3% across the two versions.

We then enabled both software prefetching and our best scheduled region prefetching together, and found that the benefits of software prefetching are largely subsumed by region prefetching for these benchmarks. None of the benchmarks improved noticeably with software prefetching (2% at most). Galgel again dropped by 10%. Interestingly, software prefetching decreased performance on mgrid and swim by 8% and 3% respectively, in spite of its benefits on the base system. Not only does region prefetching subsume the benefits of software prefetching on these benchmarks, but it makes them run so efficiently that the overhead of issuing software prefetch instructions has a detrimental impact. Of course, these results represent only one specific compiler; in the long run, we anticipate synergy in being able to schedule compiler-generated prefetches along with hardware-generated region (or other) prefetches on the memory channel.

5. Related 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 [20,18]. Using smaller blocks but prefetching additional sequential or neighboring blocks on a miss is a common approach to circumventing this trade-off. Smith [21] analyzes some basic sequential prefetching schemes.

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 [9,25], hardware detection of strided accesses [17] or spatial locality [12,14,25], or compiler annotation of load instructions [23]. Optimal off-line algorithms for fetching a set of non-contiguous words [24] or a variable-sized aligned block [25] on each miss provide bounds on these techniques. Pollution may also be reduced by prefetching into separate buffers [13,23].

Our work limits prefetching by prioritizing memory channel usage, reducing bandwidth contention directly and pollution indirectly. Driscoll et al. [8,9] 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 [18] analyzed cancelling an ongoing demand fetch (after the critical word had returned) on a subsequent miss, but found that performance was reduced, probably 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 [9,12,14,17,25] 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 [27]. 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 non-integrated 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. They 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 to 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 [5,11,15,16,19]. Vector/streaming memory accesses are typically bandwidth bound, may have little spatial locality, and expose numerous non-speculative 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 non-speculative accesses to schedule. For these reasons, our controller issues demand misses in order, reordering only speculative prefetch requests.

6. 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 57% 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 latency. Large block sizes improve performance on benchmarks with spatial locality, but fail to provide an overall performance gain unless wider channels are used to provide higher DRAM bandwidth. Tuning DRAM address mappings to reduce row-buffer misses and bank conflicts—considering both read and writeback accesses—provides significant benefits. 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 negatively 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, in effect making the prefetch controller programmable on a per-application basis. Other future work includes reordering demand misses and writebacks as well as prefetches, throttling region prefetches when spatial locality is poor, aggressively scheduling the Rambus channels for all accesses, and evaluating the effects of complex interleaving of the multiple channels.

References

[1] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, pages 176–186, November 1991.

[3] Doug Burger and Todd M. Austin. The simplescalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, Madison, WI, June 1997.

[4] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[5] Jesus Corbal, Roger Espasa, and Mateo Valero. Command vector memory systems: High performance at low cost. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 68–77, October 1998.

[6] Richard Crisp. Direct rambus technology: The new main memory standard. *IEEE Micro*, 17(6):18–27, December 1997.

[7] Vinodh Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge. A performance comparison of contemporary dram architectures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 222–233, May 1999.

[8] 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*, 25:1286, August 1982. http://www.patents.ibm.com/tdbs/tdb?o=82A%2061161.

[9] 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*, 25:1285, August 1982. http://www.patents.ibm.com/tdbs/tdb?o=82A%2061160.

[10] Linley Gwennap. Alpha 21364 to ease memory bottleneck. *Microprocessor Report*, 12(14):12–15, October 26, 1998.

[11] S.I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor, and Wm.A. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pages 80–89, January 1999.

[12] T.L. Johnson and W.W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[13] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[14] Sanjeev Kumar and Christopher Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, July 1998.

[15] Binu K. Mathew, Sally A. McKee, John B. Carter, and Al Davis. Design of a parallel vector access unit for sdram memory systems. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, January 2000.

[16] Sally A. McKee and Wm. A. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First International Symposium on High-Performance Computer Architecture*, pages 253–262, January 1995.

[17] Subbarao Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 24–33, April 1994.

[18] Steven Przybylski. The performance impact of block sizes and fetch strategies. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 160–169, May 1990.

[19] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 128–138, June 2000.

[20] A. J. Smith. Line (block) size choice for cpu cache memories. *IEEE Transactions on Computers*, 36(9):1063–1075, September 1987.

[21] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[22] Gurindar S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[23] O. Temam and Y. Jegou. Using virtual lines to enhance locality exploitation. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 344–353, July 1994.

[24] Olivier Temam. Investigating optimal local memory performance. In *Proceedings of the Eighth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 218–227, October 1998.

[25] Peter Van Vleet, Eric Anderson, Linsay Brown, Jean-Loup Baer, and Anna Karlin. Pursuing the performance potential of dynamic cache line sizes. In *Proceedings of the 1999 International Conference on Computer Design*, pages 528–537, October 1999.

[26] Wayne A. Wong and Jean-Loup Baer. Dram caching. Technical Report 97-03-04, Department of Computer Science and Engineering, University of Washington, 1997.

[27] Chengqiang Zhang and Sally A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *Proceedings of the 14th International Conference on Supercomputing*, May 2000.

[28] John H. Zurawski, John E. Murray, and Paul J. Lemmon. The design and verification of the alphastation 600 5-series workstation. *Digital Technical Journal*, 7(1), August 1995.