

# Hardware-Only Stream Prefetching and Dynamic Access Ordering

Chengqiang Zhang and Sally A. McKee  
Department of Computer Science  
University of Utah  
Salt Lake City, UT 84112  
{ czhang | sam }@cs.utah.edu

## ABSTRACT

Memory system bottlenecks limit performance for many applications, and computations with strided access patterns are among the hardest hit. The streams used in such applications have extremely poor cache behavior. These access patterns have the advantage of being predictable, though, and this can be exploited to improve the efficiency of the memory subsystem in two ways: memory latencies can be masked by prefetching stream data, and the latencies can be reduced by reordering stream accesses to exploit parallelism and locality within the DRAMs. Many researchers have studied hardware prefetching in its various forms. Others have examined dynamic memory scheduling to help bridge the performance gap between processors and DRAM memory systems. This study builds on these results, combining a stride-based reference prediction table, a mechanism that prefetches L2 cache lines, and a memory controller that dynamically schedules accesses to a Direct Rambus memory subsystem. We find that such a system delivers good speedups for scientific applications with regular access patterns without negatively affecting the performance of non-streaming programs.

## 1. INTRODUCTION

Memory system bottlenecks are becoming the limiting performance factors for many applications, and streamed computations with strided access patterns are among those whose performance suffers most acutely. The vectors used in such applications lack temporal and often spatial locality, and thus have poor cache behavior. Nonetheless, these access patterns have the advantage of being predictable, and this predictability can be exploited to improve the efficiency of the memory subsystem — the memory controller and the DRAM back end.

Previous work has examined memory scheduling mechanisms in the context of compiler or application-supplied in-

formation about access patterns [22, 16, 20]. Here we investigate whether it makes sense to reorder accesses within the memory controller in the absence of compiler- or application-supplied access pattern information. For current processor and memory technologies, the processor's natural reference stream provides the ordering mechanism with few choices about which access to issue next, and this lack of choice severely limits the memory controller's ability to exploit properties of the DRAM back end or to alleviate burstiness on the bus. Fortunately, access ordering techniques have much more opportunity to improve performance when knowledge of future access patterns is available (e.g., when prefetching is used).

Others have studied hardware prefetching in depth [17, 14, 11, 26]. We leverage their work to provide an access-ordering memory controller with information about stream reference patterns. We investigate the extent to which a particular combination of a hardware prefetching mechanism and reordering memory controller can improve performance for a suite of benchmarks ranging from vector kernels to irregular heap- and pointer-intensive programs to regular scientific applications. The systems we study combine a stride-based reference prediction table (RPT), a mechanism that prefetches L2 cache lines, and a memory controller that dynamically schedules accesses to a Direct Rambus memory subsystem [28]. We implement a simple reordering policy and evaluate its performance impact on a set of integer and floating-point applications and a set of vector kernels. By avoiding DRAM bank conflicts and bus-turnaround delays, our approach consistently delivers performance exceeding that of prefetching alone. Applications with little streaming potential may suffer negligible performance degradations, but these adverse effects are rare for a system that combines incremental prefetching with access ordering. Most of our results demonstrate at least a small performance improvement, and for several regular, memory-intensive programs and kernels, run time is halved.

## 2. ARCHITECTURE

Our approach reduces access latency and improves bus utilization by combining dynamic access ordering within the memory controller with a prefetching mechanism that incorporates transparent hardware stream detection. Figure 1 illustrates the system organization, which includes a *Reference Prediction Table* [2] (RPT) between the CPU and the L2 cache. The RPT observes the reference pattern gener-

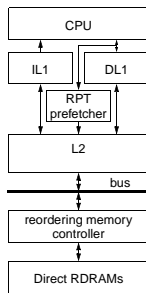


Figure 1: System Architecture.

ated by the CPU to detect strided access patterns and then to prefetch L2 cache lines based on these patterns. The prefetches increase the average number of ready accesses at the memory controller, allowing it to prioritize references in an attempt to avoid unnecessary bus turnaround delays and to exploit parallelism in a Direct Rambus DRAM back end.

## 2.1 Prefetching Mechanism

Our prefetching hardware is based on the reference prediction tables introduced by Chen and Baer [2], and is organized as a 64-entry, four-way associative cache indexed by the addresses of memory reference instructions. The RPT is not on the critical path to memory, and does not slow normal cache accesses. Each RPT entry maintains four fields:

- *tag*: the address of the load/store instruction,
- *prev\_address*: the previous operand address for that instruction,
- *stride*: the difference between the last two operand addresses, and
- *state*: two bits used to indicate past history for this reference pattern (one of  $\{initial, transient, irregular, steady\}$ ).

Figure 2 depicts the state transition mechanism. By tracking stores as well as loads, the RPT prefetches cache lines for the write-allocate/write-back L2 cache. Prefetch requests are issued when an RPT entry is at steady state and has correctly predicted the current operand address (i.e.,  $addr - prev\_addr = stride$ ). For a prefetch distance of  $d$ , the RPT issues requests for  $addr + stride$ ,  $addr + 2 \times stride$ ,  $\dots$   $addr + d \times stride$ . The prefetching operations for a given stream obey a sliding window protocol: when  $addr$  is referenced, if the prefetches for  $addr + stride$  to  $addr + (d-1) \times stride$  have already been issued, then only one request ( $addr + d \times stride$ ) is generated. The window status is represented by two registers,  $[L, R]$ , indicating the range of offsets (from the most recently referenced stream element  $addr$ ) for which prefetch requests may be issued (so the valid prefetch addresses range from  $addr + L \times stride$  to  $addr + R \times stride$ ). When the request for  $addr + L \times stride$  is issued, the window is updated to  $[L+1, R]$ . As outstanding prefetches arrive, the implicit base  $addr$  is incremented by  $stride$ , and the window slides to  $[L, R-1]$ . If  $L > R$ ,

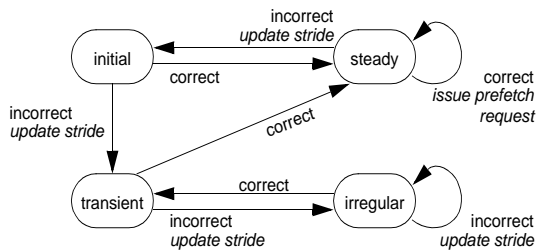


Figure 2: Reference Prediction Table state transitions.

the window is empty, and no prefetches for this stream remain outstanding. The regular nature of the request sequence eliminates the need for buffering requests within the prefetcher.

We investigate mechanisms with both fixed and adaptive prefetch distances (as in the designs of Farkas *et al.* [14] for a different machine model with dedicated stream buffers). The adaptive scheme prefetches streams incrementally, starting with a unit prefetch distance, and doubling it (up to the maximum distance supported) every time another element of the stream is referenced. We refer to a reference with the appropriate stride as a “prefetch hit”, even though the element the processor references to trigger a prefetch operation and the element for which we initiate that prefetch are separated by the prefetch distance multiplied by the stream stride. As distance increases, so does the threshold for the number of references that must be observed in a stream pattern before future cache lines in the sequence are prefetched. This decreases the likelihood that spurious stream prefetches will be issued, but it can increase cache contention. Prefetch misses (data addresses that do not fit the established pattern for that memory instruction) signal the stream end. The RPT we model can support up to 32 outstanding requests.

## 2.2 Memory Controller

Dynamic stream detection mechanisms have only a local view of the program’s behavior, and thus are inherently limited in the amount of “future information” that they can provide to the access ordering hardware. This restricts the choices available to the ordering hardware, which limits the extent to which the memory controller can exploit the parallelism of multiple memory devices and many interleaved banks. On the other hand, the limited choice simplifies the burden on the access ordering mechanism. The reordering circuitry cannot lengthen the timing path to memory, and therefore must be simple.

We model only one ordering algorithm here. The ordering mechanism implements a greedy policy that attempts to keep the pipelined Rambus memory channel busy by giving highest priority to the access that can be issued soonest. The circuitry maintains a candidate for the next memory access whenever there is more than one access queued, and for each incoming request it compares the soonest-issue time with that of the candidate to decide which to issue next. When two accesses have the same issue time, they are serviced in FIFO order, with the restriction that reads do not bypass

writes in the case of a conflict. We investigated an ordering scheme that gives demand accesses priority over prefetches, but the differences in execution time were less than 1% for our benchmarks, and such a scheme requires a mechanism to let the memory controller distinguish between prefetches and demand cache line fills. The computation of the next candidate can be completely overlapped with other memory activity, thus our approach requires only a single additional comparison to do the ordering. For the purposes of this study, we assume that this comparison can be accomplished within the memory cycle time.

### 3. EXPERIMENTAL METHODOLOGY

We use cycle-level simulation to evaluate the effectiveness of the proposed memory system. Our dynamically scheduled superscalar simulator is based on *sim-outorder* from the SimpleScalar toolset version 2.0 [5]. These tools use a MIPS-like instruction set, and they only execute user-level code, thus we do not model the effects of OS interactions on application memory performance, nor do we quantify the impact of invalidating the RPT on context switches.

Our simulator models a Direct Rambus memory and the stream detection mechanism from Section 2.2. We choose the Rambus model for several reasons: it represents the state of the art in affordable, high performance memory systems; its unique organization and interface presents the memory system designer with an interesting set of challenges; the performance potential of its pipelined interface and highly parallel subsystem can only be realized by carefully scheduling operations to overlap as much as possible.

#### 3.1 Machine Model

We model a 4:1 ratio in CPU cycles to memory cycles, or a 1.6 GHz processor and 400 MHz Rambus Channel. The CPU performs out-of-order execution with a 16-entry instruction window, issuing up to four instructions per cycle. The two-level, nonblocking cache hierarchy consists of separate, identical first-level instruction and data caches of 64KB each. The L1 caches are two-way associative, virtually indexed, and physically tagged; each has 32-byte lines, a one-cycle hit latency, and can support up to eight outstanding misses. The L2 cache is 256 kilobytes and four-way associative with 64-byte lines. It has a six-cycle hit latency, is physically indexed and tagged, and has eight MSHRs (Missing Status Holding Registers). We scaled down the L2 capacity because the workloads we use are relatively small compared to the working sets of real applications (see Table 3 for details of our benchmarks’ on-chip memory hierarchy performance). The chip real estate consumed by the mechanisms we introduce to support prefetching and reordering might reduce the L2 capacity slightly, but certainly not by the factor we have modeled here.

#### 3.2 Direct Rambus DRAMs

Although the memory core—the banks and sense amps—of RDRAMs is similar to that of other DRAMs, the architecture and interface are unique. An RDRAM is actually an interleaved memory system integrated onto a single memory chip. Its pipelined microarchitecture supports up to four outstanding requests. The Direct Rambus interface converts the 10 ns on-chip bus, which provides 16 bytes on each internal clock, to a two-byte wide, external, 1.25 ns bus. By

name	cycles	description
$t_{PACK}$	4	packet transfer time
$t_{RC}$	28	row (i.e., page miss) cycle time or RDRAM banks: interval between successive ROW ACT requests to same bank
$t_{RAS}$	20	RAS-asserted time of RDRAM bank: interval between ROW ACT packet and next ROW packet w/ PRER to same bank
$t_{RP}$	8	row precharge time: interval between ROW PRER and ROW ACT packets to same bank
$t_{RR}$	8	RAS-to-RAS time of RDRAM device: interval between successive ROW ACT packets to same device (any banks)
$t_{RCD}$	9	RAS-to-CAS delay: interval between ROW ACT and COL RD or WR packets
$t_{CAC}$	8	CAS access delay (page hit latency): delay between start of COL RD packet and valid data
$t_{CWD}$	6	CAS write delay: interval between COL WR packet and write data
$t_{CC}$	4	CAS-to-CAS time of RDRAM bank: interval between successive COL packets
$t_{RDP}$	4	interval between last COL RD packet and ROW PRER packet

Table 1: Direct Rambus timings for Min -45 -800 part

transferring 16 bits of data on each edge of the 400MHz interface clock, even a single Direct RDRAM chip can yield up to 1.6 Gbytes/sec in bandwidth.

All communication to and from an RDRAM is performed using packets. Each command or data packet requires four 2.5 ns clock cycles to transfer. ROW command packets are used for activate (ACT) or precharge (PRER) operations. COL command packets are used to initiate data transfer between the sense amps and the data bus (via RD or WR commands), or to retire data in the chip’s write buffer. The smallest addressable data size is 128 bits (two 64-bit stream elements). The full memory bandwidth cannot be utilized unless all words in a DATA packet are used. Note the distinction between the RDRAM transfer rate (800 MHz), the RDRAM interface clock rate (400 MHz), and the packet transfer rate (100 MHz). “Memory cycles” refer to the 400 MHz interface clock.

Table 1 gives the relevant Direct RDRAM timing parameters used in our simulations. The RDRAM cores incorporate 16 banks in a “double bank” architecture, where adjacent banks share sense amplifiers. (so no two adjacent banks can be active simultaneously) [28]. Note that we do not model the DRAM write buffers in detail, but all other timing interactions are simulated accurately.

#### 3.3 Memory Models

The memory systems we model consist of eight 64 Mbit Direct Rambus devices on a single channel. We examine two address mappings with respect to the RDRAM devices: cache-line interleaved (for the 64-byte L2 cache lines), and page interleaved. Both organizations use a closed-page precharge policy, i.e., the DRAM page is closed and the sense amplifiers are precharged after each access. The memory controller tries to hide precharge latencies by overlapping them with references to other banks or devices whenever possible.

The organization of our memory systems differs substantially from the single-device systems studied by Hong *et al.* [16], since our systems contain eight devices (affording much more parallelism). Their systems fetch individual stream elements, instead of performing cache line fills as we do here. Accessing larger granularities of data can re-

Kernel	Access Pattern
copy	for (i=0; i<L×S; i+=5) y[i]=x[i];
daxpy	for (i=0; i<L×S; i+=5) y[i] += a × x[i];
swap	for (i=0; i<L×S; i+=5) {reg=x[i]; x[i]=y[i]; y[i]=reg;}
vaxpy	for (i=0; i<L×S; i+=5) y[i] += a[i] × x[i];

Table 2: Benchmark kernel access patterns.

duce the opportunity to overlap precharges with accesses to other banks or devices. Nonetheless, we find that precharge delays can usually be overlapped with other activity, and thus we do not investigate systems with open-page policies here.

### 3.4 Benchmark Suite

Most of our benchmarks come from the SPEC95 suite; we simulate them for the test inputs, unless otherwise noted. `m88ksim` is a chip simulator for the Motorola 88100 microprocessor. `gcc` is the `cc1` pass of the version 2.5.3 gcc compiler (for SPARC architectures) used to compile the 67-kilobyte file “`integrate.s`”. `compress` is the SPEC95 data compression program run on an input file of ten thousand characters. `li` is an Xlisp interpreter run on the “queens” problem for a  $8 \times 8$  board. `su2cor` applies a Monte-Carlo method to the computation of masses of elementary particles in the framework of the Quark-Gluon theory, and `hydro2d` solves hydrodynamical Navier Stokes equations to compute galactical jets, `mgrid` is a multi-grid solver in 3D potential field, and `swim` solves shallow water equations using finite difference approximations (all three are run on the ref input).

In addition, we investigated a set of smaller benchmarks with both pointer- and array-based access patterns [1]. `anagram` computes the set of anagrams for its input string. `bc` is the gnu basic calculator. `ks` is a graph partitioning tool. `ft` performs a minimum span calculation, and `yacr2` is a channel routing program.

### 3.5 Microbenchmarks

To put our results in perspective with previous results on compiler-assisted dynamic access ordering, we also use the benchmark kernels of Hong *et al.* [16] to evaluate our design’s performance potential. Table 2 lists these kernel access patterns. `daxpy`, `copy`, and `scale` are from the BLAS (Basic Linear Algebra Subroutines) [12]. `vaxpy` denotes a “vector axpy” operation that occurs in matrix-vector multiplication by diagonals: a vector  $a$  multiplied by a vector  $x$  plus a vector  $y$ . We run each kernel for 10,000 iterations for two access patterns: unit-stride and stride-ten (the size causes only one data element to reside in each L2 cache line, and being even causes more DRAM bank conflicts).

## 4. RESULTS

Table 3 details the performance characteristics of each benchmark, including the execution time on both memory organizations, the number of instructions retired, and the number of those that are loads or stores. The table also gives statistics on the on-chip memory hierarchy performance and on the number of streams recognized and the average length of dynamically recognized stream access patterns for the whole program.

Figure 3 and Figure 4 summarize the performance (relative to the each benchmark baseline) of the adaptive-distance prefetching schemes for the two memory interleavings, with and without dynamic reference reordering at the memory controller. The similarity in these graphs illustrates that for the benchmarks used, the mapping of addresses to memory banks makes little performance difference for applications that hardly benefit from stream prefetching. For the applications and kernels that do benefit from streaming, the performance differences between the organizations ranges from 7% to less than 1% of the baseline execution time, but which system performs better varies from benchmark to benchmark. The hardware stream detection and memory-controller access scheduling benefit the stream-oriented applications, regardless of the underlying memory interleaving.

`swim`, `mgrid`, and `hydro2d` are the most stream-oriented applications of the suite. `swim` uses over three million dynamically recognized streams, and has the longest mean stream length of the non-kernel benchmarks. `mgrid` accesses exhibits about twice as many stream patterns, but the average stream length is only half as long. `hydro2d` accesses slightly fewer streams, with a slightly smaller average length, but it enjoys similar speedups to `mgrid`. The explanation lies in `hydro2d`’s cache performance: with an L1 miss rate of 10% and an L2 miss rate of over 36%, this application exhibits little locality and is very memory intensive (over 36% of the instructions executed are loads or stores). In contrast, `su2cor` exhibits behavior similar to `hydro2d` in terms of number and average length of streams, yet it derives almost no benefit from stream prefetching and reordering. In this case, the on-chip cache hierarchy performs quite well. With fewer than 3% of the references missing in the L1 cache and fewer than 8% missing in the L2, `su2cor` leaves little opportunity for memory prefetching to affect performance.

The pointer benchmarks exhibit a relatively high number of stream patterns, but the ones with the most streams have the shortest average stream lengths, and those with the longest stream lengths use fewer streams. Performance for these applications does not improve from streaming and reordering, but neither does it degrade. The larger integer benchmarks with similar stream characteristics (`gcc`, `m88ksim`, and `li`) also exhibit stable performance. The dynamic behavior of these programs is not conducive to streaming: in the case of `bc`, fewer than 2% of the memory accesses occur in detectable streams. For `gcc`, the rate rises to almost 10%, but the average stream length is seven references, and only about 9% of the references make up strided access patterns. `li` has such a small working set that fewer than 1% of the L2 accesses miss, and the reference patterns for this benchmark are sufficiently irregular that the average stream length is under five.

Figure 5 and Figure 6 illustrate how different RPT prefetch distances affect the performance of a representative set of the benchmarks. Applications that benefit from streaming tend to perform better with larger thresholds, but this is not always the case. For example, on a page-interleaved system, the unit-stride `copy` kernel in Figure 6(e) slows down by up to 20% at a prefetch distance of two, and by over 4% at a prefetch distance of either, whereas simply prefetching the next cache line (a scheme that sometimes slows performance,

Benchmark	Cycles		Instructions (thousands)	Loads	Stores	L2 Misses	DTLB Misses	Total Streams	Mean Length
	cache line interleaved	page interleaved							
gcc	220928	217253	224072	58587	31770	426005	10088	177862	7
m88ksim	229730	229977	492995	85451	41825	13891	350	3295413	14
ii	545036	545008	956747	286320	168848	2380	8513	16058335	4
hydro2d	2363036	2141605	967197	191856	58693	12022852	896	6643674	25
mgrid	1250866	1266649	1137368	399864	16229	4627143	279	7349938	39
su2cor	787081	783229	1034337	250716	79981	704370	43599	3077429	56
swim	1821060	1676789	1306225	327039	85805	5463881	1014565	3528529	89
anagram	9659	9676	17946	4298	1740	8221	160	100425	10
bc	9286	9268	14262	3190	1803	2264	55	52132	3
ft	15131	15120	23957	4170	1462	3506	24	59769	32
ks	10338	10320	12307	5310	109	1695	1691	473260	4
yacr2	17946	17921	37972	6091	3846	2127	47	417072	13
copy	564	444	336	70	23	2817	9	19	1728
copy stride 10	3526	2643	346	70	23	28338	53	62	528
daxpy	611	484	356	80	23	2818	9	19	1728
daxpy stride 10	3603	2657	366	80	23	28345	53	62	528
vaxpy	699	656	426	90	23	4146	12	305	140
vaxpy stride 10	4041	3854	436	90	23	36235	78	111	374
swap	492	462	426	90	43	1917	9	20	2141
swap stride 10	2762	2614	436	90	43	23149	53	63	678

Table 3: Characteristics of each baseline run.

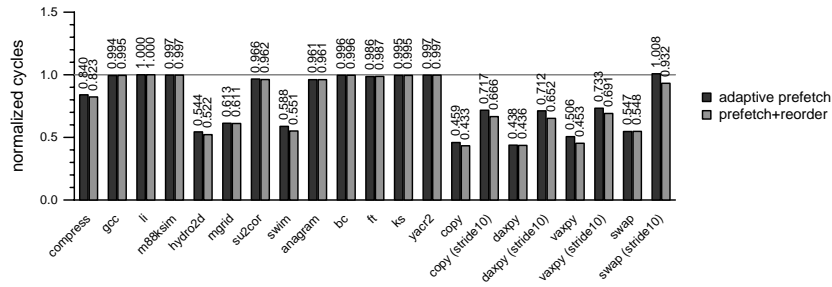


Figure 3: Normalized execution times for a cache line interleaved system and prefetching with an adaptive prefetch distance, with and without dynamic access ordering at the memory controller.

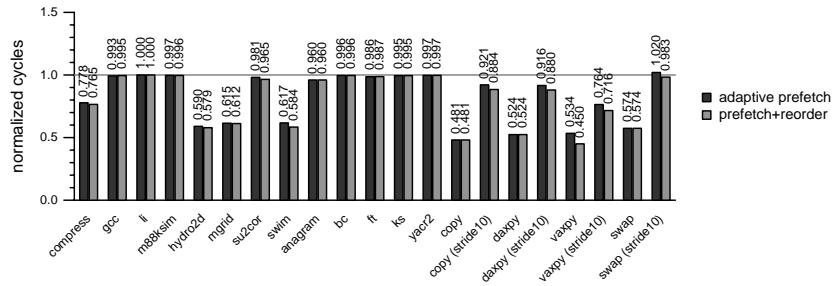


Figure 4: Normalized execution times for a page interleaved system and prefetching with an adaptive prefetch distance, with and without dynamic access ordering at the memory controller.

since prefetches are issued without the prefetcher's actually recognizing any access patterns) improves performance by over 6%. Note that none of our experiments measurably affects TLB performance.

Common sense dictates that the prefetch distance needs to be about the same as or larger than the cache line to realize much benefit beyond the prefetching side effects of the usual demand cache line fills for unit stride patterns. Table 4 gives statistics on the number and effectiveness of prefetches issued for each of the prefetching schemes and each of the benchmarks highlighted in Figures 5 and 6. Increasing the window up to 16 puts the portion of prefetch requests that have arrived in cache by the time they're accessed by the processor between 26% and 81% for the application benchmarks. For the `copy` kernel, this effectiveness rate hits 92%. Nearly all the prefetches hit either in cache or in the MSHRs, so even when prefetched data is not ready when the processor requests it, the latency observed by the CPU will be reduced. Table 5 shows how the L2 miss rate goes down with the prefetch distance, but these improvements do not continue to scale. Larger prefetching distances run the risk of lowering memory system performance by generating prefetches beyond the ends of the streams and by increasing contention in the cache. Also, prefetched data that arrives too early could be evicted before it is used.

For stream-intensive programs, using an incremental prefetch distance performs almost as well as the largest prefetch distance we study (16). The adaptive mechanism sometimes slows these applications slightly, but it also mitigates performance degradations for programs with pathological access patterns (i.e., lots of very short streams). This approach delivers robust performance, especially when combined with access ordering to exploit the characteristics of the underlying memory system.

Note that the benefit of prefetching and reordering goes down dramatically for the kernels with a stride-ten access pattern. Each stream element accessed brings in a cache line whose other contents are unneeded. This prevents the memory latency from being amortized over more than one access, and the streams' large cache footprint creates significant contention. Nonetheless, with prefetching and reordering the stride-ten `copy` kernel suffers only about half the L2 cache misses of the baseline (but twice as many as the unit-stride version of the kernel), and overall execution time is reduced by about 10% of the baseline's.

All of our simulation results reinforce the conclusion that reordering never hurts. Realizing nontrivial speedups requires a prefetch distance that is large enough to give the reordering mechanism a choice about which banks it accesses when. For the benchmarks we simulate, dynamically scheduling DRAM accesses at the memory controller lowers the execution time by up to another 8.4% of the baseline's performance for an incremental prefetch mechanism that can "look" into the future by up to sixteen stream references. As the pressure on the memory system increases, and so does the opportunity for the mechanism to improve performance. For example, on an 8-issue processor with a 32-instruction window, `swim` gains a speedup of 9.9% from reordering, compared to 7.1% in the 4-wide machine configuration. Since

the RDRAMs we model use a closed-page precharge policy, this speedup from reordering comes from better exploiting the parallelism in the RDRAM systems.

## 5. RELATED WORK

Prefetching encompasses a broad range of memory access techniques involving software, hardware, or both. The purely software approach relies on a compiler to generate instructions to preload data [24, 23], or an application writer to modify source code to achieve the desired behavior [4, 27, 19]. Hybrid approaches include hardware support for prefetch operations, exposing those mechanisms to software. For instance, they might augment the ISA with a prefetch instruction [13], redefine a load to a specific register (e.g., to register 0, as in the PA-RISC architectures [18]), or provide programmable prefetch engines [8] or programmable stream buffers [22].

Baer and Chen [2], Fu and Patel [15], and Sklenar [31] propose dynamic vector prefetch units that induce stream parameters at run-time. The cache-based sequential hardware prefetching of Dahlgren *et al.* [11] eliminates the need for detecting strides dynamically. To minimize the number of unnecessary prefetches, the prefetch distance of these run-time techniques is generally limited to a few loop iterations (or a few cache lines). As in the approach we investigate, the prefetched data may replace other needed data, or may be evicted before it is used. Hardware-only prefetching [2, 17, 11, 14, 31] thus has the advantage of being transparent, but because of its speculative nature, care must be taken to keep from lowering application performance by increasing contention in the caches and wasting bus bandwidth on useless prefetches. Nonetheless, some commercial machines include such mechanisms [10, 30, 7].

Prefetching masks memory latency, but generally does not attempt to improve the operation of the memory system back end. Prefetching techniques can be rendered more effective by combining them with *access ordering* — static or dynamic techniques to improve memory performance by changing the order of memory requests [21] — to exploit the architectural and device characteristics of the underlying memory system.

Most dynamic access ordering approaches to date have relied on the compiler or the application to supply reference pattern information. For instance, Palacharla and Kessler [27] investigate code restructuring techniques to exploit a unit-stride *read-ahead* stream buffer and page mode memory devices on the Cray T3D [10]. The read-ahead mechanism operates like Jouppi's stream buffers [17], prefetching the next sequential cache line on a demand cache-line fill. In Palacharla and Kessler's approach, the order in which vectors are fetched is decided at compile-time, but they avoid cache conflicts by determining at run-time the amount of each vector to fetch at once. They measure a performance improvement of up to 75% in two, three, and four-stream examples. The performance benefits are substantial, but this approach offers little flexibility: "programming" the streaming mechanism amounts to rearranging the source code to present the hardware with an appropriate sequence of addresses. Effectively exploiting these stream buffers thus requires significant modifications to the source program. Get-

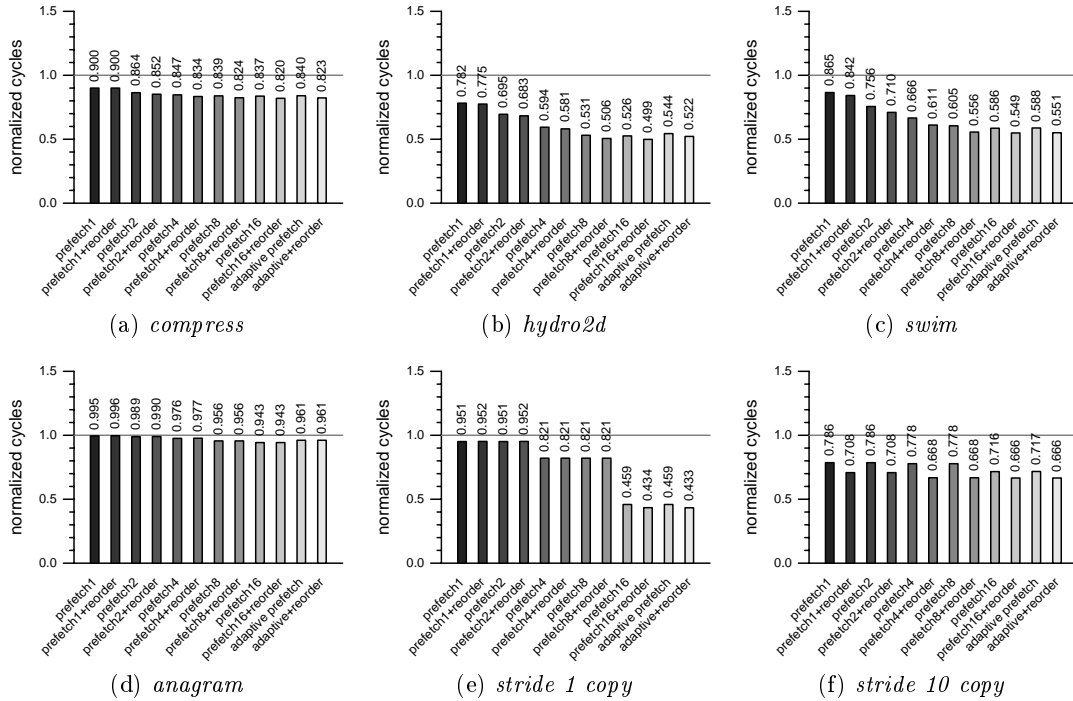


Figure 5: Performance details for selected benchmarks on a cache line interleaved system.

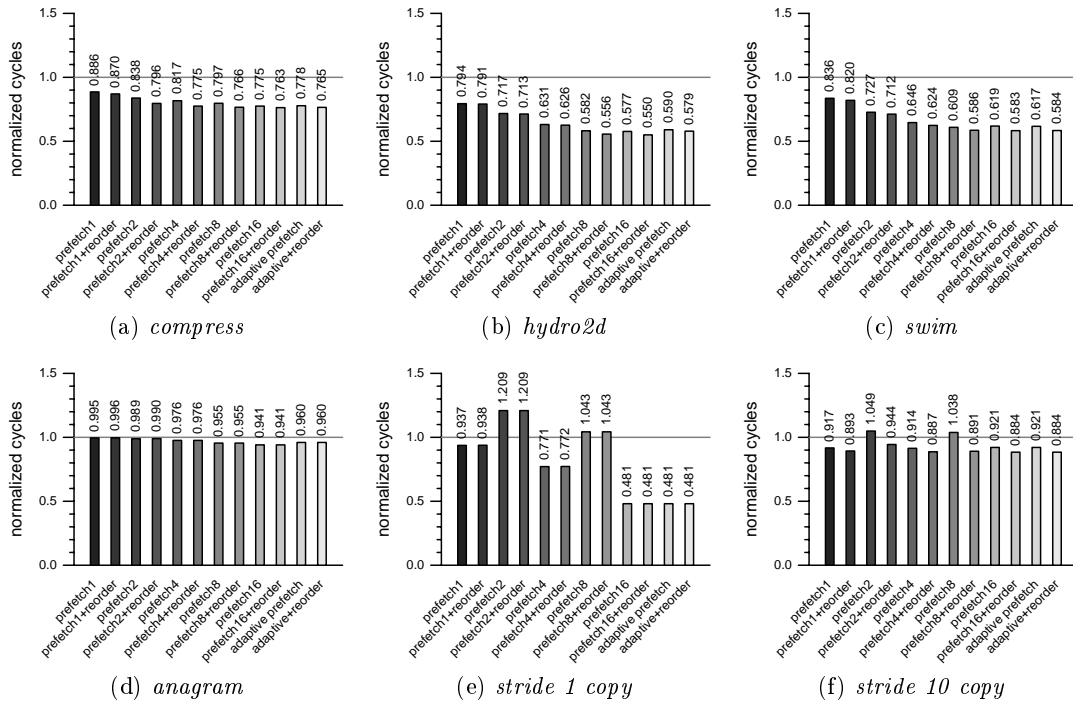


Figure 6: Performance details for selected benchmarks on a page interleaved system.

Benchmark		cache line interleaved						page interleaved					
		$d = 1$	$d = 2$	$d = 4$	$d = 8$	$d = 16$	adaptive	$d = 1$	$d = 2$	$d = 4$	$d = 8$	$d = 16$	adaptive
		compress	prefetched % in cache % in MSHRs	98980 2.4 97.1	99082 5.2 94.3	99142 11.2 88.2	99307 20.6 78.7	99707 26.3 72.8	99598 24.4 74.7	98980 2.6 96.9	99082 16.2 83.3	99142 16.6 82.8	99307 25.0 74.3
hydro2d	prefetched % in cache % in MSHRs	11402061 6.0 91.5	11412350 11.6 85.8	11428218 20.9 76.5	11433220 45.2 52.2	11437641 53.0 44.4	11433632 61.8 35.6	11402047 3.5 93.9	11412339 8.3 89.1	11428177 20.6 76.8	11433221 45.1 52.3	11437608 50.2 47.2	11433629 61.6 35.8
swim	prefetched % in cache % in MSHRs	7819037 7.1 91.4	7828283 17.3 85.1	7839100 29.4 81.8	7840869 51.5 39.3	7844758 63.2 17.2	7841088 63.3 5.3	7818296 9.2 59.2	7827452 16.0 52.5	7839100 29.4 39.3	7840342 52.9 15.6	7847694 64.4 4.1	7844146 64.6 3.9
anagram	prefetched % in cache % in MSHRs	3402 1.8 91.7	3883 6.1 87.8	4830 12.9 81.8	5536 27.9 66.5	6205 81.0 12.0	5015 33.3 61.7	3402 1.8 91.6	3883 6.3 87.6	4830 13.6 81.0	5536 28.4 66.0	6205 80.2 12.9	5015 32.8 62.2
copy	prefetched % in cache % in MSHRs	2666 0.0 98.4	2666 0.0 98.4	2667 0.1 98.3	2666 0.1 98.3	2671 74.8 23.5	2669 74.5 23.8	2667 0.0 98.4	2666 0.0 98.4	2668 0.2 98.1	2666 0.1 98.3	2672 92.8 5.4	2670 92.5 5.7
copy stride 10	prefetched % in cache % in MSHRs	19770 4.1 95.9	19770 4.1 95.9	19795 14.8 85.2	19794 14.8 85.2	19797 61.2 38.8	19795 61.8 38.2	19770 2.8 97.2	19770 4.1 95.9	19795 8.8 91.2	19794 14.8 85.2	19797 52.5 47.5	19795 53.2 46.8

Table 4: Number of prefetches issued and the percentage thereof that were useful (i.e., that hit in either L2 cache or the RPT’s MSHRs) for the benchmarks in Figure 5 and Figure 6.

Benchmark	Percentage of Baseline L2 Cache Misses											
	cache line interleaved						page interleaved					
	$d = 1$	$d = 2$	$d = 4$	$d = 8$	$d = 16$	adaptive	$d = 1$	$d = 2$	$d = 4$	$d = 8$	$d = 16$	adaptive
compress	99.3	97.3	92.7	85.6	81.3	82.7	99.1	88.9	88.6	82.2	78.0	79.4
hydro2d	95.5	90.4	81.7	58.9	51.5	43.0	98.1	93.7	82.2	50.0	54.2	43.3
swim	90.1	75.5	58.1	26.4	9.8	9.8	87.0	77.3	58.1	24.3	8.0	7.7
anagram	102.4	100.6	96.4	86.4	47.6	83.5	102.4	100.5	96.0	86.1	48.2	83.9
copy	100.0	100.0	99.9	99.9	29.1	29.4	100.0	100.0	99.8	99.9	12.0	12.3
copy stride 10	97.2	97.2	89.7	89.7	56.9	56.5	98.0	97.2	93.8	89.7	62.9	62.5

Table 5: Percentage of L2 cache misses for each streaming scheme relative to the baseline for the benchmarks in Figure 5 and Figure 6.

ting the best performance requires that interference be taken into account, and thus the optimal amount to preload for each data structure of cannot be generated until run time.

McKee *et al.* [22] rely on the compiler [3] to detect streams and generate code to program their memory controller’s stream buffers at run time. The memory controller reorders the stream accesses to exploit the parallelism of the interleaved banks and to exploit locality of reference within the DRAM’s page buffers. They demonstrate speedups of up to a factor of thirteen for streaming kernels on their uniprocessor prototype hardware. That system contained only two interleaved DRAM banks, and thus increasing the number of references that hit in the page buffers accounted for most of the performance improvements. Hong *et al.* [16] adapt the approach to single-device Direct Rambus memory systems. Bus-turnaround delays become a limiting performance factor for these highly parallel, pipelined memory systems. Most of the improvement from access ordering comes from overlapping operations to multiple banks and from minimizing the number of times the memory controller switches between reading and writing. Neither of these studies evaluates the impact of reordering stream accesses on whole-program performance.

Corbal *et al.*’s Command Vector Memory System [9] also exploits parallelism and locality of reference to improve effective bandwidth for vector accesses on out-of-order vector processors with dual-banked SDRAM memories. The vectorizing compiler generates *vector commands* requesting multiple, independent words. Instead of sending individual requests to specific devices, the memory controller broadcasts these vector commands. The memory subsystem orders requests to each dual-banked device, attempting to overlap precharge operations to each internal SDRAM bank

with access operations to the other. This system buffers stream data in vector registers within the CPU.

Like the Command Vector Memory System, Mathew *et al.*’s Parallel Vector Access unit [20] operates on vector commands and exploits SDRAM device characteristics, gathering strided data even more efficiently. The PVA is part of a memory controller [6] that increases processor cache and memory bus utilization by dynamically remapping physical memory, letting applications control how data is cached on chip. This approach prefetches and buffers data within the memory controller until the CPU requests them.

## 6. CONCLUSION

Several recent architectures propose to migrate more intelligence into the memory system [6, 25, 29] to help bridge the processor/memory performance gap. This paper explores the potential for diverse applications to benefit from hardware-only memory access ordering, and shows how relatively simple mechanisms can realize at least some of that potential. The strided prefetcher provides the memory controller’s access-ordering mechanism with enough choice to make more efficient use of the memory subsystem.

We investigate a particular point in the design spectrum for streaming hardware, and demonstrate that a straightforward and modest-sized reference prediction table that prefetches into the L2 cache, coupled with a simple DRAM scheduling mechanism, can deliver substantial performance gains for memory-bound, stream-intensive applications. The more references in sequence required to define a stream, the more robust the performance benefits for all types of applications. Next-cache-line prefetchers generate too many spurious prefetches, and distance-two prefetchers often perform even worse. Distances of eight or sixteen

yield the best performance for our benchmarks and systems. Combining these with simple access ordering at the memory controller further decreases execution time from an insignificant margin up to 8%.

For the prefetching schemes we study, the memory interleaving does not affect the memory controller's ability to optimize performance. Larger prefetch distances equate to better performance for nearly all applications, and our prefetching model delivers consistent performance increases, even in the absence of memory reordering, and in spite of additional cache contention. Our results indicate that, given sufficient choice with respect to scheduling DRAM accesses, an access-ordering memory controller can deliver nontrivial speedups. A memory subsystem that combines incremental prefetching with a reordering DRAM scheduler decreases the run times of a set of memory-intensive inner loops by over a factor of two (rivaling Hong *et al*'s compiler-assisted, single-RDRAM dynamic access ordering system [16]). Our system delivers comparable benefits for scientific applications, without slowing the performance of non-stream applications.

## ACKNOWLEDGMENTS

This research was sponsored in part by National Science Foundation award 9806043. The authors thank Steve Reinhardt and Wei-fen Lin for providing the initial Rambus model. Discussions with John Carter, Lixin Zhang, and Mike Parker helped shape this study.

## REFERENCES

- [1] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Proc. of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 290–301, June 1994.
- [2] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proc. of SC'91*, pp. 176–186, Nov. 1991.
- [3] M. Benitez and J. Davidson. Code generation for streaming: An Access/Execute mechanism. In *Proc. of the 4th ASPLOS*, pp. 132–141, Apr. 1991.
- [4] J. Brooks. Single PE optimization techniques for the cray T3D system. In *Proc. of the 1st European T3D Workshop*, Sept. 1995.
- [5] D. Burger and T. Austin. The simplescalar toolset, version 2.0. TR 1342, University of Wisconsin, June 1997.
- [6] J. Carter, W. Hsieh, L. Stoller, M. Swanson, L. Zhang, E. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proc. of the Fifth HPCA*, pp. 70–79, Jan. 1999.
- [7] K. Chan, C. Hay, J. Keller, G. Kurpanek, F. Schumacher, and J. Zheng. Design of the HP PA 7200 CPU. *Hewlett-Packard Journal*, 47(1):25–33, Feb. 1996.
- [8] T.-F. Chen. Effective programmable prefetch engine for on-chip caches. In *Proc. of IEEE/ACM 28th International Symposium on Microarchitecture*, pp. 237–242, Nov. 1995.
- [9] J. Corbal, R. Espasa, and M. Valero. Command vector memory systems: High performance at low cost. In *Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 68–77, Oct. 1998.
- [10] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, Sept. 1993.
- [11] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 6(7):733–746, July 1995.
- [12] J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Trans. on Mathematical Software*, 16(1):1–17, Mar. 1990.
- [13] J. Edmondson, et al. Internal organization of the alpha 21164, a 300-mhz 64-bit quad-issue cmos risc microprocessor. *Digital Technical Journal*, 7(1):119–135, 1995.
- [14] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. of the 24th ISCA*, pp. 133–143, June 1997.
- [15] J. Fu and J. Patel. Data prefetching in multiprocessor vector cache memories. In *Proc. of the 18th ISCA*, pp. 54–65, Toronto, Canada, May 1991.
- [16] S. Hong, S. McKee, M. Salinas, R. Klenke, J. Aylor, and W. Wulf. Access order and effective bandwidth for streams on a direct rambus memory. In *Proc. of the Fifth HPCA*, pp. 80–89, Jan. 1999.
- [17] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proc. of the 17th ISCA*, pp. 364–373, May 1990.
- [18] G. Kane. *PA-RISC 2.0 Architecture*, 1996.
- [19] K. Lee. The NAS860 library user's manual. TR NAS TR RND-93-003, NASA Ames Research Center, Mar. 1993.
- [20] B. Mathew, S. McKee, J. Carter, and A. Davis. Design of a parallel vector access unit for sdram memories. In *Proc. of the Sixth HPCA*, pp. 39–48, Jan. 2000.
- [21] S. McKee and W. Wulf. Access ordering and memory-conscious cache utilization. In *Proc. of the First HPCA*, pp. 253–262, Jan. 1995.
- [22] S. McKee, et al. Design and evaluation of dynamic access ordering hardware. In *Proc. of the 1996 ICS*, May 1996.
- [23] L. Meadows, S. Nakamoto, and V. Schuster. A vectorizing software pipelining compiler for LIW and superscalar architectures. In *RISC'92*, pp. 331–343, 1992.
- [24] T. Mowry, M. S. Lam, , and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proc. of the 5th ASPLOS*, pp. 62–73, Oct. 1992.
- [25] M. Oskin, F. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proc. of the 25th ISCA*, pp. 192–203, June 1998.
- [26] S. Palacharla and R. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. of the 21st ISCA*, pp. 24–33, May 1994.
- [27] S. Palacharla and R. Kessler. Code restructuring to exploit page mode and read-ahead features of the cray T3D. TR Internal Report, Cray Research, Feb. 1995.
- [28] Rambus Inc. *Direct RDRAM 64/72-Mbit Data Sheet*, 1999. <http://www.rambus.com/developer/downloads/>.
- [29] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and D. Owens. A bandwidth-efficient architecture for media processing. In *Proc. of IEEE/ACM 31st International Symposium on Microarchitecture*, Dec. 1998.
- [30] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proc. of the 7th ASPLOS*, Oct. 1996.
- [31] I. Sklenar. Prefetch unit for vector operation on scalar computers. *Computer Architecture News*, 20(4):31–37, Sept. 1992.