Hardware-Only Stream Prediction + Cache Prefetching + Dynamic Access Ordering

Chengqiang Zhang and Sally A. McKee

{czhang | sam}@cs.utah.edu
http://www.cs.utah.edu/projects/impulse [1]

UUCS-99-020

Department of Computer Science
3190 Merrill Engineering Building
University of Utah
Salt Lake City, UT 84112
December 20, 1999

Abstract

The speed gap between processors and memory system is becoming the performance bottleneck for many applications, and computations with strided access patterns are among those that suffer most. The vectors used in such applications lack temporal and often spatial locality, and are usually too large to cache. In spite of their poor cache behavior, these access patterns have the advantage of being predictable, which can be exploited to improve the efficiency of the memory subsystem.

As a promising technique to relieve memory system bottleneck, prefetching has been studied in its various forms, and so is dynamic memory scheduling. 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 impressive speedups for scientific applications with regular access patterns (reducing execution time by almost a factor of two) without negatively affecting the performance of non-streaming programs.

Keywords: memory architecture, memory latency, memory bandwidth, prefetching, access ordering
Technical Areas: Architecture, Memory Systems, Vector Processing

[1] This research was sponsored in part by National Science Foundation award 9806043.

1 Introduction

Processor speeds continue to increase rapidly; so do memory speeds, but in a much slower fashion. Dynamic caching has long been used to bridge the gap between microprocessor and DRAM performance, but the effectiveness of reactive memory hierarchies is rapidly diminishing [BGK95, BGK96]. As a result, 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 are not reused quickly, and are often too large to cache, and thus they lack temporal locality. In addition, when the strides are not small, the accesses lack spatial locality. In spite of their poor cache behavior, 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 information about access patterns, either in uniprocessors [McK96, HMS+99, Mat99] or vector machines [CEV98]. Our work evaluates the potential benefits of hardware-only dynamic access ordering, and strives to address two questions: 1) how much information about future access patterns is necessary for access ordering to be profitable? and 2) given information about access patterns, how much can performance be improved?

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 backend 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). This paper investigates 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.

Many others have studied hardware prefetching in depth [Jou90, FCJV97, DDS95, PK94]. This study leverages their work to provide an access-ordering memory controller with information about reference patterns. The systems we investigate combine 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 [Ram99]. We implement a simple, greedy reordering policy, and evaluate its performance impact on a set of integer and floating-point applications as well as a set of vector kernels. By avoiding DRAM bank conflicts and bus-turnaround delays, our approach consistently delivers performance that exceeds that of prefetching alone. We find that our mechanisms deliver consistently good performance on two systems with different memory interleavings. Applications with little streaming potential occasionally suffer negligible performance degradations, but these adverse effects are rare for a system that combines incremental prefetching with access ordering. Most of our experiments demonstrate at least a small improvement in performance, and for several regular, memory-intensive programs and kernels, execution time is more than halved.

2 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 [MLG92, MNS92], or an application writer to modify source code to achieve the desired behavior [Bro95, PK95, Lee93]. Hybrid approaches include hardware support for prefetch operations, exposing those mechanisms to software. For instance, they might augment the ISA with a prefetch instruction [Edm95], redefine a load to a specific register (e.g., to register 0, as in the PA-RISC architectures [Kan96]), or provide programmable prefetch engines [Che95] or programmable stream buffers [McK96].

Chiueh [Chi94] proposes a programmable prefetch engine that fetches vector data for the next loop iteration. This data is stored in a special buffer, the *Array Register File*, until the corresponding iteration is executed, at which point the prefetched data is transferred to cache. Using a separate prefetch buffer avoids cache conflicts between the current and future working sets of vector data, but not between the vectors and the scalar data that they may displace. The scheme has a limited *prefetch distance*, the time between a prefetch operation and the corresponding load instruction.

Baer and Chen [BC91], Fu and Patel [FP91], and Sklenar [Skl92] propose dynamic vector prefetch units that induce stream parameters at run-time. The cache-based sequential hardware prefetching of Dahlgren *et al.* [DDS95] 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 [BC91, Jou90, DDS95, FCJV97, Skl92] 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 [Cra93, Sco96, CHK+96].

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 [MW95] — 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 [PK95] investigate code restructuring techniques to exploit a unit-stride *read-ahead* stream buffer and page mode memory devices on the Cray T3D [Cra93]. The read-ahead mechanism operates like Jouppi's stream buffers [Jou90], 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. Getting 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.* [McK96] rely on the compiler [BD91] 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.* [HMS+99] 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 [CEV98] 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 Vectory Memory System, Mathew's Parallel Vector Access unit [Mat99] operates on vector commands and exploits SDRAM device characteristics, gathering strided data even more efficiently. This subsystem is part of a memory controller [CHS+99] that increases processor cache and memory bus utilization by dynamically remapping physical memory, thereby letting applications control how their data is cached on chip. This approach prefetches and buffers data within the memory controller until the CPU requests them.

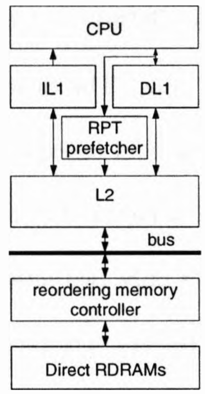

Figure 1: System Architecture.

3 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* [BC91] (RPT) between the CPU and the L2 cache. The RPT observes the reference pattern generated 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 the Direct Rambus RDRAM back end.

3.1 Prefetching Mechanism

Our prefetching hardware is based on the reference prediction tables introduced by Chen and Baer [BC91], 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 entry of RPT 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*, *stready*}).

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 $addr + stride$, $addr + 2 \times stride$, $\ldots addr + d \times stride$. To avoid re-issuing reqests for pending data.The prefetching operations for a given stream obey a sliding window protocol: if at the point when address $addr$ is referenced the prefetches for $addr + stride$ $\ldots addr + (d-1) \times stride$ have already been issued, then only one new request ($addr + d \times stride$) will be 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 $L > R$, the window is empty, and no prefetches for this stream remain outstanding. If stream elements are consumed at the same rate that they are being prefetched, the window enters a steady state. The regular nature of the request sequence eliminates the need for any buffering of requests within the prefetcher. We can simply invalidate the RPT when doing context switches. However, since we are not simulating mutiple programming in this study, we leave quantifying the impact of context switches for future work.

We investigate mechanisms with both fixed and adaptive prefetch distances (as in the designs of Farkas *et al.* [FCJV97] for a different machine model with additional 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, triggering 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 threshhold 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 contention in the cache.

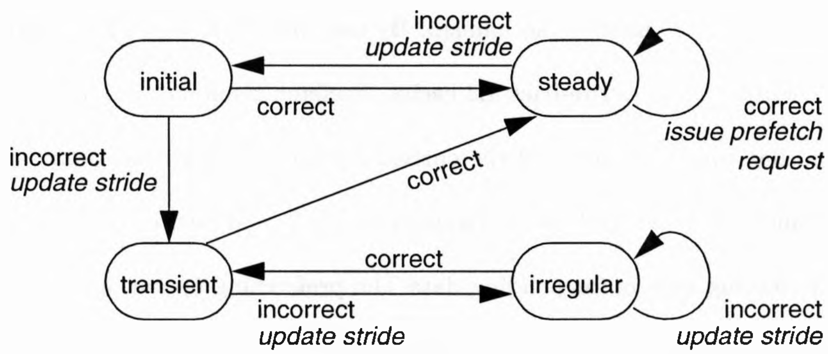

Figure 2: Reference Prediction Table state transitions.

Prefetch misses (data addresses that do not fit the established pattern for that memory instruction) signal the stream ends. The RPT we model can support up to 32 outstanding requests.

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 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.

parameter	cycles (2.5 nsec)	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 timing parameters for Min -45 -800 part [Ram99].

(so no two adjacent banks can be active simultaneously) [Ram99]. Note that we do not model the DRAM write buffers in detail, but all other timing interactions are simulated accurately.

3.3 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 needs to 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.

4 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*, one of the simulation models from the SimpleScalar toolset version 2.0 [BA97]. 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. The simulator we use has been modified to model a Direct Rambus memory system, in addition to modeling to stream detection mechanism described in Section 3.3. We choose the Rambus model for several reasons: it represents the state of the art in affordable, high performance memory systems; its organization and interface are unique, presenting the memory system designer with an interesting set of challenges; and it achieves its performance through a pipelined interface to a highly parallel subsystem, realizing the performance potential of which requires that operations be carefully scheduled and overlapped as much as possible.

4.1 Machine Model

Table 2 lists details of our SimpleScalar configuration. 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. To investigate how the pressure on the memory system affects the opportunity for access reordering mechanisms, we also model a more aggressive superscalar processor with the instruction window size and the issuing width to be 32 and 8 respectively. The two-level, nonblocking cache hierarchy consists of separate, identical first-level instruction and data caches of 64 kilobytes each. The L1 caches are two-way associative and are virtually index and physically tagged; each has 32-byte lines, and a one-cycle hit latency, and can support up to eight outstanding misses. The L2

ifetch queue size	4
issue width	4
load-store queue	8
register update unit	16
L1 instruction cache L1 data cache	nonblocking 64K 2-way 32-byte lines 1 cycle latency virtually indexed, physically tagged write-allocate, write-back 8 MSHRs
L2 unified cache	nonblocking 256K 4-way 64-byte lines 6 cycle latency physically indexed, physically tagged write-allocate, write-back 8 MSHRs
memory	Direct Rambus DRAMs 1/4 CPU clock speed

Table 2: SimpleScalar configuration parameters

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 can support up to eight outstanding misses. The L2 cache capacity is scaled down, since the workloads we use to evaluate our system are relatively small compared to the working sets of real applications (see Table 4 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.

4.2 Memory Models

The memory systems we model consist of eight 64 Mbit Direct Rambus devices on a single channel. We examine two memory organizations: 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.* [HMS+99], since our systems contain eight devices (and afford substantially more parallelism). Their systems fetch individual stream elements instead of performing cache line fills, as we do here: accessing the larger granularity of data can reduce 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.

4.3 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 "1integrate.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 8×8 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 investigate a set of smaller benchmarks with both pointer- and array-based access patterns [ABS94]. `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.

4.4 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.* [HMS+99] to evaluate our design's performance potential. Table 3 lists these kernel access patterns. *daxpy*, *copy*, and *scale* are from the BLAS (Basic Linear Algebra Subroutines) [DDDH90]. *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 (large enough that only one data element resides in each L2 cache line, and even, giving rise to more DRAM bank conflicts).

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

Table 3: Benchmark kernel access patterns.

Benchmark	Cycles (thousands)		Instructions (thousands)	Loads (thousands)	Stores (thousands)	L2 Misses	DTLB Misses	Total Streams	Mean Length
	cache line interleaved	page interleaved							
compress	41209	44079	35684	7366	5989	130325	74	119666	27
gcc	220928	217253	224072	58587	31770	426005	10088	1778862	7
m88ksim	229730	229977	492995	85451	41825	13891	350	3295413	14
li	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
swap	492	462	426	90	43	1917	9	20	2141
swap stride 10	2762	2614	436	90	43	23149	53	63	678

Table 4: Characteristics of each baseline run.

5 Results

Table 4 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

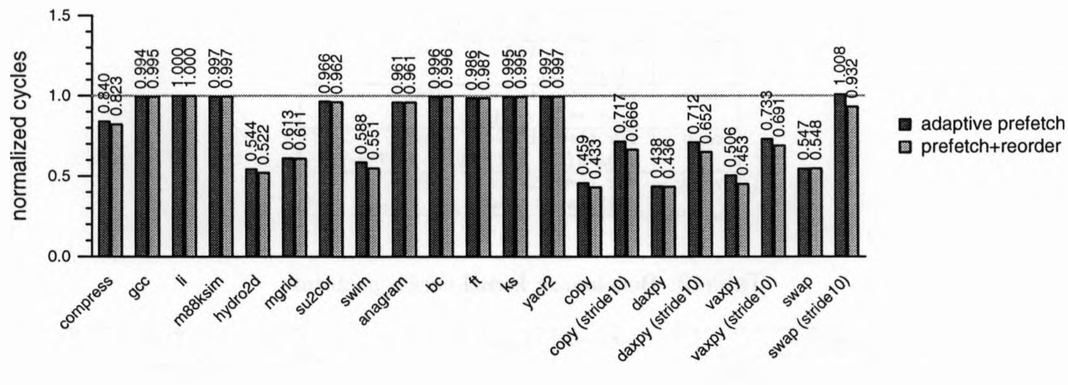

Figure 3: Normalized execution times for a cache line interleaved system and prefetching with an adaptive prefetch distance, with and without dynamic access ordering at the memory controller.

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 its 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

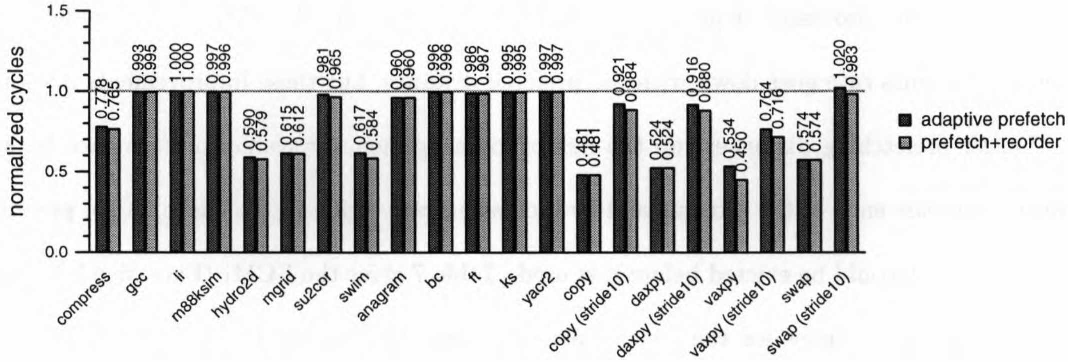

Figure 4: Normalized execution times for a page interleaved system and prefetching with an adaptive prefetch distance, with and without dynamic access ordering at the memory controller.

the case of bc, for instance, fewer than 2% of the memory accesses occur in detectable streams. For gcc, this rate rises to almost 10%, but the average stream length is but seven references, and only about 9% of the memory references make up strided access patterns. li has such a small working set that fewer than 1% of the L2 accesses miss, and the memory 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, since prefetches are issued without the prefetcher's actually recognizing any access patterns) improves performance by over 6%.

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 5 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 6 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. Table 7 gives the ECMP(Excessive L2 Cache Misses per Successful Prefetch) values for the benchmarks we used. We can see that though cache pollution by prefetching exists for most of the benchmarks, none of the benchmarks have more than one L2 cache miss caused by a successful prefetch. In another word, for the prefetching mechanism we used, the speedup from prefetching outperforms contention pollution, so there's overall performance increase. It's of interest to note that, for some of the benchmarks, there's no cache pollution at all. On the other hand, the L2 cache misses plus successful prefetches are even fewer than those without prefetching. This is because in some of the prefetched data, there happen to be unexpected but useful data that turn out to serve future cache requests, thus reducing cache misses.

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 one basic conclusion: 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

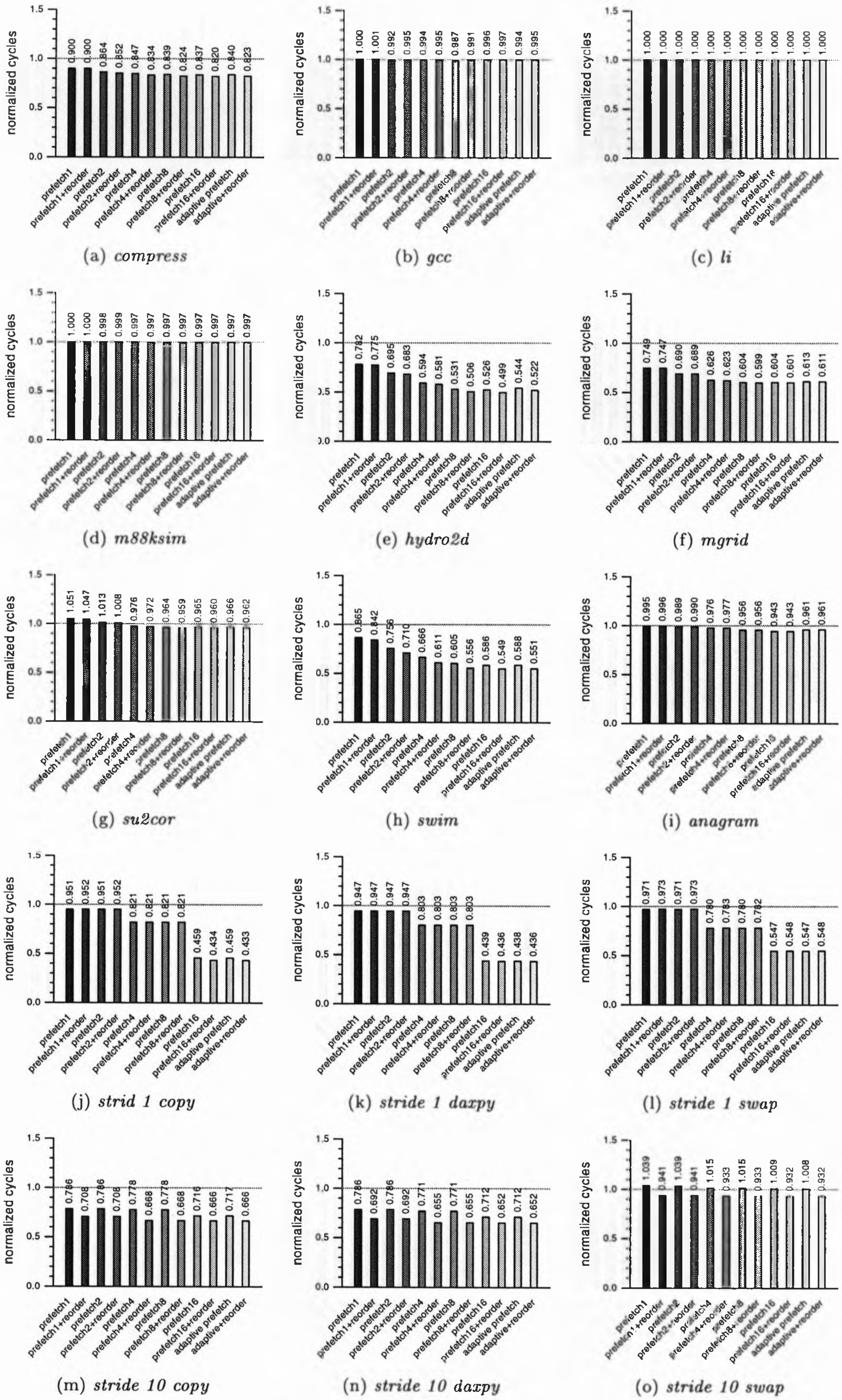

Figure 5: Performance details for selected benchmarks on a cache line interleaved system.

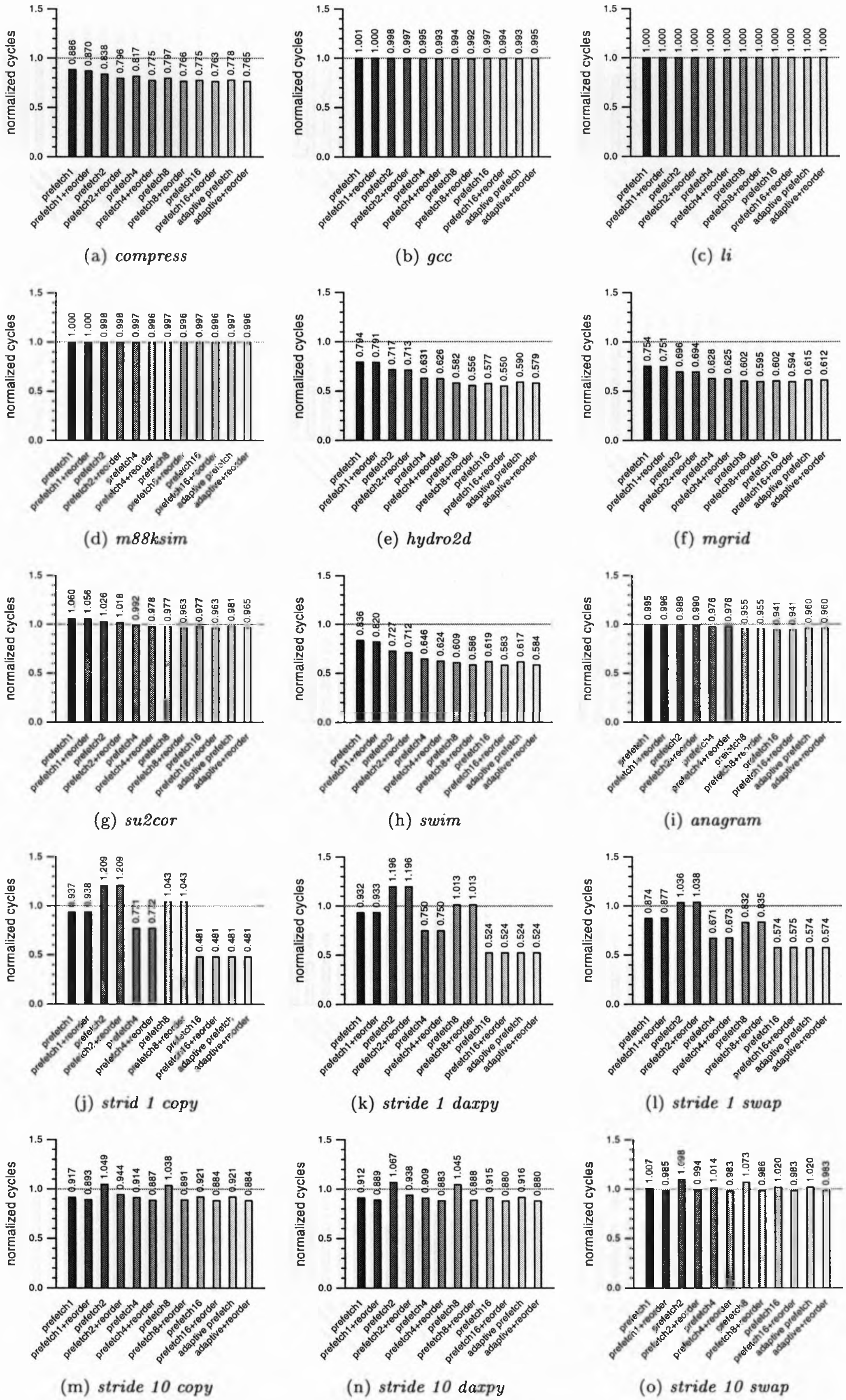

Figure 6: Performance details for selected benchmarks on a page interleaved system.

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	98980	99082	99142	99307	99707	99598	98980	99082	99142	99307	99707	99598
	% in cache	2.4	5.2	11.2	20.6	26.3	24.4	2.6	16.2	16.6	25.0	30.6	28.8
	% in MSHRs	97.1	94.3	88.2	78.7	72.8	74.7	96.9	83.3	82.8	74.3	68.4	70.3
gcc	prefetched	32432	36136	43325	55459	81273	44594	31305	34954	41575	53621	76865	45154
	% in cache	16.7	21.7	32.6	40.3	46.2	36.2	17.1	23.3	34.1	42.9	46.4	35.7
	% in MSHRs	65.0	55.6	44.7	29.5	16.9	38.4	64.6	57.2	43.9	30.2	16.3	37.4
m88ksim	prefetched	9574	9603	9644	9721	9867	9760	0	9603	9643	9720	9867	9760
	% in cache	0.4	1.1	4.5	8.5	16.7	15.1	0.0	1.4	14.4	13.1	21.4	20.0
	% in MSHRs	97.4	96.6	92.9	88.3	78.9	81.3	0.0	96.3	83.0	83.7	74.2	76.4
li	prefetched	1064	1095	1134	1171	1218	1172	1064	1096	1134	1154	1206	1158
	% in cache	0.4	3.0	7.8	14.9	24.4	17.2	1.2	3.9	11.1	15.3	26.3	18.7
	% in MSHRs	95.3	92.5	87.1	78.3	66.1	75.1	95.0	91.8	84.5	77.1	63.4	72.8
hydro2d	prefetched	11402061	11412350	11428218	11433220	11437641	11433632	11402047	11412339	11428177	11433221	11437608	11433629
	% in cache	6.0	11.6	20.9	45.2	53.0	61.8	3.5	8.3	20.6	45.1	50.2	61.6
	% in MSHRs	91.5	85.8	76.5	52.2	44.4	35.6	93.9	89.1	76.8	52.3	47.2	35.8
mgrid	prefetched	4250296	4336082	4464169	4472589	4488109	4484937	4250201	4336047	4464147	4472574	4488092	4484882
	% in cache	13.6	19.7	75.2	83.0	88.1	82.6	15.5	22.3	77.2	86.1	90.4	84.3
	% in MSHRs	86.2	80.1	24.4	16.6	11.2	16.7	84.2	77.4	22.4	13.4	8.8	15.0
su2cor	prefetched	2133760	2157283	2208597	2264564	2353542	2310965	2133770	2157287	2208580	2264585	2353590	2311037
	% in cache	4.3	8.2	21.7	27.3	28.3	28.7	4.3	9.5	21.2	27.6	28.7	28.8
	% in MSHRs	32.6	28.6	14.5	8.8	7.1	7.7	32.6	27.4	14.9	8.5	6.8	7.5
swim	prefetched	7819037	7828283	7839100	7840869	7844758	7841088	7818296	7827452	7839100	7840342	7847694	7844146
	% in cache	7.1	17.3	29.4	51.5	63.2	63.3	9.2	16.0	29.4	52.9	64.4	64.6
	% in MSHRs	61.4	51.3	39.3	17.2	5.3	5.3	59.2	52.5	39.3	15.6	4.1	3.9
anagram	prefetched	3402	3883	4830	5536	6205	5015	3402	3883	4830	5536	6205	5015
	% in cache	1.8	6.1	12.9	27.9	81.0	33.3	1.8	6.3	13.6	28.4	80.2	32.8
	% in MSHRs	91.7	87.8	81.8	66.5	12.0	61.7	91.6	87.6	81.0	66.0	12.9	62.2
copy	prefetched	2666	2666	2667	2666	2671	2669	2667	2666	2668	2666	2672	2670
	% in cache	0.0	0.0	0.1	0.1	74.8	74.5	0.0	0.0	0.2	0.1	92.8	92.5
	% in MSHRs	98.4	98.4	98.3	98.3	23.5	23.8	98.4	98.4	98.1	98.3	5.4	5.7
copy stride 10	prefetched	19770	19770	19795	19794	19797	19795	19770	19770	19795	19794	19797	19795
	% in cache	4.1	4.1	14.8	14.8	61.2	61.8	2.8	4.1	8.8	14.8	52.5	53.2
	% in MSHRs	95.9	95.9	85.2	85.2	38.8	38.2	97.2	95.9	91.2	85.2	47.5	46.8
daxpy	prefetched	2666	2666	2667	2666	2671	2669	2667	2666	2668	2666	2672	2670
	% in cache	0.0	0.0	0.1	0.1	83.8	83.6	0.0	0.0	0.2	0.1	92.8	92.6
	% in MSHRs	98.4	98.4	98.2	98.3	14.5	14.6	98.4	98.4	98.1	98.3	5.4	5.7
daxpy stride 10	prefetched	19770	19770	19795	19794	19797	19795	19770	19770	19795	19794	19797	19795
	% in cache	6.5	6.5	14.4	14.4	61.5	61.9	2.9	6.5	9.4	14.4	54.0	54.6
	% in MSHRs	93.5	93.5	85.6	85.6	38.5	38.1	97.1	93.5	90.6	85.6	46.0	45.4
swap	prefetched	2666	2666	2667	2666	2673	2671	2667	2666	2668	2666	2674	2672
	% in cache	0.0	0.0	0.1	0.1	59.2	59.0	0.0	0.0	0.2	0.1	59.3	59.1
	% in MSHRs	64.3	64.3	64.2	64.2	5.0	5.2	64.3	64.3	64.1	64.2	4.9	5.1
swap stride 10	prefetched	19793	19793	19794	19793	19796	19794	19793	19793	19794	19793	19796	19794
	% in cache	1.6	1.6	7.3	7.3	34.5	34.3	1.6	1.6	6.3	7.3	31.7	31.6
	% in MSHRs	61.3	61.3	55.7	55.7	28.5	28.6	61.4	61.3	56.7	55.7	31.3	31.4

Table 5: 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.

	Percentage of Baseline L2 Cache Misses											
	cache line interleaved						page interleaved					
Benchmark	$d = 1$	$d = 2$	$d = 4$	$d = 8$	$d = 16$	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
gcc	100.6	98.1	99.2	96.1	97.3	98.9	99.7	99.1	97.9	96.6	95.4	99.2
m88ksim	100.2	99.8	97.4	94.6	88.7	89.8	0.0	99.3	90.2	91.2	85.2	86.1
su2cor	141.4	129.8	86.2	69.9	65.9	67.9	141.1	125.7	87.0	69.0	64.7	67.2
li	99.6	98.6	97.2	94.7	90.3	93.3	99.4	98.8	98.2	96.5	91.2	94.6
mgrid	87.6	81.7	27.6	20.0	14.9	20.3	85.8	79.2	25.7	17.0	12.7	18.7
hydro2d	95.5	90.4	81.7	58.9	51.5	43.0	98.1	93.7	82.2	59.0	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
daxpy	100.0	100.0	99.9	99.9	20.5	20.8	100.0	100.0	99.8	99.9	12.0	12.3
daxpy stride 10	95.5	95.5	90.0	90.0	56.8	56.4	98.0	95.5	93.5	90.0	61.9	61.5
swap	100.0	100.0	99.8	99.8	17.4	17.8	100.0	100.0	99.7	99.8	17.3	17.7
swap stride 10	98.6	98.6	93.8	93.8	70.6	70.7	98.6	98.6	94.6	93.8	72.9	73.1

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

Benchmark	Excessive L2 Cache Misses per Successful Prefetch * 100											
	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	0.63	0.65	0.66	0.71	0.91	0.87	0.64	0.65	0.66	0.70	0.92	0.86
gcc	18.36	-17.62	23.95	-0.54	45.11	26.09	33.32	28.20	29.93	34.59	50.08	18.72
li	-1.28	-1.53	1.11	4.92	5.81	4.23	0.10	1.91	7.02	9.37	10.25	9.69
hydro2d	0.01	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.02	0.01
mgrid	0.04	0.08	0.10	0.14	0.23	0.22	0.04	0.08	0.10	0.14	0.23	0.22
su2cor	47.85	47.65	46.10	46.00	47.02	48.23	47.85	47.65	46.10	46.00	47.01	48.21
swim	0.09	0.10	0.12	0.14	0.18	0.15	0.21	0.22	0.24	0.27	0.31	0.29
anagram	8.18	7.87	7.35	8.26	12.52	6.74	8.18	7.86	7.34	8.26	12.53	6.75
copy	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
copy stride 10	0.00	0.00	0.00	0.00	-0.56	-0.53	0.00	0.00	0.00	0.00	-0.46	-0.54
daxpy	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
daxpy stride 10	0.00	0.00	0.00	0.00	-0.37	-0.36	0.01	0.02	0.01	0.02	-0.47	-0.55
swap	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
swap stride 10	0.00	0.00	0.00	0.00	0.06	0.06	0.00	0.00	0.00	0.00	0.09	0.09

Table 7: ECMP(excessive L2 cache misses per successful prefetch) scaled by 100 for some of the benchmarks in Figure 5 and Figure 6.

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. For the more aggressive, 8-issue processor, access ordering gives better speedups since it increases the pressure on the memory system, and correspondingly increases the opportunity for ordering mechanisms to improve performance. For example, `swim` gains a speedup up to 7.1% after reordering in our baseline CPU set up, while it reaches 9.9% for the 8-issue processor. Since the RDRAMs we model use a closed-page precharge policy, the speedup by reordering comes from exploiting the parallelism in the RDRAM systems.

6 Conclusion

One current architectural trend is to migrate more intelligence into the memory system [CHS+99, OCS98, RDK+98] 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 is able to provide the memory controller's access-ordering mechanism with enough choice to make more efficient use of the memory system back end.

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. With respect to the "how much future knowledge = how much performance?" questions that sparked this investigation, in general we find that 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 yields further decreases in execution time ranging from an insignificant margin up to 8%.

Our results are promising:

- for the prefetching schemes we study, the memory interleaving does not materially affect the memory controller's ability to optimize performance;
- larger prefetch distances equate to better performance for nearly all applications;
- our prefetching model delivers consistent performance increases, even in the absence of memory reordering, and in spite of additional cache contention; and
- given sufficient choice with respect to scheduling DRAM accesses, an access-ordering memory controller can deliver significant speedups.

In particular, a memory subsystem that combines incremental prefetching with a reordering DRAM scheduler decreases the execution times of a set of memory-intensive inner loops by over a factor of two (rivaling the performance of Hong *et al*'s compiler-assisted, single-RDRAM dynamic access ordering system [HMS+99]). Moreover, our system delivers comparable benefits for several of the larger scientific applications we simulate (`hydro2d`, `mgrid`, and `swim`), without slowing the performance of non-stream applications.

References

[ABS94] T.M. Austin, S.E. Breach, and G.S. Sohi. Efficient detection of all pointer and array access erors. In *Proceedings of the 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 290–301, June 1994.

[BA97] D. Burger and T.M. Austin. The simplescalar toolset, version 2.0. Technical Report 1342, University of Wisconsin, June 1997.

[BC91] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*, pages 176–186, November 1991.

[BD91] M.E. Benitez and J.W. Davidson. Code generation for streaming: An Access/Execute mechanism. In *Proceedings of the 4th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 132–141, April 1991.

[BGK95] D. Burger, J.R. Goodman, and A. Kagi. The declining effectiveness of dynamic caching for general-purpose microprocessors. Technical Report 1261, University of Wisconsin, February 1995.

[BGK96] D. Burger, J.R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[Bro95] J. Brooks. Single PE optimization techniques for the cray T3D system. In *Proceedings of the 1st European T3D Workshop*, September 1995.

[CEV98] J. Corbal, R. Espasa, and M. 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.

[Che95] T.-F. Chen. Effective programmable prefetch engine for on-chip caches. In *Proceedings of IEEE/ACM 28th International Symposium on Microarchitecture*, pages 237–242, November 1995.

[Chi94] T.-C. Chiueh. Sunder: A programmable hardware prefetch architecture for numerical loops. In *Proceedings of Supercomputing '94*, pages 488–497, November 1994.

[CHK+96] 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, February 1996.

[CHS+99] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, January 1999.

[Cra93] Cray Research, Inc. *CRAY T3D System Architecture Overview*, hr-04033 edition, September 1993.

[DDDH90] J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[DDS95] F. Dahlgren, M. Dubois, and P. Stenstrom. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[Edm95] J.H. 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.

[FCJV97] K.I. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 133–143, June 1997.

[FP91] J.W.C. Fu and J.H. Patel. Data prefetching in multiprocessor vector cache memories. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 54–65, Toronto, Canada, May 1991.

[HMS+99] 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 Annual Symposium on High Performance Computer Architecture*, pages 80–89, January 1999.

[Jou90] N.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, May 1990.

[Kan96] Gerry Kane. *PA-RISC 2.0 Architecture*, 1996.

[Lee93] K. Lee. The NAS860 library user's manual. Technical Report NAS Technical Report RND-93-003, NASA Ames Research Center, March 1993.

[Mat99] B.K. Mathew. Parallel access ordering for SDRAM memories. Technical Report UUCS-99-006, University of Utah Department of Computer Science, June 1999.

[McK96] S.A. McKee, et al. Design and evaluation of dynamic access ordering hardware. In *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.

[MLG92] T.C. Mowry, M. S. Lam, , and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[MNS92] L. Meadows, S. Nakamoto, and V. Schuster. A vectorizing software pipelining compiler for LIW and superscalar architectures. In *RISC'92*, pages 331–343, 1992.

[MW95] S.A. McKee and Wm.A. Wulf. Access ordering and memory-conscious cache utilization. In *Proceedings of the First Annual Symposium on High Performance Computer Architecture*, pages 253–262, January 1995.

[OCS98] M. Oskin, F.T. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 192–203, June 1998.

[PK94] S. 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, May 1994.

[PK95] S. Palacharla and R.E. Kessler. Code restructuring to exploit page mode and read-ahead features of the cray T3D. Technical Report Internal Report, Cray Research, February 1995.

[Ram99] Rambus Inc. *Direct RDRAM 64/72-Mbit Data Sheet*, 1999. http://www.rambus.com/developer/downloads/.

[RDK+98] S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A. Lopez-Lagunas, P.R. Mattson, and D.J. Owens. A bandwidth-efficient architecture for media processing. In *Proceedings of IEEE/ACM 31st International Symposium on Microarchitecture*, December 1998.

[Sco96] Steven Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[Skl92] I. Sklenar. Prefetch unit for vector operation on scalar computers. *Computer Architecture News*, 20(4):31–37, September 1992.