Access Order and Memory-Conscious Cache Utilization

Sally A. McKee and Wm. A. Wulf

Computer Science Report No. CS-94-10
March 1, 1994

Access Ordering and Memory-Conscious Cache Utilization

Sally A. McKee and Wm. A. Wulf
Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22901
{mckee | wulf}@cs.virginia.edu

Abstract

As processor speeds increase relative to memory speeds, memory bandwidth is rapidly becoming the limiting performance factor for many applications. Several approaches to bridging this performance gap have been suggested. This paper examines one approach, access ordering, and pushes its limits to determine bounds on memory performance. We present several access-ordering schemes, and compare their performance, developing analytic models and partially validating these with benchmark timings on the Intel i860XR.

1. Introduction

Processor speeds are increasing much faster than memory speeds, thus memory bandwidth is rapidly becoming the limiting performance factor for many applications, particularly scientific computations. Proposed solutions range from software prefetching [4, 16, 27] and iteration space tiling [5, 8, 9, 18, 32, 38], to address transformations [12, 13], unusual memory systems [3, 10, 33, 36], and prefetching or non-blocking caches [1, 6, 34]. Here we take one technique, *access ordering*, and examine it in depth by analyzing the performance of five different access-ordering schemes. Our techniques for increasing memory bandwidth are not new, but our goal here is to determine the upper bounds on their performance in order to aid architects and compiler designers in making good choices among them.

Memory components are commonly assumed to require about the same time to access any random location, but this no longer applies to modern memory devices. For instance, nearly all current DRAMs implement a form of page-mode operation [30]. Other common devices offer similar features (nibble-mode, static column mode, or a small amount of SRAM cache on chip) or exhibit novel organizations (such as Rambus [31], Ramlink, and the new synchronous DRAM designs [14]). The order of requests strongly affects the performance of all these components.

A comprehensive, successful solution to the memory bandwidth problem must exploit the richness of the full memory hierarchy, both its architecture and its component characteristics. One way to do this is via access ordering, any technique for changing the order of memory requests to increase bandwidth. For applications that perform vector-like memory accesses, for instance, bandwidth can be increased by reordering the requests to take advantage of device properties such as fast-page mode.[1]

In order to analyze the performance of a representative subset of access ordering techniques, we have selected five implementation schemes:

- *naive ordering*, or using caching loads to access vector elements in the natural order of the computation;
- *streaming* elements using non-caching loads, and then copying them to cache;
- *block-prefetching* vector elements to cache (before entering the inner loop);
- *static access ordering* at the register level, using non-caching loads; and
- hardware-assisted *dynamic access ordering*.

The first, naive ordering, provides a basis for comparing the performance improvements of the other schemes. None of the techniques requires heroic compiler technology: the compiler need only detect streams, as in Benitez and Davidson's algorithm [2]. Dynamic access ordering requires a small amount of special-purpose hardware [25], and both static and dynamic access ordering depend on the availability of non-caching load instructions. Although rare, these instructions are available in some commercial processors, such as the Convex C-1 [37] and Intel i860 [15]. Other architectures, such as the DEC Alpha [7], provide a means of specifying some portions of memory as

1. These devices behave as if implemented with a single on-chip cache line, or *page*. A memory access falling outside the address range of the current DRAM page forces a new page to be accessed. The overhead time required to do this makes servicing such a request significantly slower than one that hits the current page.

non-cacheable (but the ability to set such things is not generally available at user level).

Our investigation targets one aspect of cache performance that has been overlooked: the time to load a vector, independent of whether data is reused. We therefore focus not on the cache miss rates, but on memory access costs, and we are concerned only with the parts of programs affected by vector access-time, the inner loops.

Although we suspect that the performance of these schemes (at least for unit-stride vectors) will be ranked as $naive < stream < prefetch < sao < dao$, we wish to verify these relationships, and to quantify the differences in performance. To this end, we develop general analytic models for each scheme. We then show what the actual performance differences between schemes is for one particular set of real machine parameters, those of the i860XR. Due to limitations of available hardware, only three of the techniques could be implemented and tested, but the results of these experiments give us at least a partial validation of our models in the context of a real system.

2. Performance models

In this section, we develop analytic performance models for a memory system composed of page-mode DRAMS. In order to derive upper bounds on performance, we assume that there are no cache conflicts, pages are infinitely long, and vectors are aligned to cache-line boundaries.

In order to avoid notational clutter, we omit the ceiling functions in our formulas. We also assume that the amount of data transferred by each bus transaction (or by a caching reference) is the size of one vector element. These formulas are independent of the number of banks in an interleaved memory system, since we assume that the page-miss latencies for separate banks can be overlapped. Let:

- s be the vector stride;
- b be the number of vector elements in a block, or submatrix, of data that we wish to load;
- m be the size of the memory (in terms of the size of vector elements) that must be fetched to load the block (for caching loads, $m = b \times min(s, l)$, and for non-caching loads, $m = b$); and
- l be the number of vector elements that fit in a cache line.

We add a few definitions to characterize memory access costs and the amount of useful data in a cache line. Let:

- t_{crd} be the cost of reads that hit in the cache;
- t_{cwr} be the cost of writes that hit in the cache;
- t_{miss} be the DRAM page-miss cost, in cycles;
- t_{hit} be the DRAM page-hit cost, in cycles; and
- d_l be the density of vector elements in a cache line ($d_l = 1/min(s, l)$).

Section 2.1 through Section 2.5 introduce each scheme and present the corresponding performance model. Comparative results for the models are given in Section 3.

2.1 Naive accessing

As a baseline for comparison, we wish to determine performance for a computation in which no attempt is made to tailor access order to memory system parameters. We calculate the average number of cycles used by caching instructions to load vector elements in the natural order of the computation. We assume that for each cache-line fill, the first access incurs the DRAM page-miss overhead. The DRAM page status may have been flushed by accesses to other data in between cache line fills for a particular vector. Each remaining access in the line hits the current page. Unfortunately, when $s > 1$, some of these accesses fetch data that will not be used.

Assuming that the cost of reading from cache is subsumed by the cost of performing a cache-line fill, the average per-element cost of using caching loads in this manner is the number of cycles to fill a line, divided by the amount of useful data contained therein:

$$T_{naive} = \frac{t_{miss} + (l-1)\, t_{hit}}{l d_l} \qquad (1)$$

This formula describes effective bandwidth whenever vectors are accessed in the computation's natural order, even when loop-unrolling is applied. Note that the effectiveness of naive ordering decreases rapidly as vector stride increases.

2.2 Block prefetching

Blocking or *tiling* changes a computation so that subblocks of data are repeatedly manipulated [8, 9, 18, 32, 38]. A familiar example is multiplication of $n \times n$ matrices stored in row-major order:

```
for i = 1 to n do
  for j = 1 to n do
    load A[i,j] into register r
    for k = 1 to n do
      C[i,k] = C[i,k] + r * B[j,k];
```

Unless the cache is large enough to hold at least one of the matrices, the elements of B in the inner loop will be evicted by the time they are reused on the next iteration of the outer loop. Likewise, whether the row of C remains resident until the next iteration of the j loop depends on the size of the cache. If instead the code is modified to act on a $b \times b$ submatrix of B, this data will be reused b times each time it is loaded. The blocking factor b is chosen so that the submatrix and a corresponding portion of a row of C fit in cache:

```
for j_block = 1 to n by b do
  for k_block = 1 to n by b do
    for i = 1 to n do
      for j = j_block to min(j_block+b-1, n) do
        load A[i,j] into register r
        for k = k_block to min(k_block+b-1, n) do
          C[i,k] = C[i,k] + r * B[j,k];
```

This optimization can be applied at many levels of the memory hierarchy, including registers, cache, the TLB, and virtual memory. Blocking reduces average access latency by reusing data at a faster level of the hierarchy.

We can also apply the notion of blocking to caching vector-accesses: to minimize the total DRAM page-miss overhead, vector elements can be prefetched into the cache in chunks. When the processor uses the vector block within an inner loop, the data should still be cache-resident. Even though we are not specifically concerned with data reuse, there may be other memory references between when the data is fetched and when it is referenced by the processor, thus we must still consider issues of interference. Determining optimal block size in the presence of cache conflicts may be difficult, but algorithms to address this problem have been presented elsewhere [18, 35]. The ideas presented here can be incorporated into those algorithms to yield even better memory performance.

Ideally, we would like to use non-blocking loads. Even if these are not available, the optimization may still be worthwhile, provided that the cost of DRAM page misses is sufficiently high in relation to page hits and cache accesses: the overhead of preloading the data and retrieving it from cache is offset by the number of fast accesses that can be performed. Note that the processor need not explicitly read *all* data values in order to preload the vector: touching one element per line will bring the entire line into cache (of course, the cache controller must still fetch each word from memory). Even fewer instructions are required on architectures that prefetch larger blocks, such as the Alpha with its 512-byte FETCH [7].

The mean cost of block-prefetching a vector element to cache and reading it from there during the computation is:

$$T_{prefetch} = \frac{t_{miss} + (m-1)\,t_{hit}}{b} + t_{crd} \qquad (2)$$

For unit-stride vectors and large blocks, the first term approaches the minimum t_{hit} cycles per vector element.

2.3 Streaming into local memory

Copying improves memory system performance by moving non-contiguous data to be reused into a contiguous area, much like a vector-processor *gather* operation. For instance, in parallelizing a Fast Fourier Transform, Gannon and Jalby use copying to generate the transpose of a matrix, giving both row-wise and column-wise array accesses the same locality of reference [9]. Lam et. al. investigate blocking in conjunction with copying in order to eliminate *self-interference*, or cache misses caused by more than one element of a given vector mapping to the same location [18]. This optimization also reduces TLB misses and increases the number of data elements that will fit in cache when the vector being copied is of non-unit stride.

Copying essentially attempts to explicitly manage the cache as a fast, local memory. By exploiting memory properties, this technique may also benefit single-use vectors and those that do not remain in cache between uses. For example, when accessing non-unit stride vectors, *streaming* data via non-caching loads and then writing it to cache avoids fetching extraneous data, and thus may yield better performance than the previous, block-prefetching scheme. Since each read of a vector element incurs a read from memory as well as a cache write and read, streaming will provide the most benefit when cache accesses and DRAM page hits cost much less than page misses. This optimization may also prove valuable for caching unit-stride vectors if page misses are relatively expensive and block prefetching is inefficient due to hardware limitations.

Assuming a write-back cache, the cost per element copied includes the costs of reading the data using non-caching loads, writing it to the cache, and reading it back from cache later:

$$T_{copy} = \frac{t_{miss} + (b-1)\,t_{hit}}{b} + (t_{cwr} + t_{crd}) \qquad (3)$$

Note that the cost of initially allocating the local memory is not reflected in this formula. For unit-stride vectors, the T_{copy} differs from $T_{prefetch}$ only by the time to write the vector elements to cache. On some architectures, it may be possible to overlap the writes to cache with non-caching loads, in which case t_{cwr} drops out of the equation.

2.4 Static access ordering

Moyer introduces static access ordering to maximize bandwidth for non-caching register loads, and derives compile-time access-ordering algorithms relative to a precise analytic model of memory systems [29]. This approach unrolls loops and orders non-caching memory operations to exploit architectural and device features of the target memory system. As an example, consider the familiar dot product:

```
sum = 0
for i = 1 to n do
  sum = sum + A[i] * B[i]
```

Naive scalar code for this example involves fetching an alternating sequence of A's and B's: $\langle A_1, B_1, A_2, B_2, \ldots \rangle$. Using a single memory bank with page-mode DRAMS and

non-caching loads, the alternating sequence may flush the page status on each request, negating the potential gains from this type of memory. Unrolling once and accessing the data as $\langle A_1,A_2,B_1,B_2,\ldots\rangle$ improves the performance of this particular loop; unrolling it more would be even better.

Using this approach, the average per-element cost for fetching a block of the vector is:

$$T_{sao} = \frac{t_{miss} + (b-1)\,t_{hit}}{b} \qquad (4)$$

This formula assumes that the first access to each block incurs the DRAM page-miss overhead. Subsequent accesses in that block hit the current page, and thus happen faster. This allows us to amortize the overhead of the page miss over as many accesses as there are registers available to hold data. The Intel i960MM has a local register cache with 240 entries that could be used to store vector elements for this scheme [17], but most processors have far fewer registers at their disposal. Assuming $b = 8$ for double-word vectors would probably be optimistic for most computations and current architectures. Note that for unit-stride vectors, T_{sao} differs from $T_{prefetch}$ only by the last term in the latter, which is constant for a given architecture.

2.5 Dynamic access ordering

Performing register-level access ordering at compile time can significantly improve effective memory bandwidth, but the extent to which the optimization can be applied is limited by the number of available registers and by the lack of alignment information generally available only at run-time. Cache-level access ordering by block prefetching or streaming alleviates register pressure, but these are still compile-time approaches, thus they also suffer from the lack of data placement and alignment information. As with other forms of cache blocking, the effectiveness of these techniques depends on the amount of cache interference. For good performance, block size should be adapted to cache and computation parameters. Finally, caching vectors inevitably displaces scalar data that would otherwise remain resident.

These limitations exist in part because the ordering is being done at compile time, and in part because of the program's demands on registers and cache. A system that reorders accesses at runtime and provides separate buffer space can reap the benefits of access ordering without these disadvantages, at the expense of adding a relatively small amount of special-purpose hardware.

One such scheme is depicted in Figure 1 [23, 25]. In this organization, memory is interfaced to the processor through a controller (or *Memory Scheduling Unit*) that includes logic to issue memory requests and logic to determine the order of requests during streaming computations. A set of control registers allow the processor to specify stream parameters (base address, stride, length, and data size), and a set of high-speed buffers holds stream operands. The stream buffers are implemented logically as a set of FIFOs, with each stream assigned to one FIFO.

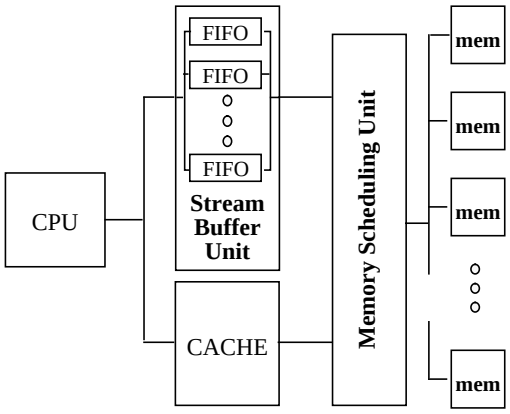


Figure 1 Dynamic access ordering system

Detailed performance models and simulation results for this organization are presented elsewhere [23, 24, 25]. What follows is an approximate model to determine memory performance for a single vector of a computation. Accurate prediction requires knowledge of the entire computation, since performance for each stream depends on the nature and number of other streams.

Let f be the FIFO depth in vector elements, and let F represent the number of elements that can be fetched in succession. FIFO depth here is analogous to the block size of previous examples. If we assume that the FIFO is initially empty, the mean time to load a vector element is:

$$T_{dao} = \frac{t_{miss} + (F-1)\,t_{hit}}{F} \qquad (5)$$

Obviously as F grows, this tends to t_{hit}, the minimum time to perform a DRAM access. If the vector is completely fetched before the processor starts consuming data, then $F = f$, but if the processor consumes data from the FIFO while the memory system is filling it, F becomes more complex. Let v represent the number of vectors in the computation. If the processor accesses the FIFOs (in round robin order) at the same rate as the memory system, then while the memory is filling a FIFO of depth f, the processor will consume f/v more data elements from that stream, freeing space in the FIFO. While the memory supplies f/v more elements, the processor removes f/v^2, and so on. The total number of accesses required to fill the FIFO can be represented as a series that, in the limit, converges to:

$$F = f\left(1 + \frac{1}{v} + \left(\frac{1}{v}\right)^2 + \left(\frac{1}{v}\right)^3 + \ldots\right) = \frac{f}{1-\frac{1}{v}} = \frac{fv}{v-1} \qquad (6)$$

3. Examples

For purposes of comparison, we wish to focus on a single platform in both the analytic and experimental portions of this work. The Intel i860XR was selected because it provides the non-caching load instructions necessary for our experimental measures. Unless otherwise specified, the data presented here is generated using parameters from that system:

- vector elements are double words,
- cache lines are 32 bytes, or 4 double words ($l = 4$),
- pipelined loads fetch one double word, and DRAM page misses and page hits take 10 and 2 cycles, respectively,
- caching loads and stores that hit the cache can transfer a quad word ($c = 2$) in one cycle ($t_{crd} = t_{cwr} = 1$),
- the write-back cache holds 8K bytes, and is two-way set associative with pseudo-random replacement, and
- DRAM pages are 4K bytes.

3.1 Analytic results

Figure 2 illustrates the comparative performance of the five access schemes described in Section 2. Although blocking is not relevant to accessing vector elements in their natural order — all blocks are the size of a cache line — we include that line for reference. The dynamic access ordering results given here are for a computation involving three vector operands (such as the first and fifth Livermore Loops, *hydro fragment* and *tri-diagonal elimination* [26]). Average cycles per element will be slightly lower for computations involving fewer vectors and slightly higher for computations requiring more. Note that for dynamic access ordering, block size corresponds to FIFO depth.

Figure 2(a) shows the average cycles per element to fetch a unit stride vector using each of our schemes. The four schemes that consider access order consistently perform better than the naive, natural-order access pattern. Note that the *stream*, *prefetch*, and *sao* curves are a constant distance apart: they differ only by the cost of the cache accesses involved in each. The curve for *sao* may be a bit misleading, since most architectures provide too few registers for static access ordering to be used with block sizes greater than 8. These curves are grayed slightly in Figure 2 to call attention to this fact. Nonetheless, we depict the theoretical performance attainable if such an implementation were possible.

To emphasize the impact that order has on effective bandwidth, Figure 2(b) illustrates the corresponding percentages of peak system bandwidth delivered by each of the ordering schemes. Naive ordering uses only 50% of the available bandwidth. Streaming and block-prefetching can deliver over 65% and 78%, respectively, for block sizes of 128 or more elements. Using blocks of size 8, static access ordering achieves 67% of the total system bandwidth.This scheme could deliver 80% of peak with 16 registers to hold stream operands. Of the five schemes, dynamic access ordering makes most efficient use of the memory system, delivering over 96% of peak bandwidth for a FIFO depth of only 32 elements. Performance converges on 100% for FIFOs that are over 128 elements deep.

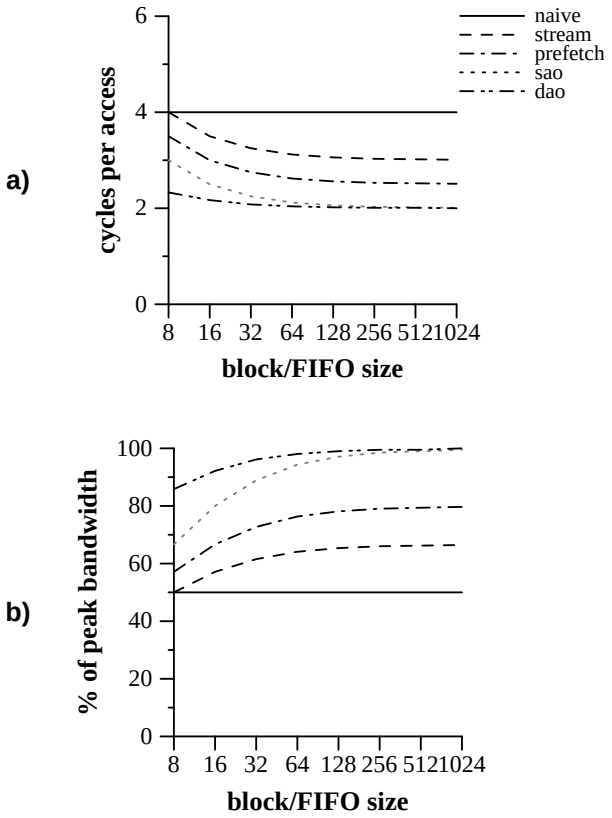
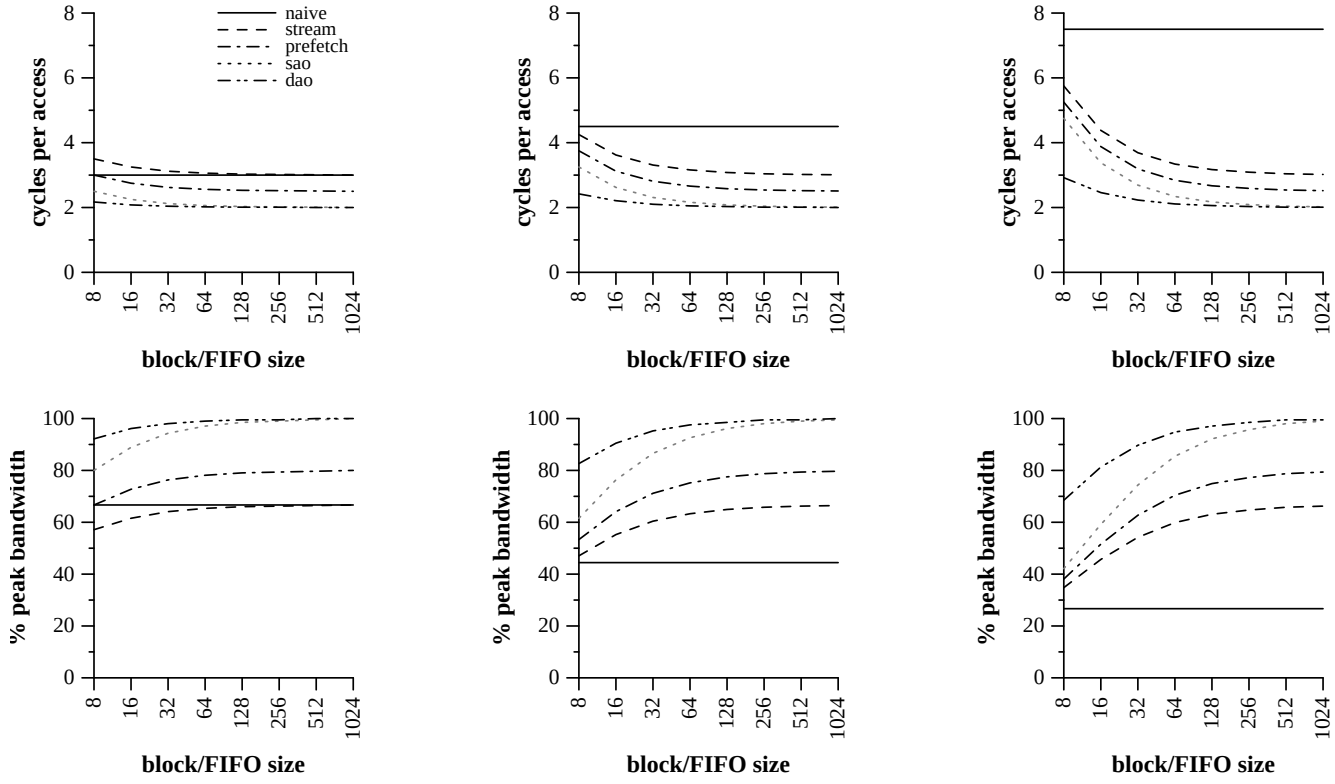


Figure 2 Vector load performance

As DRAM page misses become comparatively more expensive, accessing data in the natural order delivers less and less bandwidth, but the performance of the other four schemes stays relatively constant for block sizes of 64 or more. Figure 3 depicts both average time to access a vector element and percentage of peak bandwidth attained as DRAM access costs change.

Figure 3(a) shows memory performance when page hits occur three times as fast as page misses, a ratio representative of current technology. Static access ordering, dynamic access ordering, and block prefetching all out-perform the naive ordering for block sizes greater than 8. Dynamic access ordering delivers data at nearly the



a) miss = 6, hit = 2 **b) miss = 12, hit = 2** **c) miss = 24, hit = 2**

Figure 3 Vector load performance for increasing page miss/hit cost ratios

maximum rate for FIFO depths of 32 or more. Streaming only makes sense for these parameters if it can be done in large blocks, since the extra cache write and read are expensive relative to memory access costs.

When DRAM page misses cost six times as much as page hits, naive ordering delivers less than half of the available bandwidth, as depicted in Figure 3(b). In this case, all four of the other schemes yield better performance for all block sizes. At a page-miss/page-hit cost ratio of twelve, shown in Figure 3(c), the differences are even more striking: naive ordering barely uses one-quarter of the peak system bandwidth. In contrast, at a block size of only 64, streaming, block-prefetching, and dynamic access ordering deliver 60%, 70%, and 95% of peak, respectively.

Note that the miss/hit ratio is likely to increase as a result of a reduction in page-hit time, rather than an increase in page-miss time, hence the cycle time for Figure 3(c) is much less than that in Figure 3(a). The curves in Figure 3(c) represent a percentage of a *much* larger bandwidth. To illustrate this, we held page-miss costs constant, and reduced page-hit times proportionately to create Figure 4, which depicts how the best and worst performances for the system in Figure 3(a) compares to those for Figure 3(c).

Figure 5 illustrates the results of using each of our schemes for non-unit stride vectors. As stride increases, the performance of naive ordering degrades sharply — from 50% of available bandwidth at stride one to 25% at stride two, 16.7% at stride three, and 12.5% at strides of four or more. Cache performance is constant for strides greater than the line size, since for such strides only one element resides in each line. Like naive ordering, block-prefetching suffers from fetching extraneous data. Prefetching amortizes page-miss overheads over a greater number of accesses, thus it yields better performance than accessing data in the natural order.

The cost of performing the extra cache write and read limit *stream*'s performance to 50% of available bandwidth. For non-unit strides, however, streaming is always preferable to block-prefetching. Again, dynamic access ordering exploits nearly the full bandwidth for FIFOs of depth 64 or more. Note that the percentage of bandwidth delivered for any of the schemes using non-caching loads is independent of vector stride: performance begins to degrade only when vector stride becomes large with respect to DRAM page size.

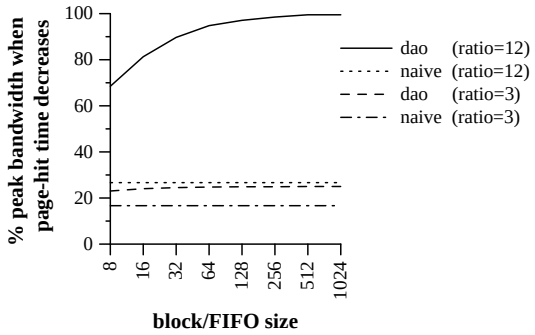


Figure 4 Scaled vector load performance for decreasing page-hit costs

3.2 Empirical results

In order to at least partially validate our formulas, we have implemented three of the accessing schemes on an Intel i860XR processor: naive ordering, streaming, and static access ordering. The i860XR cache controller prevents us from implementing block-prefetching as described in Section 2.2. On this processor, each successive cache-line fill incurs a seven cycle delay [28]. This is long enough for the memory controller to transition to its idle state, causing the next memory access to take the same time as a DRAM page-miss, regardless of whether or not it lies in the same page as the previous access. The i860XR supports a dual-instruction mode, however, allowing writes to cache to be overlapped with pipelined, non-caching loads. Under these circumstances, block-prefetching unit-stride vectors uses the same number of instruction cycles as streaming. We may therefore take the measured streaming performance to be some indication of the performance we could expect from an implementation of block-prefetching.

Hardware to support dynamic access ordering is under development, but is not yet available for gathering general empirical data. We currently have an initial, partial implementation that has shallow FIFOs, but runs at-speed; a full implementation is currently being fabricated. We have thus far generated only a single data point, and it appears to agree with our analysis and simulation results. Obviously, it is still too early to make any definitive claims about the hardware's performance. Based on the results of Section 3.1, though, we expect an efficient implementation of dynamic ordering to perform asymptotically about the same as static access ordering; this is part of the motivation for investigating the performance of static ordering for unrealistically large block sizes.

The measured results presented here represent the performance of three routines to load doubleword vectors:

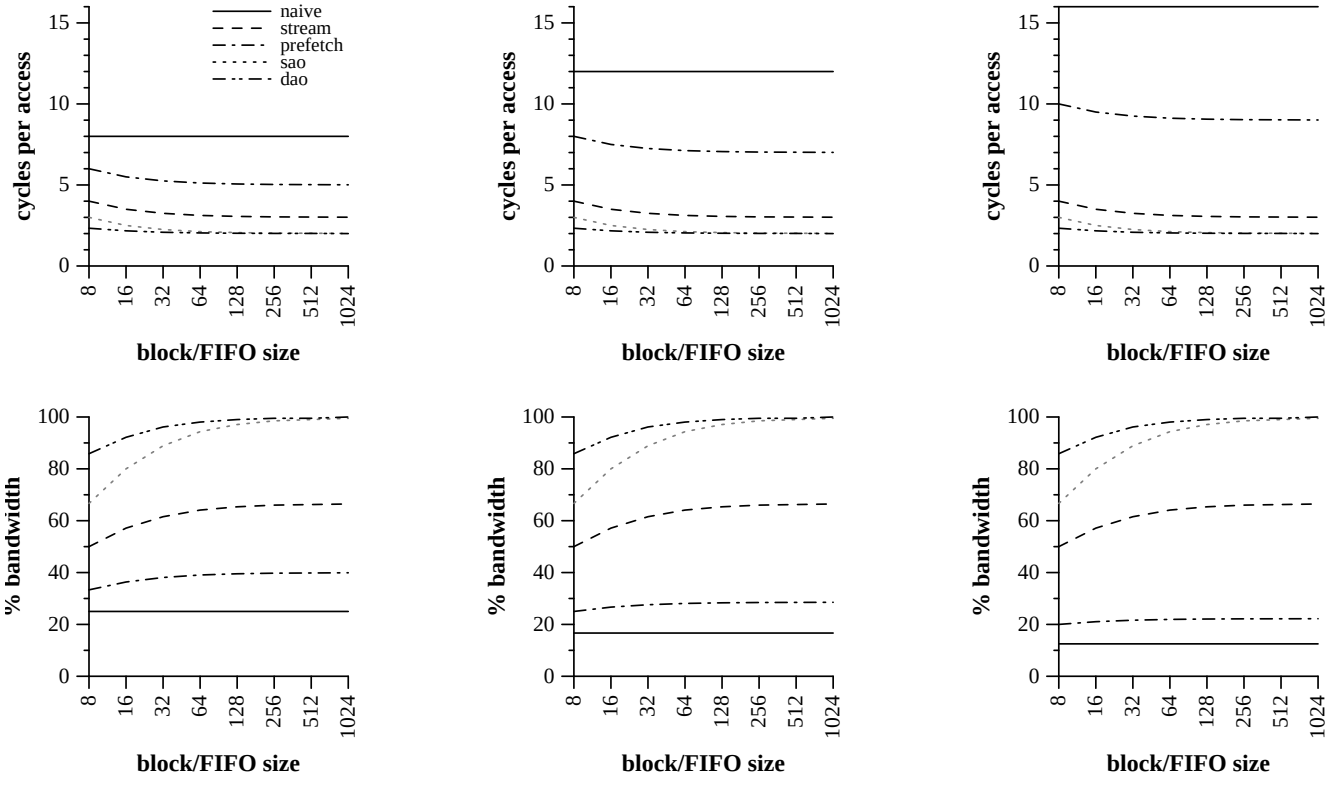


Figure 5 Vector load performance for increasing stride

- *naive()* uses caching loads (*fld.q* for stride one, *fld.d* for others) to bring the vector into cache.
- *sao()* uses non-caching loads (*pfld.d*) to read the vector. The routine reuses registers in order to simulate large block sizes.
- *stream()* overlaps non-caching instructions loads with quadword stores to local (cache-resident) memory, reloading the data to registers after the entire block has been written to cache.

Recall that here we are only concerned with determining *bounds* on memory system performance, hence these routines are designed to put maximum stress on the memory by assuming that arithmetic computation is infinitely fast. We've examined the effects of these techniques on other programs: the results are similar, but space limitations prevent us from discussing them here. The cache was flushed before each call, and each routine was timed 100 times using dclock(). The mean of these timings is presented. All vectors used here are 1024 doubleword elements in length. The time to allocate local memory is not included in the streaming results. If the local memory will be reused, this overhead will be amortized over many vector accesses that hit the cache. If not, the allocation cost must be taken into account when deciding whether to apply the optimization.

Figure 6 presents vector-load performance for vectors of various strides. Note that the analytic results for streaming were generated using Eq. 3 from Section 2.3, modified to account for the overlapping of writes with non-caching reads. In all cases, measured performance approaches theoretical performance for large block sizes. Differences for smaller blocks can be attributed to overhead costs for each subroutine and to page misses caused by crossing DRAM page boundaries (our models do not account for such "compulsory" misses).

Note that the performance of *stream* and *sao* is relatively independent of vector stride, whereas the average cost per access of naive ordering rises steadily as the stride grows up to the cache line size. For these machine parameters, static access ordering always beats naive ordering for blocks larger than the cache-line size. The point at which streaming yields better memory performance than naive caching depends on stride and the amount of overhead in the subroutine call. If the code to perform streaming were generated by the compiler, or if function in-lining were used to mitigate the costs of the subroutine call, streaming might become profitable for even smaller block sizes.

4. Related work

There is a large body of research characterizing and evaluating the memory performance of scientific codes. Most of this research focuses on:

a) hiding or tolerating memory latency,
b) decreasing the number of cache misses, or
c) avoiding bank conflicts in an interleaved memory system.

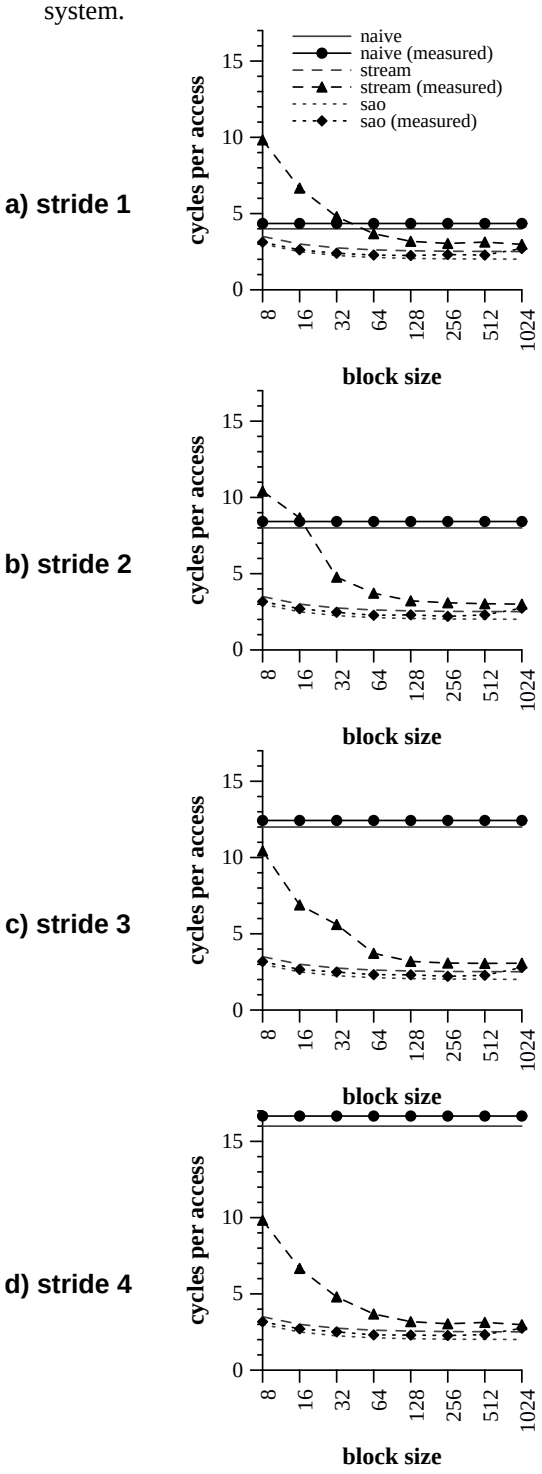


Figure 6 Vector load performance for the i860XR

Prefetching and nonblocking caches can be used to overlap memory accesses with computation, or to overlap the latencies of more than one access [1, 4, 11, 16, 27, 34]. These methods can improve processor performance, but techniques that simply mask latency do nothing to increase effective bandwidth. Such techniques are still useful, but they will be most effective when combined with complementary technology to exploit memory component capabilities.

Modifying the computation to increase the reuse of cached data can improve performance dramatically [8, 9, 5, 32, 38, 18, 35]. These studies assume a uniform memory access cost, thus they don't address minimizing the time to load vector data into cache. These techniques will also deliver better performance when integrated with methods to make more efficient use of memory resources.

Lam et. al. develop a model of data conflicts and demonstrate that the amount of cache interference is highly dependent on block size and vector stride, with large variations in performance for matrices of different sizes [18]. For best results, block size for a computation must be tailored to matrix size and cache parameters, and efficient blocked access patterns tend to use only a relatively small portion of the cache. This may limit the applicability of cache-based access ordering techniques discussed here. Block-size limitations can be circumvented by providing a separate buffer space for vector operands.

Lee develops a subroutine library, called NASPACK, to mimic Cray instructions on the Intel i860XR [20]. Included are routines for streaming vector elements into cache. Data is read in blocks via non-caching load instructions, and is written into pre-allocated local memory. Meadows, et. al., describe a similar scheme used by the PGI i860 compiler [22], and a Loshin and Budge give a general description of the technique [21]. The Loshin and Budge article is intended only to introduce the concept of *Memory Hierarchy Management* (MHM) by the compiler, and the NASA and PGI studies address streaming in conjunction with other operations. Thus these reports neither develop a general performance model nor present measured timing results specific to this optimization.

Copying incurs an overhead cost proportional to the amount of data being copied, but the benefits often outweigh the cost [18], and Temam et. al. present a compile-time technique for determining when copying is advantageous [35]. Using caching loads to create the copy can cause subtle problems with self-interference, though. As new data from the original vector is loaded, it may evict cache lines holding previously copied data. Explicitly managing the cache becomes easier when a cache bypass mechanism is available. Note that coherency issues must be addressed when vectors are shared.

Research on blocking and copying has focused primarily on improving performance for data that is reused, the traditional assumption being that there is no advantage to applying these transformations to data that is only used once. In contrast, reports on the NASPACK routines and the PGI compiler suggest that by exploiting memory properties, these techniques may also benefit single-use vectors and those that do not remain in cache between uses. Our results support these conclusions.

Several schemes for avoiding bank contention, either by address transformations, skewing, or prime memory systems, have been published [3, 10, 12, 13, 33]; these, too, are complementary to the techniques for improving bandwidth that we analyze here.

Moyer [28] and Lee [19] investigate the floating point and memory performance of the i860XR. Our examples for this architecture agree largely with their findings.

5. Conclusions

As processors become faster, memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to vector-like algorithms. Here we have examined the time to load a vector using five different access-ordering schemes, putting maximum stress on the memory system in order to determine performance bounds. Four of these schemes are purely software techniques; one requires the addition of a modest amount of supporting hardware. The more efficient schemes exploit the ability to bypass the cache.

A comprehensive, successful solution to the memory bandwidth problem must exploit the richness of the *full* memory hierarchy: it cannot be treated as though it were uniform access-time RAM. This requires not only finding ways to improve cache performance, but providing alternatives for computations that don't exhibit the properties necessary to make caching effective.

This knowledge should guide processor designs and operating system implementations. To get good memory performance, the user needs more control over what gets cached and how, and mechanisms to take advantage of memory component capabilities should be readily available. Unfortunately, this is not the case for most current microprocessor systems. For cases where such mechanisms are available, we have demonstrated how several straightforward techniques can improve bandwidth dramatically. These schemes require no heroic compiler technology, and are complementary to other common code optimizations. Our results indicate that access ordering can deliver nearly the full memory system bandwidth.

Acknowledgments

This work was supported in part by a grant from Intel Supercomputer Division and by NSF grant MIP-9307626. Access to the i860XR was provided by the University of Tennessee's Joint Institute for Computer Science. Norman Ramsey and Greg Lindahl gave valuable comments on early drafts of this paper. The memory systems group at UVa includes Assaji Aluwihare, Jim Aylor, Alan Batson, Bob Klenke, Trevor Landon, Chris Oliver, Bob Ross, Max Salinas, Chenxi Wang, Dee Weikle, and Kenneth Wright. Former group members Charlie Hitchcock, Sean McGee, Steve Moyer, and Andy Schwab also made invaluable contributions during their tenure. Kim Gregg's administrative miracles keep everything running smoothly.

References

[1] Baer, J.-L., and Chen, T.-F., "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty", Supercomputing '91, November, 1991.

[2] Benitez, M.E., and Davidson, J.W., "Code Generation for Streaming: An Access/Execute Mechanism", ASPLOS-IV, April, 1991.

[3] Budnik, P., and Kuck, D., "The Organization and Use of Parallel Memories", IEEE Trans. Comput., 20, 12, 1971.

[4] Callahan, D., et.al., "Software Prefetching", ASPLOS-IV, April, 1991.

[5] Carr, S., Kennedy, K., "Blocking Linear Algebra Codes for Memory Hierarchies", Fourth SIAM Conference on Parallel Processing for Scientific Computing, 1989.

[6] Chen, T.-F., and Baer, J.-L., "Reducing Memory Latency via Non-blocking and Prefetching Caches", TR UW-CSE-92-06-03, Univ. of Washington, July, 1992.

[7] *Alpha Architecture Handbook*, Digital Equipment Corp., 1992.

[8] Gallivan, K., et.al., "The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design", TR UIUCSRD 625, Univ. of Illinois, 1987.

[9] Gannon, D., and Jalby, W., "The Influence of Memory Hierarchy on Algorithm Organization: Programming FFTs on a Vector Multiprocessor", in *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[10] Gao, Q.S., "The Chinese Remainder Theorem and the Prime Memory System", 20th ISCA, May 1993.

[11] Gupta, A., et.al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques", 18th ISCA, May, 1991.

[12] Harper, D. T., Jump, J., "Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme", IEEE Trans. Comput., 36, 12, 1987.

[13] Harper, D. T., "Address Transformation to Increase Memory Performance", 1989 International Conference on Supercomputing.

[14] "High-speed DRAMs", Special Report, IEEE Spectrum, vol. 29, no. 10, October, 1992.

[15] *i860 XP Microprocessor Data Book*, Intel Corp., 1991.

[16] Klaiber, A., et. al., "An Architecture for Software-Controlled Data Prefetching", 18th ISCA, May, 1991.

[17] Laird, M., "A Comparison of Three Current Superscalar Designs", Computer Architecture News, 20:3, June, 1992.

[18] Lam, M., et.al., "The Cache Performance and Optimizations of Blocked Algorithms", ASPLOS-IV.

[19] Lee, K. "On the Floating Point Performance of the i860 Microprocessor", NAS TR RNR-90-019, NASA Ames Research Center, Moffett Field, CA, July, 1992.

[20] Lee, K. "The NAS860 Library User's Manual", NAS TR RND-93-003, NASA Ames Research Center, Moffett Field, CA, March, 1993.

[21] Loshin, D., and Budge, D., "Breaking the Memory Bottleneck, Parts 1 & 2", Supercomputing Review, January/February, 1992.

[22] Meadows, L., et.al., "A Vectorizing Software Pipelining Compiler for LIW and Superscalar Architectures", RISC'92.

[23] McKee, S.A, "Hardware Support for Dynamic Access Ordering: Performance of Some Design Options", Univ. of Virginia, Department of Computer Science, TR CS-93-08, August, 1993.

[24] McKee, S.A., et.al., "Experimental Implementation of Dynamic Access Ordering", HICSS-27, Maui, HI, January, 1994.

[25] McKee, S.A., et.al., "Increasing Memory Bandwidth for Vector Computations", Lecture Notes in Computer Science 782 (PLSA, Zurich, Switzerland, March 1994), Springer Verlag, 1994.

[26] McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range", Lawrence Livermore National Laboratory, UCRL-53745, December, 1986.

[27] Mowry, T.C., et.al., "Design and Evaluation of a Compiler Algorithm for Prefetching", ASPLOS-V, September, 1992.

[28] Moyer, S.A., "Performance of the iPSC/860 Node Architecture," Univ. of Virginia, IPC-TR-91-007, 1991.

[29] Moyer, S.A., "Access Ordering and Effective Memory Bandwidth", Ph.D. Dissertation, Department of Computer Science, Univ. of Virginia, TR CS-93-18, April, 1993.

[30] Quinnell, R., "High-speed DRAMs", EDN, May 23, 1991.

[31] "Architectural Overview", Rambus Inc., 1992.

[32] Porterfield, A.K., "Software Methods for Improvement of Cache Performance on Supercomputer Applications", Ph.D. Thesis, Rice Univ., May, 1989.

[33] Rau, B. R., "Pseudo-Randomly Interleaved Memory", 18th ISCA, May, 1991.

[34] Sohi, G., and Franklin, M., "High Bandwidth Memory Systems for Superscalar Processors", ASPLOS-IV, April, 1991.

[35] Temam, O., et.al., "To Copy or Not to Copy: A Compile-Time Technique for Assessing When Data Copying Should Be Used to Eliminate Cache Conflicts", Supercomputing'93, December, 1993.

[36] Valero, M., et. al., "Increasing the Number of Strides for Conflict-Free Vector Access", 19th ISCA, May, 1992.

[37] Wallach, S., "The CONVEX C-1 64-bit Supercomputer", Compcon Spring 85, February, 1985.

[38] Wolfe, M., "More Iteration Space Tiling", Supercomputing '89, 1989.