Increasing Memory Bandwidth for Vector Computations

Sally A. McKee, Steven A. Moyer[1], Wm. A. Wulf

Department of Computer Science, Thornton Hall
University of Virginia, Charlottesville, VA 22903

Charles Hitchcock[2]

Thayer School of Engineering
Dartmouth College, Hanover NH 03755

Abstract. Memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to vector-like algorithms, including the "Grand Challenge" scientific problems. Caching is not the sole solution for these applications due to the poor temporal and spatial locality of their data accesses. Moreover, the nature of memories themselves has changed. Achieving greater bandwidth requires exploiting the characteristics of memory components "on the other side of the cache" — they should not be treated as uniform access-time RAM. This paper describes the use of hardware-assisted *access ordering*, a technique that combines compile-time detection of memory access patterns with a memory subsystem that decouples the order of requests generated by the processor from that issued to the memory system. This decoupling permits the requests to be issued in an order that optimizes use of the memory system. Our simulations show significant speedup on important scientific kernels.

1 Increasing Vector Memory Bandwidth

As processor speeds increase, memory bandwidth is becoming the limiting performance factor for many applications, particularly scientific computations. Although the addition of cache memory is often a sufficient solution to the memory latency and bandwidth problems in general-purpose scalar computing, the vectors used in scientific computations are normally too large to cache, and many are not reused soon enough to derive much benefit from caching. For computations in which vectors are reused, iteration space tiling [5, 21, 41] can partition the problem into cache-size blocks, but the technique is difficult to automate. Caching non-unit stride vectors may actually reduce a computation's effective memory bandwidth by fetching extraneous data. Thus, as noted by Lam *et al* [21], "while data caches have been demonstrated to be effective for general-purpose applications ..., their effectiveness for numerical code has not been established".

---

1. Current address: Department of Mathematics and Computer Science, Emory University, Atlanta, GA 30322.
2. Current address: Fostex R&D, 2 Buck Rd., Suite 2, Hanover, NH 03755.

The traditional scalar processor concern has been to minimize memory latency in order to maximize processor performance. For scientific applications, however, the processor is not the bottleneck, and as processor speeds continue to increase relative to memory speeds, optimal system performance will leave the processor idle at times. Bridging this performance gap requires changing the way we think about the problem — to maximize bandwidth for scientific applications, we need to minimize *average* latency over a coherent set of accesses.

While many scientific computations are limited by memory bandwidth, they are by no means the only such computations. Any computation involving linear traversals of vector-like data, where each element is typically visited only once during lengthy portions of the computation, can suffer: examples include string processing, image processing and other DSP applications, some database queries, some graphics applications, and DNA sequence matching.

After defining *access ordering*, our technique for improving vector memory bandwidth, we describe a hardware Stream Memory Controller (SMC) used to perform access ordering dynamically at run time, and discuss how this technique relates to other methods for improving memory system performance. We then describe the simulation environment used to evaluate SMC systems, and present results demonstrating the effectiveness of our technique. For long vectors, an SMC achieves nearly the full bandwidth that the memory system can deliver.

2 RAM *Isn't*

The assumptions made by most memory architectures simply don't match the physical characteristics of the devices used to build them. Memory components are usually presumed to require about the same amount of time to access any random location; it was this notion of uniform access time that originally gave rise to the term RAM, for Random Access Memory. Many computer architecture textbooks ([2, 14, 15, and 26] among them) specifically cultivate this view. Others skirt the issue entirely [25, 38].

Somewhat ironically, this assumption no longer applies to modern memory devices: most components manufactured in the last ten to fifteen years provide special capabilities that make it possible to perform some access sequences faster than others. For instance, nearly all current DRAMs implement a form of page-mode operation [32]. These devices behave as if implemented with a single on-chip cache line, or *page* (this should not be confused with a virtual memory 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 set up the new page makes servicing such an access significantly slower than one that hits the current page.

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 [33], Ramlink, and the new synchronous DRAM designs [16]). The order of requests strongly affects the performance of all these components. For instance, Rambus devices provide high bandwidth for large transfers, but offer little performance benefit for single-word accesses.

For multiple-module memory systems, the order of requests is important on yet another level: successive accesses to the same memory bank cannot be performed as quickly as accesses to different banks. To get the best performance out of such a system, we must take advantage of the architecture's available concurrency.

Most computers already have memory systems whose peak bandwidth is matched to the peak processor bus rate. But the nature of an algorithm, its data sizes, and placement all strongly affect memory performance; an architecture that works well on one problem may perform quite poorly on another. This was put in sharp focus for the authors while attempting to optimize numerical libraries for the iPSC/860. On some applications, even with *painstakingly* handcrafted code, inadequate memory bandwidth limited us to 20% of peak processor performance [30]. Our experience is not unique; results similar to ours have been reported by Lee [24], for example.

To illustrate one aspect of the bandwidth problem — and how it might be addressed at compile time —consider the effect of executing the fifth Livermore Loop (tridiagonal elimination) using non-caching accesses to reference a single bank of page-mode DRAMs. Figure 1(a) represents the natural reference sequence for a straightforward translation of the computation:

$$\forall i \qquad x_i \leftarrow z_i \times (y_i - x_{i-1})$$

This computation occurs frequently in practice, especially in the solution of partial differential equations by finite difference or finite element methods [9]. Since it contains a first-order linear recurrence, it cannot be vectorized. Nonetheless, the compiler can employ the recurrence detection and optimization algorithm of [6] to generate streaming code: each computed value x_i is retained in a register so that it will be available for use as x_{i-1} on the following iteration. Except in the case of very short vectors, elements from x, y, and z are likely to reside in different pages, so that accessing each vector in turn incurs the page miss overhead on each access; memory references likely to generate page misses are highlighted in the figure.

```
loop:                          loop:
   load z[i]                      load z[i]
   load y[i]                      load z[i+1]
   stor x[i]                      load y[i]
   jump loop                      load y[i+1]
                                  stor x[i]
                                  stor x[i+1]
                                  jump loop
     (a)                            (b)
```

Figure 1 *tridiag* Code

In the loop of Figure 1(a), a page miss occurs for every reference. Unrolling the loop and grouping accesses to the same vector, as in Figure 1(b), amortizes the page-miss cost over a number of accesses; in this case three misses occur for every six references. Reducing the page-miss count increases processor-memory bandwidth significantly. For example, consider a device for which the time required to service a page miss is

four times that for a page hit, a miss/hit cost ratio that is representative of current technology. The natural-order loop in Figure 1(a) only delivers 25% of the attainable bandwidth, whereas the unrolled, reordered loop in Figure 1(b) delivers 40%. External effects such as bus turnaround delays are ignored for the sake of simplicity.

Figure 2 illustrates effective memory bandwidth versus depth of unrolling, given a page-miss/page-hit cost ratio of four. For the bottom curve, the loop body of Figure 1(a) is essentially replicated the appropriate number of times, as is standard practice; for the middle curve, accesses have been arranged as per Figure 1(b); the top curve depicts the bandwidth attainable if all accesses were to hit the current DRAM page. Reordering the accesses realizes a performance gain of almost 130% at an unrolling depth of four, and over 190% at a depth of eight. Although in theory we could improve performance almost 240% by unrolling to a depth of sixteen, in most cases the size of the register file won't permit unrolling that far.

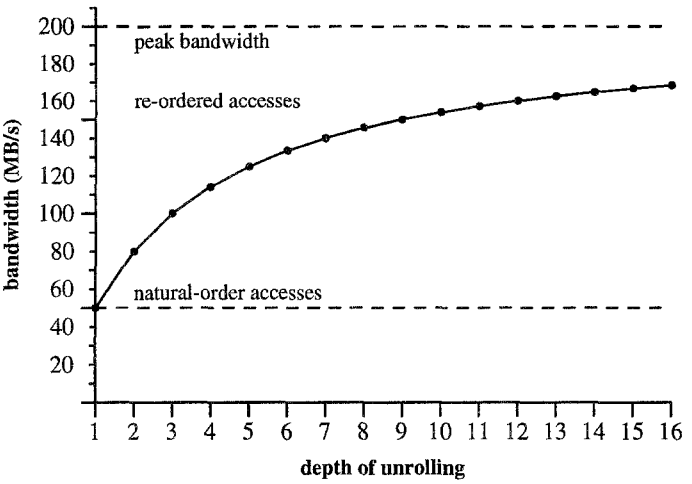

Figure 2 *tridiag* Memory Performance

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*, which we define as any technique for changing the order of memory requests to increase bandwidth. Here we are especially concerned with ordering a set of vector-like "stream" accesses.

As our example illustrates, the performance benefits of doing such static access ordering can be quite dramatic. Unfortunately, without the kinds of address alignment information that are usually only available at run time, the compiler can't generate the optimal access sequence. As pointed out above, the extent to which a compiler can perform this optimization is further constrained by such things as the size of the processor register file [31]. The beneficial impact of access ordering on effective memory bandwidth along with the limitations inherent in implementing the technique statically motivate us to consider an implementation that reorders accesses dynamically at run time.

There are a number of hardware and software techniques that can help manage the imbalance between processor and memory speeds. These include altering the placement of data to exploit concurrency [11], reordering the computation to increase locality (as in "blocking" [21]) address transformations for conflict-free access to interleaved memory [13, 34, 39], software prefetching data to the cache [4, 20, 37], and hardware prefetching vector data to cache [1, 8, 18, 35]. The main difference between these techniques and the complementary one we propose here is that we *reorder* stream accesses to exploit the architectural and component features that make memory systems sensitive to the sequence of requests.

3 A Taxonomy of Access Ordering Techniques

There are a number of options for when and how access ordering can be done, so first we provide a brief taxonomy of the design space. Access ordering systems can be classified by three key components:

- stream detection (*SD*), the recognition of streams accessed within a loop, along with their parameters (base address, stride, etc.),
- access ordering (*AO*), the determination of that interleaving of stream references that most efficiently utilizes the memory system, and
- access issuing (*AI*), the determination of when the load/store operations will be issued.

Each of these functions may be addressed at compile time, *CT*, or by hardware at run time, *RT*. This taxonomy classifies access ordering systems by a tuple (SD, AO, AI) indicating the time at which each function is performed.

Davidson [6] detects streams at compile time, and Moyer [31] has derived access-ordering algorithms relative to a precise analytic model of memory systems. Moyer's approach unrolls loops and orders memory operations to exploit architectural and device features of the target memory system. As our *tridiag* example illustrates, this (CT, CT, CT) system can improve bandwidth significantly, but is limited by the size of the processor register file and lack of vector alignment information available at compile time.

The purely compile-time approach can be augmented with an enhanced memory controller that provides buffer space and that automates vector prefetching, producing a (CT, CT, RT) system. Doing this relieves register pressure and decouples the sequence of accesses generated by the processor from the sequence observed by the memory components: the compiler determines a sequence of vector references to be issued and buffered, but the actual access issue is executed by the memory controller.

Both of these solutions are *static* in the sense that the order of references seen by the memory is determined at compile time; static techniques are inherently limited by the lack of alignment information. *Dynamic* access ordering systems introduce logic into the memory controller to determine the interleaving of a set of references.

For a dynamic (CT, RT, RT) system, stream descriptors are developed at compile time and sent to the memory controller at run time, where the order of memory references is determined dynamically and independently. Determining access order

dynamically allows the controller to optimize behavior based on run-time interactions. Our results illustrate the dramatic impact this has on bandwidth.

Fully dynamic (*RT, RT, RT*) systems implement access ordering without compiler support by augmenting the previous controller with logic to induce stream parameters. Whether or not such a scheme is superior to a (*CT, RT, RT*) system depends on the relative quality of the compile-time and run-time algorithms for stream detection and relative hardware costs. Proposals for (*RT, RT, RT*) "vector prefetch units" have recently appeared [1, 35], but these do not order accesses to fully exploit the underlying memory architecture.

4 The Stream Memory Controller

Based on our analysis and simulations, we believe that the best engineering choice is to detect streams at compile time, but to defer access ordering and issue to run time — (*CT, RT, RT*) in our notation. Choosing this scheme over an (*RT, RT, RT*) system follows a philosophy that has guided the design of RISC processors: move work to compile time whenever possible. This speeds processing and helps minimize hardware. Here we describe in general terms how such a scheme might be incorporated into an overall system architecture.

The approach we suggest is generally applicable to any uniprocessor computing system, but will be described based on the simplified architecture of Figure 3. Memory is interfaced to the processor through a controller labeled "MSU" for Memory Scheduling Unit. The MSU includes logic to issue memory requests as well as logic to determine the order of requests during streaming computations. For non-stream accesses, the MSU provides the same functionality and performance as a traditional memory controller. This is crucial — the access-ordering circuitry of the MSU is *not* in the critical path to memory and doesn't affect scalar processing.

The MSU has full knowledge of all streams currently needed by the processor: given the base address, vector stride, and vector length, it can generate the addresses of all elements in a stream. The scheduling unit also knows the details of the memory architecture, including interleaving and device characteristics. The access-ordering circuitry uses this information to issue requests for individual stream elements in an order that attempts to optimize memory system performance.

A separate Stream Buffer Unit (SBU) provides high-speed buffers for stream operands and control registers that the processor uses to specify stream parameters (base address, stride, length, and data size). As with the stream-specific parts of the MSU, the SBU is not on the critical path to memory, and the speed of non-vector accesses is not adversely affected by its presence. Together, the MSU and SBU comprise a Stream Memory Controller (SMC) system.

There are a number of options for the internal architecture of the SBU: here we describe one feasible organization. A set of memory-mapped registers provides a processor-independent means of specifying stream parameters. Setting these registers allows the processor to initiate an asynchronous stream of memory access operations for a set of string operands. Data retrieval from the streams (loads) and insertion into streams (stores) may be done in any of several ways; for instance, the SBU could appear

to be a traditional cache, or the model could include a set of FIFOs, as illustrated in Figure 3. Each stream is assigned to one FIFO, which is asynchronously filled from (or drained to) memory by the access/issue logic. The "head" of the FIFO is another memory-mapped register, and load instructions from or store instructions to a particular stream reference the FIFO head via this register, dequeueing or enqueueing data as is appropriate.

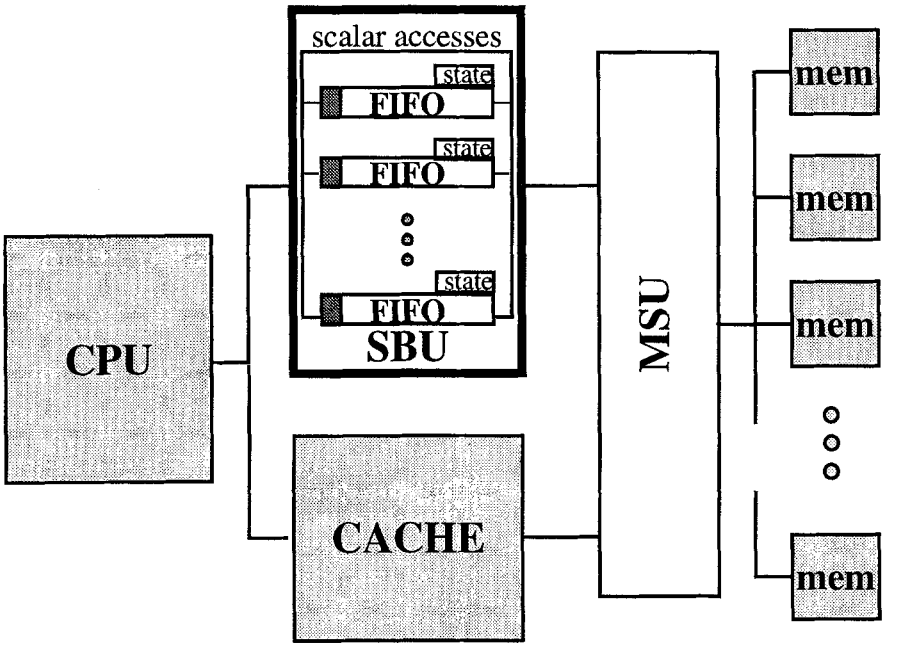

Figure 3 Stream Memory Controller

This organization is both simple and practical from an implementation standpoint: similar designs have been built. In fact, the organization is almost identical to the "stream units" of the WM architecture [42], or may be thought of as a special case of a decoupled access-execute architecture [10, 36]. Another advantage is that this combined hardware/software scheme doesn't require heroic compiler technology —the compiler need only detect the presence of streams, and Davidson's streaming algorithm [6] can be used to do this.

Continuing the *tridiag* algorithm and memory system example introduced earlier, the performance effect of such an SMC is illustrated by Figure 4. The details of this and other results are discussed later, but the gestalt is simple — performance on very short vectors is about 2.5 times that of a system without an SMC; performance on moderate length vectors is about triple that of the non-SMC system; for long vectors and deep FIFOs, bandwidth reaches 98.5% of peak.

vector length	Percentage of Peak Bandwidth						
	w/o SMC	SMC FIFO depth					
		8	16	32	64	128	256
10	25.0	63.83	62.5	62.5	62.5	62.5	62.5
100		78.53	85.71	87.98	80.43	73.53	73.53
10000		79.94	88.78	93.97	96.75	98.11	98.53

Figure 4 Bandwidth for the *tridiag* Illustration

5 Complementary Technologies

As mentioned above, there are a number of hardware and software techniques that can help manage the imbalance between processor and memory speeds. Most of these are complementary to access ordering.

Traditional Caching: Traditional caches retain their importance for code and non-vector data in a system equipped with an SMC. Furthermore, if algorithms can be blocked [5, 41] and data aligned to eliminate significant conflicts [21], the cache and SMC can be used in a complementary fashion for vector access. Under these conditions multiple-visit vector data can be cached, with the SMC used to reference single-visit vectors.

To illustrate this, consider implementing the matrix-vector multiply operation:

$$\bar{y} = (A+B)\,\bar{x}$$

where A and B are $n \times m$ matrices and $\bar{y}$ and $\bar{x}$ are vectors. Figure 5(a) depicts code for a straightforward implementation using matrices stored in column-major order; the code in Figure 5(b) strip-mines the computation to reuse elements of $\bar{y}$. Partition size depends on cache size and structure [21]. Elements of $\bar{y}$ are preloaded into cache memory at the appropriate loop level, and the SMC is then used to access elements of A and B, since each element is accessed only once. The reference to $\bar{x}$ is a constant within the inner loop, and is therefore preloaded into a processor register.

Software Prefetching: Some architectures include instructions to prefetch data from main memory into cache. Using these instructions to load data for a future iteration of a loop [4, 20, 37] can improve processor performance by overlapping memory latency with computation, but prefetching does nothing to actually *improve* memory performance. Note that prefetching can be used in conjunction with an SMC to help hide latency in FIFO references.

Software Access Ordering: Software techniques such as reordering [30] and "vectorization" via library routines [24, 29] can improve bandwidth by reordering

```
   do 20 j = 1,m
      do 10 i = 1,n
         y(i) = y(i) + (A(i,j) + B(i,j)) * x(j)
10    continue
20 continue
```

(a) straightforward implementation

```
   do 30 IT = 1,n,IS
      load y(IT) through y(min(n,IT+IS-1)) into cache
      do 20 j = 1,m
         load x(j) into processor register
         do 10 i = IT,min(n,IT+IS-1)
            y(i) = y(i) + (A(i,j) + B(i,j)) * x(j)
10       continue
20    continue
30 continue
```

(b) strip-mined implementation

Figure 5 Combining Caching and Non-Caching Accesses: $\bar{y} = (A + B)\bar{x}$

requests at compile time. Such techniques cannot exploit run-time information and are limited by processor register resources, hence they cannot outperform hardware-assisted techniques such as the SMC.

Data Placement: An SMC can provide near-optimal bandwidth for a given memory architecture, algorithm, and data placement, but cannot compensate for an unfortunate placement of operands — a vector stride that results in all elements placed in a single bank of a multi-bank memory, for example. An SMC and data placement are complementary; the SMC will perform better given a good placement.

New DRAM interfaces: Rambus [33] is a new, high-speed DRAM interface that provides both higher bandwidth for sequential accesses and true caching of two DRAM pages on the chip. Other sophisticated memory interfaces, such as RamLink and the JEDEC synchronous DRAM, provide similar benefits [16]. The more sophisticated such interfaces become, the more important it is to exploit them intelligently with controllers such as the SMC.

Alternative Storage Schemes: Skewed storage [3, 12] and dynamic address transformations [13, 34] have been proposed as methods for increasing concurrency, and hence bandwidth, in parallel memory systems. Unfortunately, these techniques do not work for interspersed multiple streams, and they do not exploit memory component features.

6 Simulation Environment

We have simulated a wide range of SMC configurations and benchmarks, varying

- FIFO depth,
- dynamic order/issue policy,
- number of memory banks,
- DRAM speed,
- benchmark algorithm, and
- vector length, stride, and alignment with respect to memory banks.

Only samples of our results are given here; complete results can be found in [28]. In particular, the results given below involve the following restrictions.

The simulations here use stride-one vectors aligned to have no DRAM pages in common, but starting in the same bank unless otherwise specified. The SMC is very robust in its ability to optimize memory bandwidth regardless of stride and alignment, so this restriction does not materially affect the results.

We model the processor as a generator of load and store requests only — arithmetic and control are assumed never to be a computational bottleneck. This places the maximum stress on the memory system by assuming a computation rate that out-paces the memory's ability to transfer data. Scalar and instruction accesses are assumed to hit in the cache for the same reason.

All memories modeled here consist of interleaved banks of page-mode DRAMs, where each page is 2K double words.

The only order/issue policy considered is exceedingly simple. The SMC looks at each FIFO in round-robin order, issuing accesses for the same FIFO stream while:

1) not all elements of the stream have been accessed, and
2) there is room in the FIFO for another read operand, or another write operand is present in the FIFO.

Results reported here are for the four kernels described in Figure 6. *Daxpy* and *swap* are from the BLAS (Basic Linear Algebra Subroutines) [22, 7], *tridiag* is the fifth Livermore Loop from our earlier example[27], and *vaxpy* is a vector *axpy*[1] computation that occurs in matrix-vector multiplication by diagonals. These benchmarks were selected because they are representative of the access patterns found in real scientific codes, including the inner loops of blocked algorithms. Nonetheless, our results show that the actual reference sequence has little effect on SMC performance.

Non-SMC results are for the "natural" reference sequence for each benchmark, using non-caching loads and stores.

SMC initialization requires two writes to memory-mapped registers for each stream; this small overhead has no significant effect on results, and is not included here.

The DRAM page-miss cycle time is four times that of a DRAM page hit, unless otherwise noted.

---

1. Here "axpy" refers to a computation involving some entity a times a vector x plus a vector y: for *daxpy*, a is a double; for *vaxpy*, a is a vector.

daxpy:	$\forall i$	$y_i \leftarrow ax_i + y_i$		
tridiag:	$\forall i$	$x_i \leftarrow z_i \times (y_i - x_{i-1})$		
swap:	$\forall i$	$tmp \leftarrow y_i$	$y_i \leftarrow x_i$	$x_i \leftarrow tmp$
vaxpy:	$\forall i$	$y_i \leftarrow a_i x_i + y_i$		

Figure 6 Benchmark Algorithms

7 Results

Figure 7 through Figure 10 illustrate the relative performance of the four kernels for a variety of memory systems using an SMC. The SMC's ability to optimize bandwidth is relatively insensitive to vector access patterns, hence the shape of the performance curves is similar for all benchmarks — asymptotic behavior approaches 100%.

Figure 7 shows SMC performance for long vectors (length 10,000) as a function of FIFO depth and number of memory banks (available concurrency) compared to the analogous non-SMC systems. On the *daxpy* benchmark, for example, an SMC system with two memory banks achieves 98.2% of peak bandwidth, compared to 18.8% for a non-SMC system. In general, SMC systems with deep FIFOs achieve in excess of 92% of peak bandwidth for all benchmarks and memory configurations. Even with FIFOs that are only sixteen double-words deep, the SMC systems consistently deliver over 75% of peak bandwidth.

Note that increasing the number of banks reduces *relative* performance, a somewhat counter-intuitive and deceptive effect. This is due in part to our keeping both the peak memory system bandwidth and the DRAM page-miss/hit delay ratio constant. Thus, the eight-bank system has four times the DRAM page-miss latency of the two-bank system. If, alternatively, we hold the performance of the memory banks constant and assume a faster bus, the peak bandwidth of the total system increases proportionally to the number of banks. Although the percentage of bandwidth delivered is still smaller in this case, the total bandwidth is much larger. The other reason for this effect is that increasing the number of banks decreases the number of accesses to each bank, thus page-miss cost is amortized over fewer accesses.

Figure 8 represents SMC results for medium-length (100 element) vectors compared with non-SMC performance. These SMC results depict the net effect of two competing performance factors. With deeper FIFOs, DRAM page misses are amortized over a larger number of total accesses, which can increase performance. At the same time, the processor has to wait longer to complete its first loop iteration while the SMC prefetches numerous operands to be used in the following loop iterations. This can decrease performance, as evidenced by the tail-off beyond length-32 FIFOs. Optimum FIFO depth could, and probably should, be run-time selectable in the SMC, since it is so closely related to stream length.

Note that performance of non-SMC systems depicted in Figure 7 and Figure 8 is independent of vector length. Since these systems employ no dynamic access ordering,

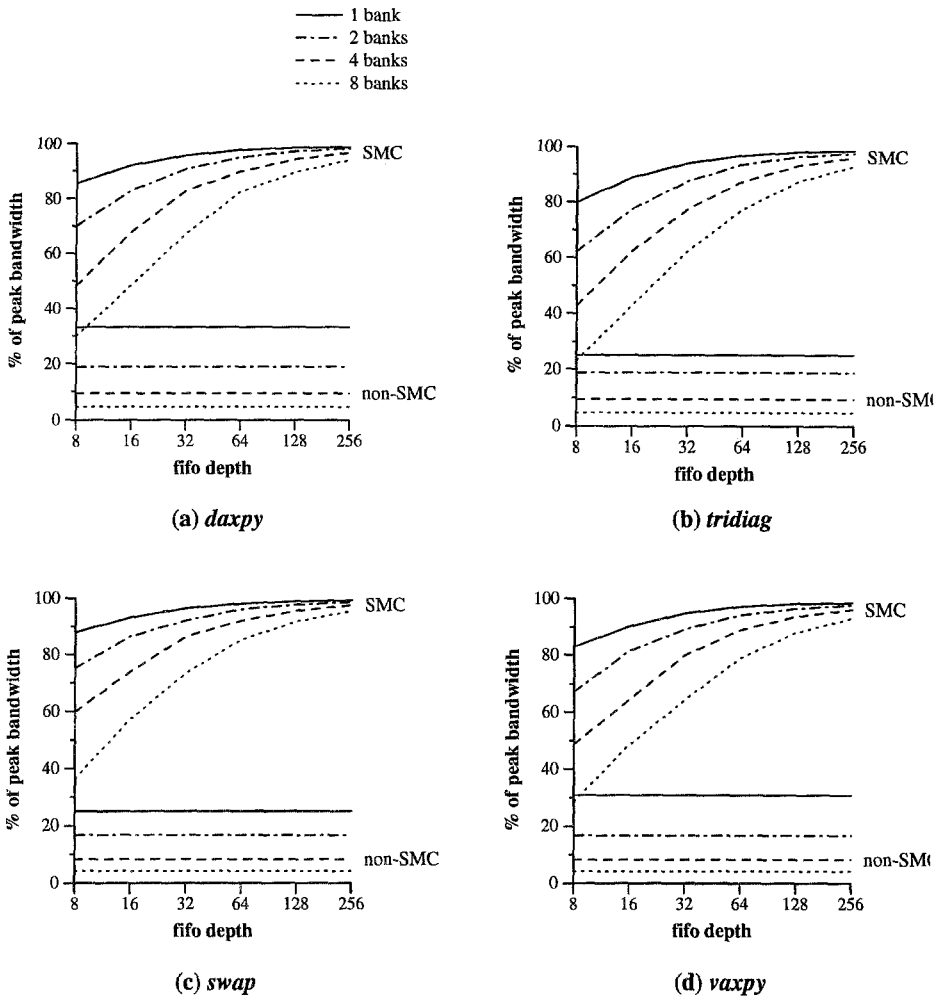

Figure 7 Long Vector Performance

the number of requests issued and the resulting percentage of total bandwidth obtained are constant for each loop iteration. This is true of any system in which access issue is determined at compile time, including those that use prefetching.

Lack of dynamic ordering renders the performance of non-SMC systems particularly sensitive to vector placement. In Figure 7 and Figure 8, the vectors are aligned so that they both compete for the same bank on each iteration; this has little effect on SMC performance (because it reorders requests), but it prevents the non-SMC systems from taking advantage of the potential concurrency.

Figure 9 represents the performance of non-SMC and SMC systems on medium-length vectors with better alignment. In these experiments the vectors are staggered such that the i^{th} vector in the pattern begins in bank (i mod n), where n is the number of banks. In spite of the more favorable alignment, non-SMC *daxpy* performance is limited

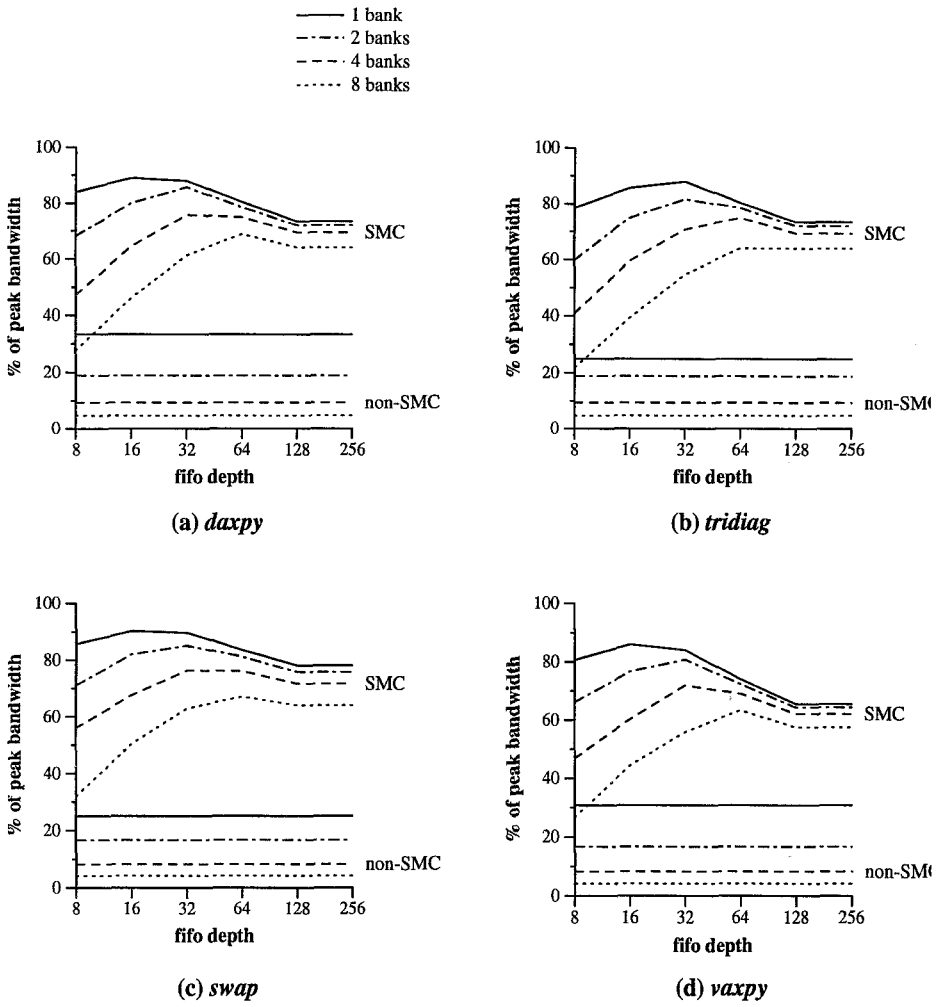

Figure 8 Medium Vector Performance

to 30.0% of total bandwidth for a two-bank memory; *tridiag*, *swap*, and *vaxpy* are limited to 18.8%, 40.0%, and 25.0%, respectively. SMC performance remains essentially the same as in Figure 8.

Figure 10 illustrates SMC performance on very short (10-element) vectors. Performance improvements are not as dramatic as for longer vectors, since there are even fewer accesses over which to amortize page-miss costs. Nonetheless, short vector computations benefit significantly from an SMC: *daxpy* run on a two-bank architecture with an SMC achieves 53.6% of the attainable bandwidth, whereas the same benchmark run on a similar non-SMC system is limited to 18.8%. The other kernels enjoy similar increases in bandwidth. Non-SMC performance is as in Figure 8 or Figure 9, depending on vector alignment; those lines are omitted here for clarity.

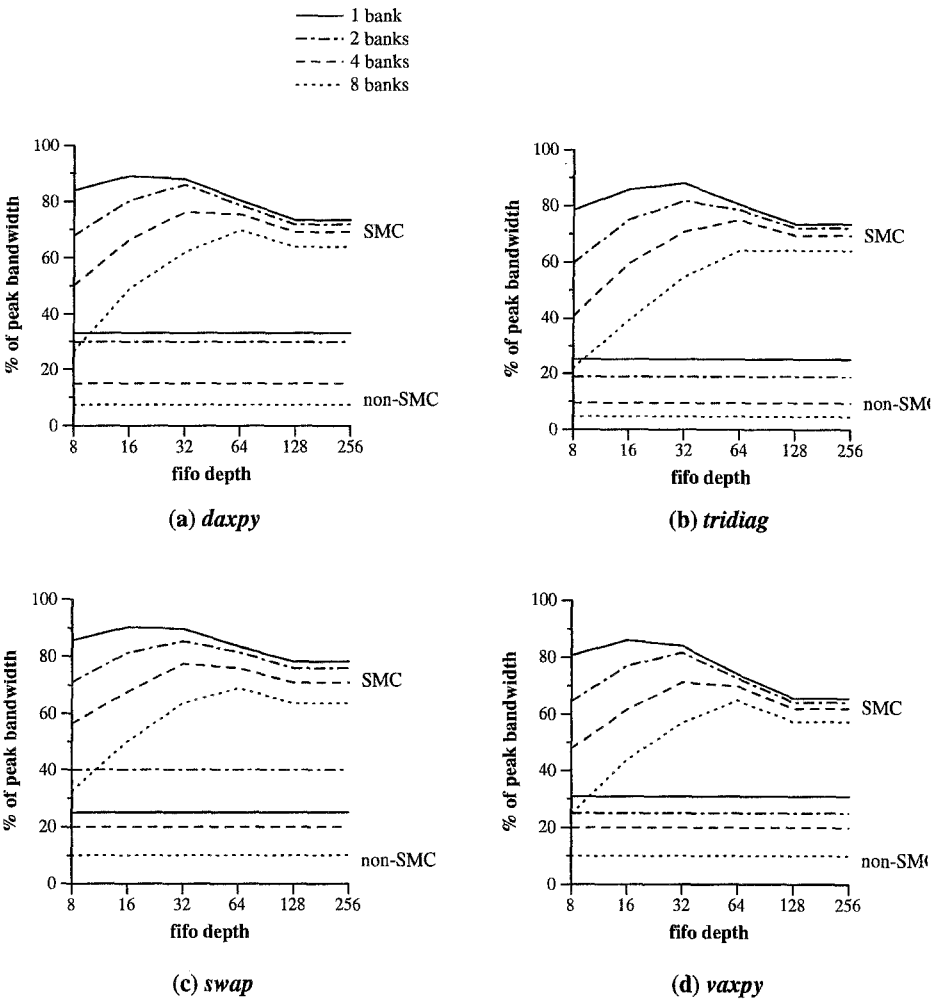

Figure 9 Medium Vector Performance for Better Vector Alignment

Figure 11(a) shows SMC performance for *daxpy* using long vectors (10,000 elements) as the page-miss to page-hit cost ratio increases. This figure may be a bit misleading: the miss/hit ratio is likely to increase primarily as the result of a reduction of the page-hit time, rather than an increase in the page-miss time. Thus, at a ratio of sixteen the SMC is delivering a somewhat smaller percentage of a *much* larger available bandwidth — resulting in a significant net increase.

If we hold the number of modules fixed and increase component performance, deeper FIFOs are required in order to amortize the page-miss costs. As evidenced by the slope of the curves in Figure 11(b), relative performance is approximately constant if we scale FIFO depth linearly with miss/hit cost. Note that the faster systems still require only modest amounts of buffer storage.

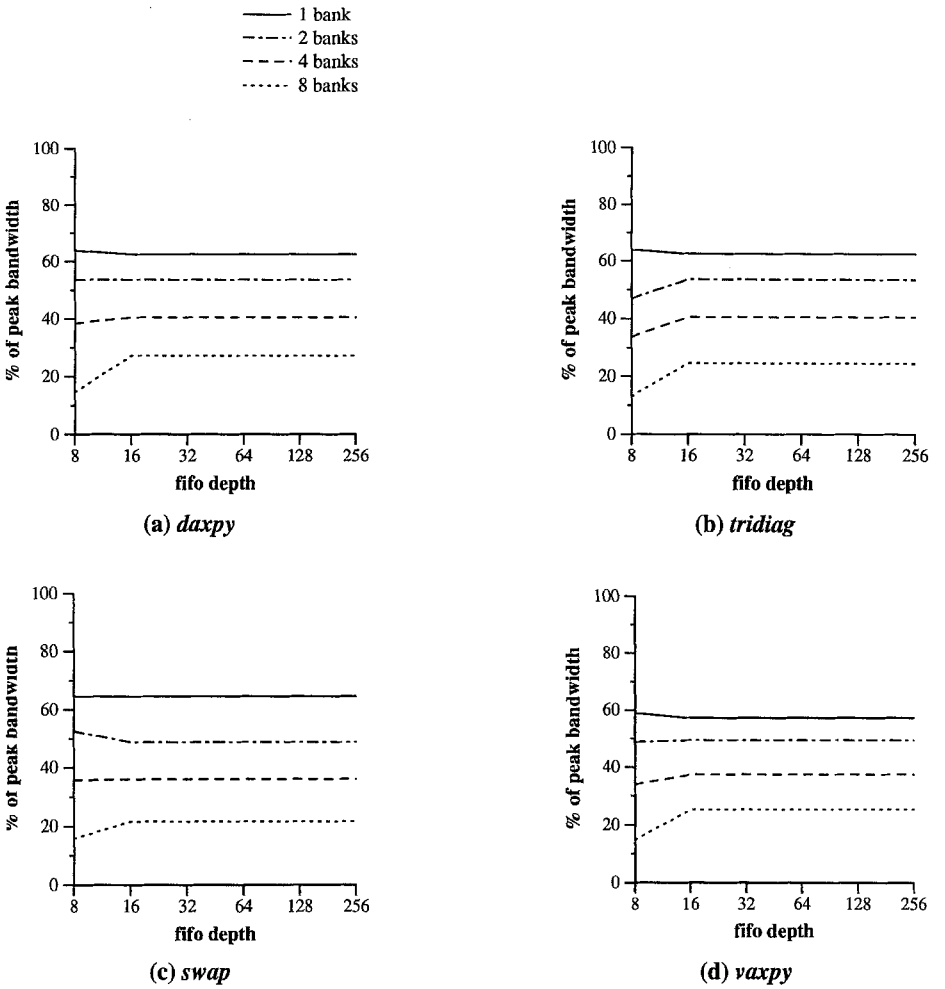

Figure 10 Very Short Vector Performance

8 Conclusions

Memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to vector-like algorithms, including many of the "grand challenge" scientific problems. Caching is not the sole solution for these applications due to their poor temporal and spatial locality.

Achieving greater bandwidth requires exploiting the characteristics of the entire memory hierarchy; it cannot be treated as though it were uniform access-time RAM. Moreover, exploiting the memory's properties will have to be done dynamically — essential information (such as alignment) will generally not be available at compile time.

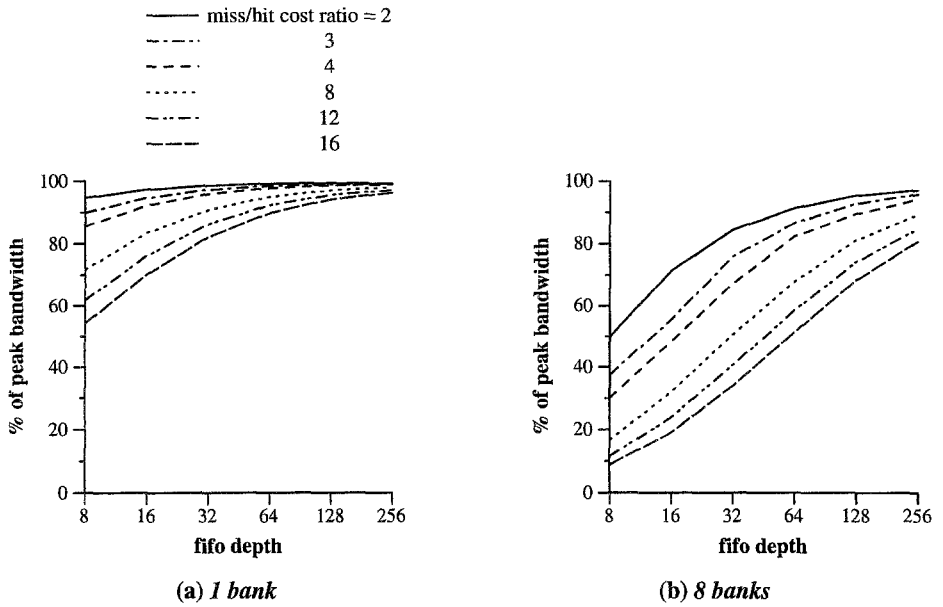

Figure 11 *daxpy* Performance for Various Page-miss/Page-hit Cost Ratios

Reordering can optimize accesses to exploit the underlying memory architecture. By combining compile-time detection of streams with execution-time selection of the access order and issue, we achieve near-optimal bandwidth for vector-like accesses relatively inexpensively. This complements more traditional cache-based schemes, so that overall effective memory performance need not be a bottleneck.

Here we have reported the basic design of a Stream Memory Controller (SMC) and have demonstrated its performance for a variety of FIFO depths, memory configurations, etc. Using current memory parts and only a few hundred words of buffer storage, an SMC system can deliver nearly the full memory system bandwidth. Moreover, it does so with naive code, and performance is independent of the alignment and stride of the operands.

9. Acknowledgments

Thanks go to the other members of Bill Wulf's research group for their valuable feedback: Scott Briercheck, Rob Craighurst, Katie Oliver, Ramesh Peri, and Alec Yasinsac. This work has been supported in part by a grant from Intel Supercomputer Division and by NSF contract MIP-9114110.

References

1. Baer, J. L., Chen, T. F., "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty", Supercomputing'91, November 1991.
2. Baron, R.L., and Higbie, L., *Computer Architecture*, Addison-Wesley, 1992.
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", Fourth International Conference on Architectural Support for Programming Languages and Systems, April 1991.
5. Carr, S., Kennedy, K., "Blocking Linear Algebra Codes for Memory Hierarchies", Proc. Fourth SIAM Conference on Parallel Processing for Scientific Computing, 1989.
6. Davidson, Jack W., and Benitez, Manuel E., "Code Generation for Streaming: An Access/Execute Mechanism", Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991.
7. Dongarra, et. al., "Linpack User's Guide", SIAM, Philadelphia, 1979.
8. Fu, J. W. C., and Patel, J. H., "Data Prefetching in Multiprocessor Vector Cache Memories", 18th International Symposium on Computer Architecture, May 1991.
9. Golub, G., and Ortega, J.M., *Scientific Computation: An Introduction with Parallel Computing*, Academic Press, Inc., 1993.
10. Goodman, J. R., et al, "PIPE: A VLSI Decoupled Architecture", Twelfth International Symposium on Computer Architecture, June 1985.
11. Gupta, R., and Soffa, M., "Compile-time Techniques for Efficient Utilization of Parallel Memories", SIGPLAN Not., 23, 9, 1988, pp. 235-246.
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. Hayes, J.P., *Computer Architecture and Organization*, McGraw-Hill, 1988.
15. Hwang, K., and Briggs, F.A., *Computer Architecture and Parallel Processing*, McGraw-Hill, Inc., 1984.
16. "High-speed DRAMs", Special Report, IEEE Spectrum, vol. 29, no. 10, October 1992.
17. *i860 XP Microprocessor Data Book*, Intel Corporation, 1991.
18. Jouppi, N., "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers", 17th International Symposium on Computer Architecture, May 1990.
19. Katz, R., and Hennessy, J., "High Performance Microprocessor Architectures", University of California, Berkeley, Report No. UCB/CSD 89/529, August, 1989.
20. Klaiber, A., et. al., "An Architecture for Software-Controlled Data Prefetching", 18th International Symposium on Computer Architecture, May 1991.
21. Lam, Monica, et. al., "The Cache Performance and Optimizations of Blocked Algorithms", Fourth International Conference on Architectural Support for Programming Languages and Systems, April 1991.
22. Lawson, et. al., "Basic Linear Algebra Subprograms for Fortran Usage", ACM Trans. Math. Soft., 5, 3, 1979.

23. Lee, K., "Achieving High Performance On the i860 Microprocessor Using Naspack Subroutines", NAS Systems Division, NASA Ames Research Center, July 1990.
24. Lee, K., "On the Floating Point Performance of the i860 Microprocessor", RNR-90-019, NAS Systems Division, NASA Ames Research Center, October 1990.
25. Maccabe, A.B., *Computer Systems: Architecture, Organization, and Programming*, Richard D. Irwin, Inc., 1993.
26. Mano, M.M., *Computer System Architecture*, 2nd ed., Prentice-Hall, Inc., 1982
27. McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range", Lawrence Livermore National Laboratory, UCRL-53745, December 1986.
28. McKee, S.A, "Hardware Support for Access Ordering: Performance of Some Design Options", University of Virginia, Department of Computer Science, Technical Report CS-93-08, July 1993.
29. Meadows, L., Nakamoto, S., and Schuster, V., "A Vectorizing, Software Pipelining Compiler for LIW and Superscalar Architectures", RISC'92, February 1992.
30. Moyer, S.A., "Performance of the iPSC/860 Node Architecture," University of Virginia, IPC-TR-91-007, 1991.
31. Moyer, S., "Access Ordering and Effective Memory Bandwidth", Ph.D. Dissertation, Department of Computer Science, University of Virginia, Technical Report CS-93-18, April 1993.
32. Quinnell, R., "High-speed DRAMs", EDN, May 23, 1991.
33. "Architectural Overview", Rambus Inc., Mountain View, CA, 1992.
34. Rau, B. R., "Pseudo-Randomly Interleaved Memory", 18th International Symposium on Computer Architecture, May 1991.
35. Sklenar, Ivan, "Prefetch Unit for Vector Operation on Scalar Computers", Computer Architecture News, 20, 4, September 1992.
36. Smith, J. E., et al, "The ZS-1 Central Processor", The Second International Conference on Architectural Support for Programming Languages and Systems, Oct. 1987
37. Sohi, G. and Manoj, F., "High Bandwidth Memory Systems for Superscalar Processors", Fourth International Conference on Architectural Support for Programming Languages and Systems, April 1991.
38. Tomek, I., *The Foundations of Computer Architecture and Organization*, Computer Science Press, 1990.
39. Valero, M., et. al., "Increasing the Number of Strides for Conflict-Free Vector Access", 19th International Symposium on Computer Architecture, May 1992.
40. Wallach, S., "The CONVEX C-1 64-bit Supercomputer", Compcon Spring 85, February 1985.
41. Wolfe, M., "Optimizing Supercompilers for Supercomputers", MIT Press, Cambridge, MA, 1989.
42. Wulf, W. A., "Evaluation of the WM Architecture", 19th Annual International Symposium on Computer Architecture, vol 20, no. 2, May 19-21, 1992.