A Memory Controller for Improved Performance of Streamed Computations on Symmetric Multiprocessors

Sally A. McKee and Wm. A. Wulf
Department of Computer Science
Thornton Hall, University of Virginia
Charlottesville, VA 22903
{mckee | wulf}@cs.virginia.edu

Abstract

The growing disparity between processor and memory speeds has caused memory bandwidth to become the performance bottleneck for many applications. In particular, this performance gap severely impacts stream-orientated computations such as (de)compression, encryption, and scientific vector processing. This paper describes the development of an intelligent memory interface that can exploit compiler-provided information on streamed memory access patterns to improve memory bandwidth. Simulation results show that such shared-memory multiprocessor systems can deliver nearly the full attainable bandwidth with relatively modest hardware costs.

1. Introduction

It has become painfully obvious that processor speeds are increasing much faster than memory speeds. For example, a 300 MHz DEC Alpha can perform more than 20 instructions in the time required to complete a single memory access to a 40ns DRAM.

Caching has often been used to bridge the gap between microprocessor and DRAM performance, but as the memory bandwidth problem grows, the effectiveness of the technique is rapidly diminishing [Bur95, Wul95]. Even if the addition of cache memory is a sufficient solution for general-purpose scalar computing (and some portions of streaming computations), its effectiveness for vector processing is still subject to debate. The streams used in these computations are normally much too large to cache in their entirety, and each element is visited only once during lengthy portions of the computation. This lack of temporal locality of reference makes caching less effective than it might be for other parts of the program. The kinds of applications that are particularly affected by the growing processor-memory performance gap include scientific computations, multi-media (de)compression, encryption, signal processing, and text searching, to name a few.

A comprehensive solution to this bandwidth problem must exploit the richness of the full memory hierarchy, including component capabilities. We have proposed part of such a solution in the form of a *Stream Memory Controller* (SMC) that reorders accesses dynamically at run-time [McK94a, McK94b].

2. Access Ordering

The performance of most memory systems is dependent upon the actual sequence of address references. An interleaved system, for example, performs better if the order of accesses permits concurrency among the banks. Order matters at an even lower level, too: most memory devices manufactured in the last decade provide special features (nibble-mode, static column mode, or a small amount of SRAM cache on chip) or exhibit novel organizations (such as Rambus, Ramlink, and synchronous DRAM designs [IEE92]) that make it possible to perform some access sequences faster than others. Effective bandwidth can be increased by arranging requests to take advantage of these capabilities.

Here we focus on *fast-page mode* devices, which behave as if they were implemented with a single on-chip cache line, or *page*. A memory access falling outside the address range of the current page forces a new one to be set up, a process that is significantly slower than repeating an access to the current page.

Access ordering is any technique that changes the order of memory requests from that generated by the issuing program. We are specifically concerned with ordering vector-like *stream* accesses to exploit multi-bank systems using devices with special properties like page-mode. In earlier work [McK94a, McK94b], we proposed a combined hardware/software scheme for implementing access ordering dynamically at run-time, and presented numerous simulation results demonstrating its effectiveness on a single-processor system.

In this paper we analyze the performance of this scheme for symmetric multiprocessor systems. We find that the way in which a computation is partitioned has a significant impact on memory performance: as expected, the highest bandwidth is achieved when processors share the same working set of DRAM pages throughout most of the computation. For long-vector computations exhibiting a high degree of DRAM page-sharing, the SMC can deliver nearly the full system bandwidth.

3. The SMC

There are many ways to approach the bandwidth problem, either in hardware or software. In addition to traditional caching, other proposed solutions range from software prefetching [Cal91, Mow92] and iteration space tiling [Car89, Lam91, Wol89], to address transformations [Har89], unusual memory systems [Gao93, Rau91, Val92], and prefetching hardware or non-blocking caches [Bae91, Che92, Soh91, Chi94, Jou90]. Most of these schemes simply mask latency without increasing effective bandwidth. They are still useful, but will be most effective when combined with complementary technology to take advantage of memory component capabilities.

Software access-ordering techniques range from Moyer's algorithms for non-caching register loads [Moy93] to schemes that stream vector data into the cache, explicitly managing it as a fast local memory [Lee93, Mea92, Pal95]. We have studied access-ordering in depth [McK95], developing performance bounds for these and other access-ordering schemes. Compile-time approaches are limited by contention for processor resources (e.g., register pressure or cache conflicts) and the lack of data placement and alignment information. These limitations motivated us to consider an implementation that dynamically reorders accesses at run-time. Benitez and Davidson's algorithm can be used to detect streams at compile-time [Ben91], and the stream parameters can be transmitted to the reordering hardware at run-time.

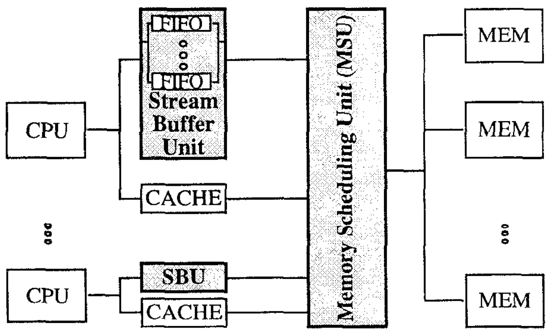

Figure 1 MP Stream Memory Controller System

Our analysis is based on the simplified architecture of Figure 1. In this system, all processors are interfaced to memory through a centralized controller, or *Memory Scheduling Unit* (MSU). The MSU includes logic to issue memory requests and to determine the order of requests during streaming computations. For non-stream accesses (which may or may not go through the cache) the MSU provides the same functionality and performance as a traditional memory controller.

Given the base address, stride, data size, and vector length (derived from compile-time analysis) of all streams currently needed by the processors, the MSU can generate the addresses of all elements in those streams. The scheduling unit also knows the details of the memory architecture, including the degree of interleaving and the characteristics of the memory components. 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) for each processor contains high-speed buffers for stream operands and provides memory-mapped registers that the processor uses to specify stream parameters. Together, the MSU and SBU comprise a *Stream Memory Controller* (SMC) system.

From the processor's point of view, the stream buffers are implemented logically as a set of FIFOs within the SBU. Each stream is assigned to one FIFO, which is asynchronously filled from or drained to memory in an order determined by the access/issue logic of the MSU. The MSU need not perform the accesses in FIFO order. The FIFO "head" is another memory-mapped register, and load or store instructions for a particular stream reference this register, dequeueing or enqueueing data as is appropriate.

Note that since cache placement does not affect the SMC, logically the system could consist of a single cache for all CPUs or separate caches for each — the choice is an implementation issue. Figure 1 depicts separate caches to emphasize the fact that the SBUs and cache reside at the same logical level of the memory hierarchy.

Due to both the high communication requirements for a fully distributed (multiple-MSU) approach and the limitations on the number of processors that may share a centralized resource, we do not expect shared-memory SMC systems to scale to large numbers of processors. Here we focus on the performance of systems with two to eight processors. The design of SMC systems that scale to more CPUs is an interesting issue for further research.

4. Parallelization Schemes

The way in which a problem is partitioned for a multiprocessor system can have a marked effect on effective memory bandwidth. In particular, SMC performance is affected by whether the working sets of DRAM pages needed by different CPUs overlap during the course of the computation. If they overlap, the set of FIFOs using data from a particular page will be larger, making it possible for the SMC to get more page-hits.

The physical layout of vectors in memory also affects the working sets of DRAM pages. For the experiments described here, we use a DRAM page size of 4K bytes, thus each page can hold 512 double-word vector elements. On an 8-way interleaved memory system, a computation incurs an initial page miss on each bank, but $512 \times 8 = 4096$ elements of a given vector can be accessed before crossing a page boundary (assuming vectors are page-aligned). On a 16-bank system, the vectors cross DRAM page boundaries at element 8192.

Here we focus on two partitioning models. The first, *cyclic scheduling*, distributes loop iterations among the CPUs, as in a FORTRAN DOALL. This scheme makes the *effective stride* at each CPU N times the original stride. If the number of banks is a multiple of the number of CPUs, then a different subset of banks will provide the data for each CPU. Since each of the N processors references every Nth vector element, all CPUs will tend to share the same set of DRAM pages throughout most of the computation. If the processors proceed at different rates, some may cross page boundaries slightly sooner than others, but recent empirical studies suggest that the slowest processor is normally not more than the mean execution time of one loop iteration behind the average processor [LiN94].

The second strategy we investigate is block scheduling. Here the vector is split into approximately equal-size pieces, and each processor performs the computation on a single piece. This partitioning scheme makes it likely that the portions of a stream assigned to different CPUs will reside in different DRAM pages, resulting in less overlap among the different processors' working sets of pages.

5. Simulation Environment

We have simulated a wide range of SMC configurations and benchmarks, varying FIFO depth; dynamic order/issue policy; number of CPUs; number of memory banks; DRAM speed and page size; benchmark kernel; and vector length, stride, and alignment with respect to memory banks. Here we focus on the general performance trends for a single ordering policy when used with both cyclic and block task partitioning. Complete uniprocessor results, including a detailed description of each access-ordering heuristic, can be found in [McK95]. The overwhelming similarity of the performance curves presented in our uniprocessor SMC studies indicates that neither the ordering strategy nor the processor's access pattern has a large effect on the MSU's ability to optimize bandwidth. Complete shared-memory multiprocessor results can be found in [McK95].

We assume the system is matched so that bandwith between the CPUs and the SMC equals that between the SMC and memory. In order to stress the memory system as much as possible, we model the CPU as a generator of non-cached loads and stores of vector elements. Our results are therefore given as a percentage of the system's peak bandwidth, or that necessary to allow each processor to perform one memory operation per cycle. Instruction and scalar data references are assumed to hit in the cache, and all stream references use non-caching loads and stores.

We model DRAMs with a page size of 4K bytes; accesses that miss the current page take four times as long as those that hit. Our vectors are of equal length and stride, share no DRAM pages in common, and are aligned such that the i^{th} elements of each vector reside in the same bank. Finally, we assume a model of operation in which each CPU accesses its FIFOs in order, consuming or producing one data item per FIFO during each loop iteration.

The ordering policy we consider here operates as follows. At each available memory bus cycle the MSU attempts to initiate an access to the next bank in sequence. If this bank is busy, the MSU moves on to the following bank and waits for the next bus cycle. If the bank is idle, the MSU looks for a *ready access* that hits the bank's current DRAM page. A ready access refers to an empty position in a read FIFO (ready to be filled with the appropriate data element) or a full position in a write FIFO (the corresponding data element is ready to be written to memory). If such an access exists, the MSU issues it. Otherwise the MSU issues an access for the FIFO requiring the most service (e.g., the emptiest read FIFO).

kernel	operation	vectors

copy	`for (i = 0; i < N; i++)`
	`y[i] = x[i];`
		1 read
		1 write

daxpy	`for (i = 0; i < N; i++)`
	`y[i] = a * x[i] + y[i];`
		1 read
		1 read-write

vaxpy	`for (i = 0; i < N; i++)`
	`y[i] = a[i] * x[i] + y[i];`
		2 read
		1 read-write

Figure 2 Benchmark Algorithms

The simulations we discuss focus on three kernels, the access patterns for which are depicted in Figure 2. *Copy* involves two vectors (two streams); *daxpy* involves two vectors (three streams); and *vaxpy* involves three vectors (four streams). *Vaxpy* denotes a "vector axpy" operation: a vector a times a vector x plus a vector y. Our technical reports explore a larger space, simulating the performance of a suite of access patterns found in real scientific codes: all our experiments indicate that 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% of peak bandwidth.

6. Bounds on Bandwidth

The percentage of peak bandwidth delivered is ultimately determined by the MSU's ability to exploit both fast accesses (in the form of DRAM page hits) and the memory system's concurrency. We can bound SMC performance for any dynamic ordering scheme by calculating the minimum number of page misses for the extreme case when all CPUs share the same working set of DRAM pages throughout the computation. Similarly, we can calculate the minimum time for a processor to execute a loop by adding the startup costs to the time required for the CPU to execute all memory references for the loop. We include versions of these formulas below, but the derivation and analysis of these bounds is given elsewhere [McK95] and is beyond the scope of this paper. The first equation gives asymptotic performance limits for very long vectors, and the others describe limits due to startup effects.

Let f be the depth of the FIFOs, n be the vector length, s and s_r be the total number of streams and the number of read-streams in a computation, respectively, and v be the number of vectors (a vector that is both read and written counts as two streams). N is the number of processors in the system, and we assume that they all participate in the computation. For simplicity in our formulas, we assume that the vector stride is small with respect to DRAM page size and is relatively prime to b, the number of interleaved memory banks.

The page-miss bound calculates the minimum fraction of memory accesses that must generate DRAM page misses, and uses this to compute the average time to complete an access. The bandwidth limit is then obtained by dividing the minimum access time by this average. The fraction of page misses for a multiple-vector computation is bounded by:

$$r = \frac{bN(s-1)(v-1)}{fN^2s^2}$$

If we let t_{ph} and t_{pm} represent the time required to complete accesses that hit and miss the current DRAM page, respectively, then the maximum percentage of peak bandwidth for the computation is:

$$\%\text{ peak bandwidth} = \frac{100t_{ph}}{(r \times t_{pm}) + ((1-r) \times t_{ph})}$$

The startup delay bound equals the minimum time to perform all accesses in the loop (i.e. the vector length times the number of streams) divided by itself plus the number of cycles the last processor must wait to receive all operands for its first iteration. The waiting time is a function of the number of read-streams in the computation, the FIFO depth, and the order in which the MSU fills the FIFOs. When cyclic scheduling is used, this bound is:

$$\%\text{ peak (cyclic)} = \frac{100ns}{(f-1)(s_r-1)t_{ph} + s_r t_{pm} + ns}$$

This equation reflects the fact that all processors share the same working set of DRAM pages throughout the computation. For block scheduled workloads in which the vector portions assigned to different CPUs begin in different DRAM pages, data from the current page only maps to one FIFO. This means that the MSU must finish filling one CPU's FIFO for a particular stream before switching pages to fill another CPU's FIFO for that stream, resulting in a longer overall startup delay. The corresponding equation is:

$$\%\text{ peak (block)} = \frac{100ns}{N((f-1)(s_r-1)t_{ph} + s_r t_{pm}) + ns}$$

SMC performance is governed by these two competing factors: the page-miss model bounds bandwidth between the SMC and memory, whereas the startup-delay model bounds bandwidth between the processors and SMC.

7. Results

Our results are given as a percentage of the system's peak bandwidth, or that necessary to allow each CPU to perform a memory operation each cycle. The vectors used for these experiments are 10,000 and 80,000 elements in length, and are aligned to begin in the same bank. Given the overwhelming similarity of the performance trends for most benchmarks and system configurations, we only discuss highlights of our results. Although it is unlikely that anyone would build an SMC system with a FIFO depth less than the number of banks, we include results for such systems for purposes of comparison.

To demonstrate the difference an SMC makes, Figure 3 illustrates effective bandwidth for our three kernels on 10,000-element vectors. The non-SMC data in Figure 3(a) represents the maximum measured performance using non-caching and caching loads in the natural order of the computation on a 2-bank Intel i860 system with an 8Kbyte write-back cache with 32-byte lines. That for Figure 3(b) represents the performance bounds generated by Moyer's static access ordering software [Moy93]. The percentage of bandwidth exploited in the natural-order (non-SMC) computations is independent of vector length, and is roughly constant for a given ratio of CPUs to memory banks. As the figure illustrates, using the SMC realizes a significant performance improvement over using either non-caching or caching loads and stores in the natural order for these computations. We have no caching performance data for the systems with eight times as many banks as CPUs, but our studies [McK95] indicate that dynamic access ordering consistently out-performs traditional caching for streaming computations. For instance, we expect caches with a line size of eight elements (64 bytes) to be limited to less than 36% of peak bandwidth, or that measured in simulating an SMC with 8-deep FIFOs.

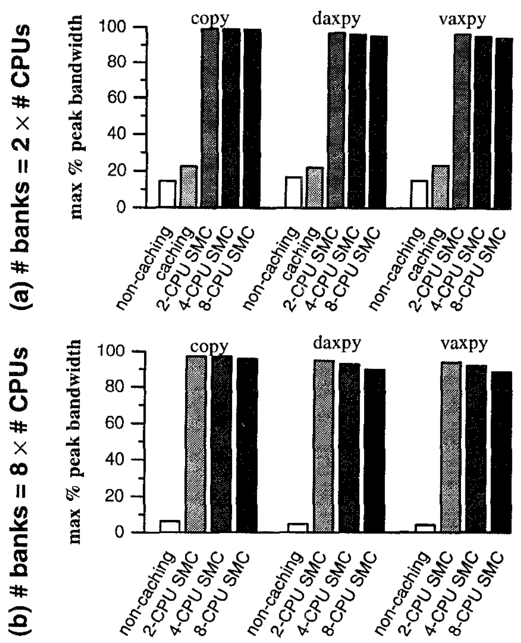

Figure 3 Comparative Performance

In the remaining graphs, the number of banks is kept proportional to the number of CPUs, thus the curves for an 8-CPU system represent performance for four times the number of banks as the corresponding curves for a 2-CPU system. We keep the peak memory bandwidth and DRAM page-miss/page-hit cost ratio constant, so an 8-bank system has four times the DRAM page-miss latency as a 2-bank system. More banks result in fewer total accesses to each, so page-miss costs are amortized over fewer fast accesses. The performance curves for systems with 16 banks thus represent a smaller percentage of a much larger bandwidth compared to those for 2-bank systems.

Recall that cyclic scheduling assigns every N^{th} iteration to each processor, whereas block scheduling breaks the vectors into chunks, assigning each chunk to a different CPU. Figure 4 through Figure 6 present multiprocessor SMC performance for our three kernels using both cyclic and block scheduling on systems with 2, 4, and 8 processors. The top curve in each graph represents the performance limits for the computation: the ascending portion is the page-miss bound for a system in which the number of banks equals the number of processors, and the descending portion is the startup-delay bound. Page-miss bounds for other systems are omitted for readability.

The top row of Figure 4 illustrates effective bandwidth when cyclic scheduling is used to partition the tasks of the *copy* kernel. The first operands from vector x for all CPUs come from the same DRAM page, and the MSU can supply these values right away. Since this kernel only reads one stream, with this partitioning strategy there is no startup delay. Bandwidth for the graphs in the top row is therefore limited only by the page-miss bounds, and with deep FIFOs, all the SMC systems deliver over 95.8% of the peak system bandwidth for these computations. The graphs in all columns are very similar, indicating that performance is relatively constant for a given ratio of processors to banks.

The bottom row of Figure 4 shows *copy* performance when block scheduling is used. The working sets of DRAM pages for each processor tend not to overlap. The MSU is forced to switch pages more often, resulting in lower overall bandwidth. This also creates a longer startup delay: the MSU provides as many operands as it can from the current DRAM page before switching to a new one, and the last CPOU ends up waiting while the MSU fills the other CPUs' FIFOs. Even though block-scheduled computations do not perform as well as cyclic-scheduled ones, the SMC still delivers over 83.5% of peak for deep FIFOs.

The shapes of the performance curves for the 8-CPU systems in the third and fourth columns of Figure 4 are particularly interesting. For shallow FIFOs, the systems with more banks deliver greater bandwidth than those with fewer banks, even though there are fewer vector elements per bank over which the SMC can amortize page-miss costs. The shallowness forces the MSU to switch FIFOs often, causing it to service the FIFOs of all CPUs relatively evenly. This prevents any processor from getting ahead of the others, creating a more even workload for the MSU, and promoting better bank utilization. Unfortunately, the FIFO depths at which this serendipity occurs are difficult to predict: they depend on the number of streams in the computation, the number of CPUs and the degree of page-sharing among them, the DRAM cycle time, and the number of memory banks.

The way the vectors are laid out in memory also affects block-scheduled workloads on systems with many banks. More banks means fewer elements per bank, so the data spans fewer DRAM pages within each bank. Depending on the length of the vector, there may be a greater overlap among the CPUs' working sets of DRAM pages. Data layout accounts for the differences in performance between equivalent systems in the graphs for the 8-CPU systems: in the second row of Figure 4's graphs, the working sets for the systems in the column three overlap more than those in the column four, resulting in better memory performance.

Figure 5 presents *daxpy* performance for both partitioning schemes. These curves illustrate the relationship between FIFO depth and vector length: as the number of processors grows and the amount of data processed by each CPU decreases, performance becomes limited by startup-delay costs for the 10,000-element vectors in columns one through three. For the 80,000-element computations of column four, each CPU has a larger share of data over which to amortize costs, thus the startup-delay bound ceases to be the limiting performance factor. These results emphasize the importance of adjusting the FIFO depth to the computation. Deeper FIFOs do not always result in a higher percentage of peak bandwidth. For good performance, FIFO depth must be adjustable at run-time. Equations for computing optimal FIFO depth can be derived from the SMC performance bounds [McK95].

Figure 6 presents effective bandwidth for the same systems running the *vaxpy* benchmark. The similarity in the shapes of the performance curves for the different benchmarks illustrates the SMC's relative insensitivity to the processors' access patterns in its ability to improve bandwidth. In all cases, asymptotic behavior for long vectors approaches 100% of the peak bandwidth that the memory system can deliver.

8. Conclusions

As processors become faster, memory bandwidth is rapidly becoming the performance bottleneck in the application of high performance microprocessors to important stream-oriented algorithms. These computations lack the temporal locality required for caching alone. Dynamic access ordering, however, can optimize such accesses. Previous papers have shown that by combining compile-time detection of streams with execution-time selection of the access order and issue, we achieve high bandwidth relatively inexpensively.

The results presented here indicate that dynamic access ordering via the SMC can be an effective means of improving memory bandwidth for streaming computations on shared-memory multiprocessor systems. Using only a modest amount of buffer space, the SMC consistently delivers nearly the full system bandwidth for cyclic-scheduled computations on long vectors. SMC performance for block-scheduled parallel computations is not as dramatic, but still represents a significant improvement over performing memory accesses in the natural order of the computation. The superiority of cyclic scheduling over block scheduling is not surprising, since explicit, cooperative management of shared resources has been shown to be an important factor in obtaining good performance on multiprocessor platforms.

In addition, our results emphasize an important consideration in the design of an efficient SMC system that was initially a surprise to us — FIFO depth must be run-time selectable so that the amount of stream buffer space to use can be adapted to individual computations.

Acknowledgments

This work was supported in part by grants from NSF (MIP-9114110 and MIP-9307626) and Intel.

References

[Bae91] J.L. Baer, T.F. Chen, "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty", Proc. Supercomputing'91, November 1991.

[Ben91] M.E. Benitez, J.W. Davidson, "Code Generation for Streaming: An Access/Execute Mechanism", Proc. ASPLOS-IV, April 1991.

[Bur95] D.C. Burger, J.R. Goodman, A. Kagi, "The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors", Univ. of Wisconsin, Department of Computer Science, Technical Report 1261, February 1995.

[Cal91] D. Callahan, K. Kennedy, A. Porterfield, "Software Prefetching", Proc. ASPLOS-IV, April 1991.

[Car89] S. Carr, K. Kennedy, "Blocking Linear Algebra Codes for Memory Hierarchies", Proc. Fourth SIAM Conference on Parallel Processing for Scientific Computing, 1989.

[Chi94] T. Chiueh, "Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops", Proc. Supercomputing '94, November 1994.

[Gao93] Q.S. Gao, "The Chinese Remainder Theorem and the Prime Memory System", Proc. 20th ISCA, May 1993.

[Har89] D.T. Harper, "Address Transformation to Increase Memory Performance", Proc. 1989 International Conference on Supercomputing.

[IEE92] "High-speed DRAMs", Special Report, IEEE Spectrum, vol. 29, no. 10, October 1992.

[Jou90] N. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers", Proc. 17th ISCA, May 1990.

[Lam91]M. Lam, E. Rothberg, M. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", Proc. ASPLOS-IV, April 1991.

[Lee93] K. Lee, "The NAS860 Library User's Manual", NAS TR RND-93-003, NASA Ames Research Center, Moffett Field, CA, March 1993.

[LiN94] Z. Li, T.N. Nguyen, "An Empirical Study of the Work Load Distribution Under Static Scheduling", Proc. Int'l. Conf. on Parallel Processing 1994.

[McK94a]S.A. McKee, "Experimental Implementation of Dynamic Access Ordering", Proc. 27th Hawaii International Conference on Systems Sciences, Maui, HI, January 1994.

[McK94b]S.A. McKee, S.A. Moyer, Wm.A. Wulf, C. Hitchcock, "Increasing Memory Bandwidth for Vector Computations", Proc. Programming Languages and System Architectures, Zurich, Switzerland, March 1994.

[McK95]S.A. McKee, http://www.cs.virginia.edu/~wm/smc.html.

[Mea92] L. Meadows, et.al., "A Vectorizing Software Pipelining Compiler for LIW and Superscalar Architectures", RISC'92.

[Mow92]T.C. Mowry, M. Lam, A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", Proc. ASPLOS-V, September 1992.

[Moy93]S.A. Moyer, "Access Ordering and Effective Memory Bandwidth", Ph.D. Thesis, Department of Computer Science, Univ. of Virginia, Technical Report CS-93-18, April 1993.

[Pal95] S. Palacharla, R.E. Kessler, "Code Restructuring to Exploit Page Mode and Read-Ahead Features of the Cray T3D", work in progress, February 1995.

[Rau91] B.R. Rau, "Pseudo-Randomly Interleaved Memory", Proc. 18th ISCA, Toronto, May 1991.

[Soh91] G. Sohi, M. Franklin, "High Bandwidth Memory Systems for Superscalar Processors", Proc. ASPLOS-IV, April 1991.

[Val92] M. Valero, et. al., "Increasing the Number of Strides for Conflict-Free Vector Access", Proc. 19th ISCA, May 1992.

[Wol89] M. Wolfe, "More Iteration Space Tiling", Proc. Supercomputing '89, 1989.

[Wul95] Wm.A. Wulf, S.A. McKee, "Hitting the Wall: Implications of the Obvious", Comp. Arch.News, 23, 1, March 1994.

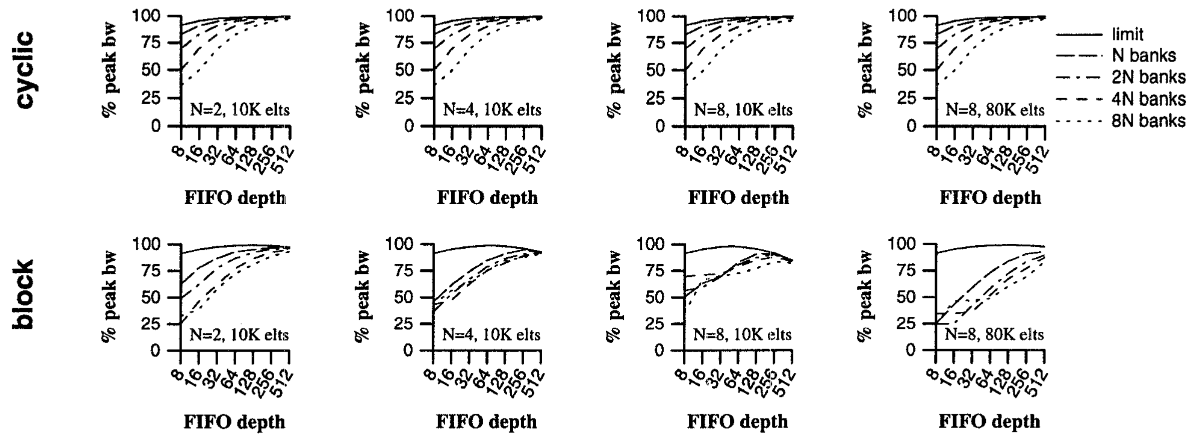

Figure 4 ***copy*** **Performance for SMC Systems with N CPUs**

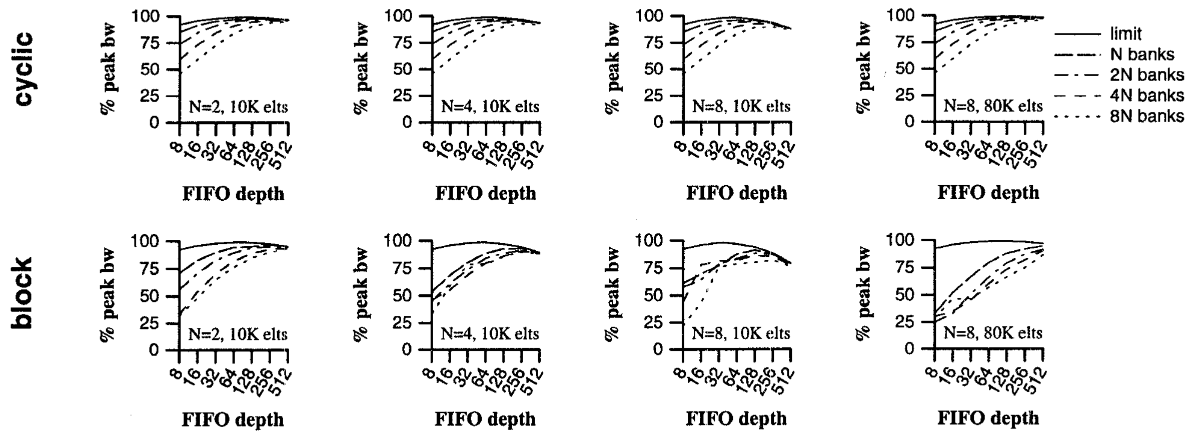

Figure 5 ***daxpy*** **Performance for SMC Systems with N CPUs**

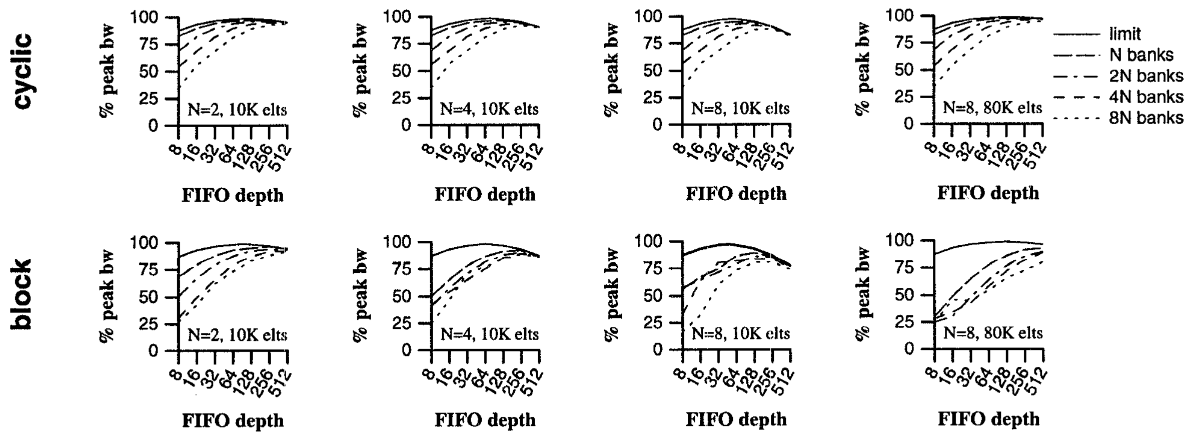

Figure 6 ***vaxpy*** **Performance for SMC Systems with N CPUs**