An Analytic Model of SMC Performance

Sally A. McKee

Computer Science Report No. CS-93-54
November 15, 1993

An Analytic Model of SMC Performance

Sally A. McKee
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
mckee@cs.virginia.edu

Abstract

Memory bandwidth is becoming the limiting performance factor for many applications, particularly scientific computations. *Access ordering* is one technique that can help bridge the processor-memory performance gap. We are part of a team developing a combined hardware/software scheme for implementing access ordering dynamically at run-time. The hardware part of this solution is the *Stream Memory Controller,* or SMC. In order to validate the SMC concept, we have conducted numerous simulation experiments, the results of which are presented elsewhere. Here we develop an analytical model to bound SMC performance, and demonstrate that the simulation behavior of our dynamic access-ordering heuristics approaches that bound.

An Analytic Model of SMC Performance

1. Introduction

The growing disparity between processor speeds and memory speeds is well known [Kat89, Hen90]. Memory bandwidth is becoming the limiting performance factor for many applications — particularly scientific computations — and alleviating this disparity is the subject of much current research.

Access ordering is one technique that can help bridge the processor-memory performance gap. [Moy93] develops and analyzes algorithms to perform access ordering statically at compile time. [McK93a] proposes a combined hardware/software scheme for implementing access ordering dynamically at run-time, and presents numerous simulation results demonstrating its effectiveness. The hardware part of this solution is the *Stream Memory Controller* (SMC) [McK93b]. Here we develop an analytical model to bound SMC performance.

2. Access Ordering

Memory components are usually assumed to require about the same amount of time to access any random location, but this assumption no longer applies to modern memory devices: most components manufactured in the last decade 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 [Qui91]. 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 modern devices offer similar features (e.g. nibble mode, static-column mode, or on-chip SRAM cache) or exhibit novel organizations (e.g. Rambus, Ramlink, or synchronous DRAM). The order of requests strongly affects the performance of all these components.

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.

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.

3. The SMC

[Moy93] develops algorithms and analyzes the performance benefits and limitations of doing compile-time access ordering. The beneficial impact of access ordering on effective memory bandwidth together with the limitations inherent in implementing the technique statically motivate us to consider an implementation that reorders accesses dynamically at run time. What follows is an overview of the architecture proposed in [McK93b, McK93c]: see those documents for more details.

Our discussion is based on the simplified architecture of Figure 1. In this system, memory is interfaced to the processors 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.

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, device characteristics, and current state. 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) contains high-speed buffers for stream operands and provides control registers that the processor uses to specify stream parameters (base address, stride, length, and data size). Together, the MSU and SBU comprise a Stream Memory Controller (SMC) system.

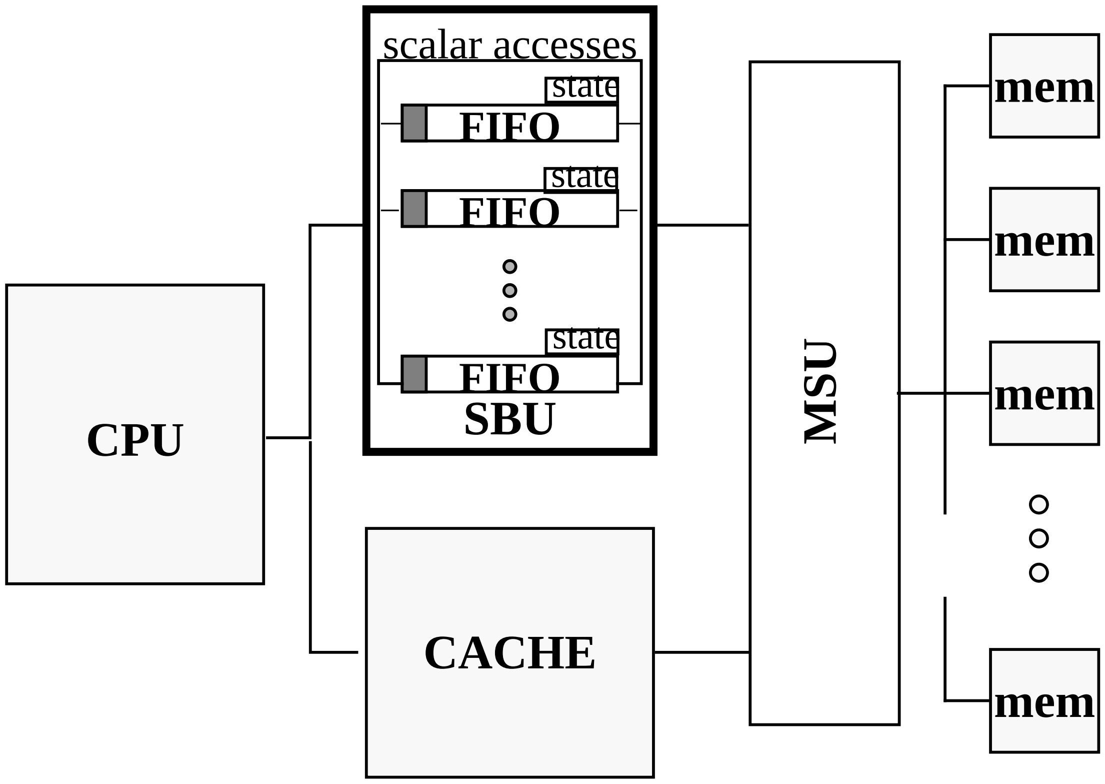


Figure 1 Stream Memory Controller for Uniprocessor System

The stream buffers are implemented logically as a set of FIFOs within the SBU, as illustrated in Figure 1. Each stream is assigned to one FIFO, which is asynchronously filled from (or drained to) memory by the access/issue logic of the MSU. 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.

4. The Analytic Model

Given an SMC system such as described in Section 3, what is the peak memory performance we can expect to achieve? The complex interactions between the many parameters affecting SMC/memory system performance make it difficult to formulate a provably optimal dynamic ordering algorithm. Moreover, implementing such an algorithm might be expensive, both in the amount of hardware necessary and in the time required for it to run. We have instead developed a number of heuristic algorithms for dynamic access ordering; simulation results for these are presented in [McK93a].

Although we do not know precisely what the optimal ordering algorithm is, we *can* bound its performance. Taking advantage of the full bandwidth afforded by the memory system requires exploiting the page-mode capabilities of the memory components. Since bandwidth is limited by the number of page-misses incurred by a computation, we can derive a bound on SMC performance by calculating the minimum number of page-misses for that computation. We can then use this bound to evaluate the performance of our heuristics.

As a practical consideration, we assume that the system is matched so that bandwith between the processor and SMC does not exceed bandwidth between the SMC and memory. External effects (e.g. bus turnaround delays) are ignored. Since we are concerned with exploiting memory component capabilities, and since page-mode is a common feature, we assume that the memory system is composed of interleaved banks of page-mode DRAMS.

In order that the bound we derive be conservative, we model the processor as a generator of non-cached loads and stores of vector elements; all other computation is assumed to be infinitely fast, putting as much stress as possible on the memory system. We assume that DRAM pages are infinitely large, thus we may ignore misses resulting from crossing page boundaries. In essence, we are assuming that these compulsory misses are subsumed by the other misses calculated in our model. Finally, we assume read FIFOs are completely empty and write FIFOs are completely full whenever the SMC begins servicing them, thus allowing the SMC to amortize page miss costs over as many accesses as possible.

We also make a few assumptions to facilitate modeling: the vectors we consider are of equal length, share no DRAM pages in common, and are long enough to make SMC startup costs a negligible portion of the memory time for the computation.

We will refer to a read-only or write-only vector as a *single-access vector.* Likewise, a read-modify-write vector will be referred to as a *double-access vector.* The terms *stream* and *FIFO* will be used interchangeably, since we have assumed an SMC model in which each stream is assigned to exactly one FIFO.

4.1 A Simplistic Performance Model

For the moment, assume a mode of operation in which the MSU and the CPU take turns completely servicing the FIFOs: the MSU fills the read FIFOs and drains the write FIFOs, then waits while the CPU drains the reads and fills the writes. Let us also assume that whenever the CPU takes its turn to access the FIFOs, it does its work instantaneously, placing maximum stress on the memory system. Later we will relax these assumptions to consider what happens when both the processor and the MSU access the FIFOs simultaneously.

In the following, let us assume without loss of generality that we are dealing with read FIFOs, unless otherwise stated; the analysis for write FIFOs is analogous. Let b be the number of interleaved memory banks, and let f be the depth of the FIFOs. If none of the memory banks is on the correct page, then the percentage of accesses that cause DRAM page misses for a single-access vector is b/f. To see this, note that every time the memory system fills the FIFO, it incurs a page miss in *each* memory bank. The number of page misses for a double-access vector is the same as for a single-access vector, for they occupy the same number of pages. The read-modify-write vector is accessed twice as many times and requires two FIFOs, one for the read stream and one for the write stream. For these vectors, the percentage of accesses that cause page misses is $b/2f$.

On the other hand, if the correct DRAM pages for a particular stream are already current in each memory bank when the SMC begins filling a particular FIFO, then none of the accesses to that stream (during this turn) must result in a page miss.

Let v be the number of distinct vectors in the computation, and let s be the number of streams (recall that a single-access vector constitutes one stream, whereas a double-access vector is comprised of two). To calculate the average DRAM page-miss rate for a single FIFO, we amortize the per-vector miss rates over all streams in the computation, i.e. we sum the miss rates for each vector and divide by the number of streams. If none of the memory banks is already on the appropriate page when the SMC begins accessing each vectors, then this average is $p_{miss} = (v/s) \times (b/f)$. But if the first FIFO to be serviced during the current turn was the last to be serviced during the previous turn, then the SMC would not have to pay the DRAM page-miss overhead again. Thus the SMC need not pay the page-miss per bank for *one* vector at each turn. When we exploit this phenomenon, our average page-miss rate becomes:

$$p_{miss} = \frac{(v-1)\,b}{sf}$$

Let *h* be the cost of servicing an access that hits the current DRAM page, and let *m* be the cost of servicing an access that misses the current DRAM page. We can now calculate the percentage of peak bandwidth as:

$$\frac{h}{(p_{miss} \times m) + ((1 - p_{miss}) \times h)} \times 100$$

Note that for a computation involving a single vector, only the first access to each bank generates a DRAM page miss. All remaining accesses will be page hits, since we have assumed that pages are infinitely large. In this case, our model produces a page-miss rate of 0, and the predicted percentage of peak bandwidth is 100. If, in actuality, our page sizes happen to be small, we could more accurately predict performance using $p_{miss} = b/p$, where *p* is the DRAM page size. The effect is that of filling a FIFO of depth *p*.

4.2 Refining the Model for Multiple-Vector Computations

The model presented in Section 4.1 provides an estimate of expected SMC performance, but it is too simple for computations involving more than one vector. A more realistic model must allow the MSU and the CPU to access the SBU concurrently. Recall that the processor is modeled as a generator of loads and stores only. Assume that in its execution of an inner loop, it accesses each active stream (i.e. FIFO) of the computation in round-robin order, dequeueing a single data element each time. If each FIFO access takes a single cycle, and there are *s* streams (FIFOs) involved in the computation, then the CPU will consume a data value of the i^{th} FIFO every *s* cycles. This increases the number of accesses that the memory system can perform before a FIFO becomes full.

If the memory system accesses the FIFOs at the same rate as the processor, then while the MSU is filling a FIFO of depth *f*, the processor will consume f/s more data elements for that stream. While the MSU is supplying those f/s elements, the processor can remove f/s^2 more, and so on. Our equation for calculating the per-vector miss rate becomes:

$$\frac{b}{f(1 + 1/s + 1/s^2 + 1/s^3 + \ldots)}$$

In the limit, the series in the denominator converges to $s/(s-1)$, and our equation reduces to $b(s-1)/fs$. The per-stream average page-miss rate is now:

$$p_{miss} = \frac{b(s-1)(v-1)}{fs^2}$$

and the percentage of peak bandwidth is calculated as before.

5. Simulation Environment

In order to validate the SMC concept, 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. Complete uniprocessor results, including a detailed description of each access-ordering heuristic, can be found in [McK93a]; highlights of these results are presented in [McK93b, McK93c]. Since our concern here is to correlate the performance predictions of our analytic model with our functional simulation results, we present only the maximum percentage of peak bandwidth attained by any order/issue policy simulated for a given memory system and benchmark.

6. Benchmark Suite

Scientific computations are perhaps the most obvious examples of severely bandwidth-limited applications. Caching may provide adequate bandwidth for some, but not all, portions of such programs. The bottlenecks in these computations usually take the form of memory-intensive inner loops, which tend to derive little benefit from caching. Thus we have chosen a suite of benchmark kernels representing access patterns found in real scientific codes. Scalar and instruction references are assumed to hit in the cache, and all stream references use non-caching loads and stores.

Our benchmark suite is depicted in Figure 2. *Daxpy, copy, scale,* and *swap* are from the BLAS (Basic Linear Algebra Subroutines) [Law79, Don79]. These vector and matrix computations occur frequently in scientific applications, thus they have been collected into a set of library routines that are highly optimized for various host architectures. *Hydro* and *tridiag* are the first and fifth Livermore Loops [McM86], a set of kernels culled from important scientific computations. The former is a fragment of a hydrodynamics computation, and the latter is a tridiagonal elimination computation. Although the computations differ, their access patterns are identical, thus results for these benchmarks are presented together. *Vaxpy* is a vector *axpy* computation that occurs in matrix-vector multiplication by diagonals; this algorithm is useful for the diagonally sparse matrices that arise frequently in the solution of parabolic or elliptic partial differential equations by finite element or finite difference methods [Gol93].

Table 1:

copy:	$\forall i$	$y_i \leftarrow x_i$
daxpy:	$\forall i$	$y_i \leftarrow ax_i + y_i$
hydro:	$\forall i$	$x_i \leftarrow q + y_i \times (r \times zx_{i+10} + t \times zx_{i+11})$
scale:	$\forall i$	$x_i \leftarrow ax_i$
swap:	$\forall i$	$tmp \leftarrow y_i \qquad y_i \leftarrow x_i \qquad x_i \leftarrow tmp$
tridiag:	$\forall i$	$x_i \leftarrow z_i \times (y_i - x_{i-1})$
vaxpy:	$\forall i$	$y_i \leftarrow a_i x_i + y_i$

Figure 2 Benchmark Algorithms

Here “axpy” refers to a computation involving some entity *a* times a vector *x* plus a vector *y*. For *daxpy*, *a* is a double-precision scalar, so the computation is effectively a scalar times a vector plus another vector. In the case of *vaxpy, a* is a vector, making the computation a vector times a second vector, plus a third vector.

Note that although these computations do not reuse vector elements, they are often found in the inner loops of algorithms that do, as with *vaxpy* for vector-matrix multiply, and blocked algorithms such as those in the Level 3 BLAS [Don90].

7. Results

Figure 3 through Figure 8 depict performance for stride-one vectors as a function of FIFO depth and available concurrency, i.e. the number of memory banks.. All results are given as a percentage of the system's peak bandwidth (the bandwidth necessary to allow the processor to perform a memory operation each cycle). The vectors used in these simulations are 10,000 doublewords in length. Table 1 through Table 6 summarize the SMC results for each benchmark and give the differences between maximum achievable bandwidth and that delivered in our simulation experiments.

Figure 3 illustrates SMC performance for the *copy* benchmark run on memory systems consisting of one, two, four, and eight banks. The solid lines indicate the achievable performance for a given benchmark, and the dashed lines illustrate the maximum performance attained in our simulations. The dotted lines denote the bandwidth attained by a non-SMC system when accesses are executed in their natural order using non-caching loads and stores.

Performance curves for the analytic and simulated results are strikingly similar: significant differences are apparent only for very shallow FIFOs or a large interleaving factor. Decreasing the FIFO size or increasing the number of memory banks reduces the number of FIFO elements that each bank must service, thus there are fewer accesses over which the SMC may amortize page-miss costs. Under these circumstances unnecessary page misses have a much larger impact on performance, thus we tend to see larger differences between the analytic model and our simulations. As FIFOs grow in depth, performance in all cases approaches 100% of the attainable bandwidth afforded by the system, and there is little or no difference between the analytical and simulation results.

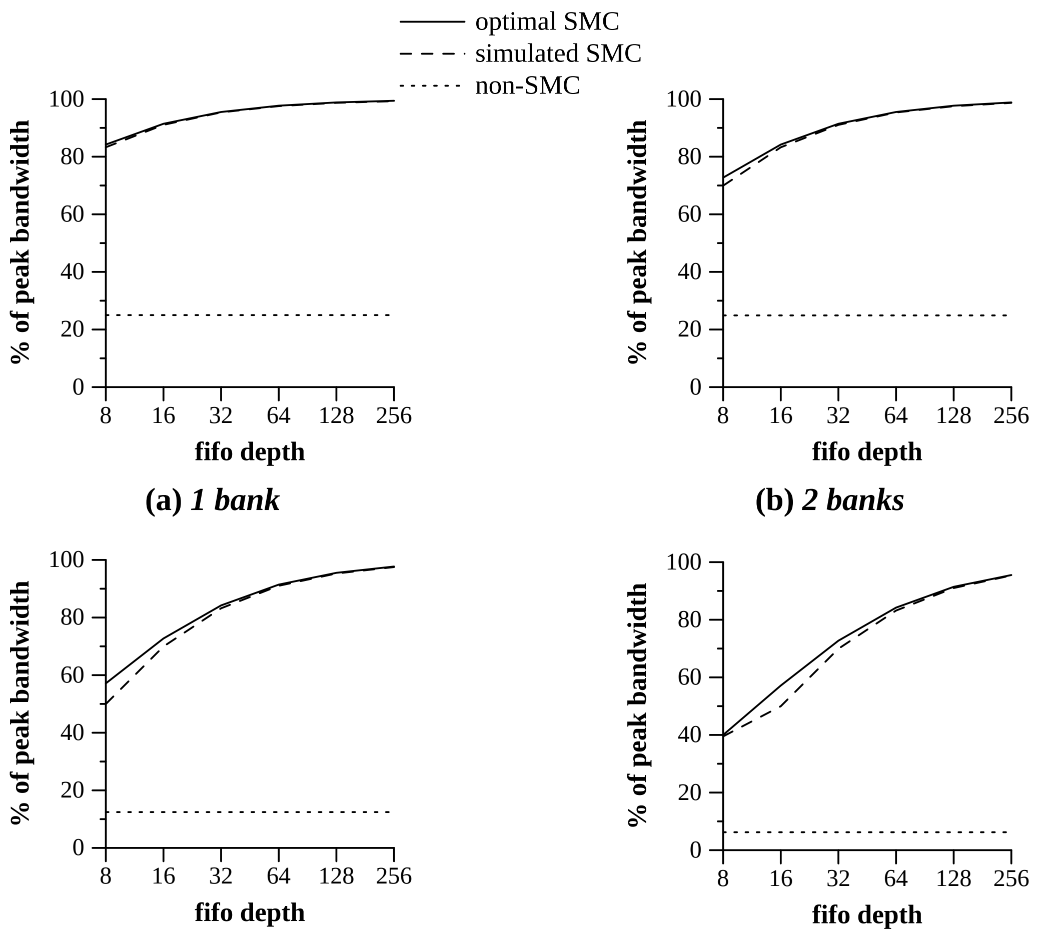


Figure 3 Analytic versus Simulated Performance for *copy*

Note that increasing the number of memory banks reduces relative performance, a somewhat deceptive effect. This is primarily an artifact of our keeping both the peak memory system bandwidth and the DRAM page-miss/hit delay ratio constant. As pictured in Figure 3, the eight-bank system has four times the DRAM page-miss latency of the two-bank system. The percentage of peak bandwidth delivered for the architectures with greater interleaving is smaller, due to the decrease in the number of FIFO positions serviced by each bank, but the system's total bandwidth is really much larger. If we make the opposite assumption and hold the page-miss cycle time of the memory components constant, decrease the page-hit cycle time, and assume a faster bus, the peak bandwidth of the system increases proportionally to the number of banks.

Table 1 recapitulates the data from Figure 3. Note that the maximum achievable SMC performance for a single-bank memory and an SBU with 8-deep FIFOs is the same as for a two-bank memory with 16-deep FIFOs or a four-bank system with 32-deep FIFOs. In other words, maximal performance is constant across a given ratio of FIFO depth to number of memory banks. Similarly, the percentage of peak bandwidth achieved in simulation (and thus the deviation from maximal performance) remains roughly constant for a given depth-to-banks ratio. Thus the SMC results in Figure 3(a) through (d) essentially show different sections of the same curve.

Table 1: Performance Details for *copy*

FIFO depth	Percentage of Peak Bandwidth											
	1 Bank			2 Banks			4 Banks			8 Banks		
	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference
8	84.21	83.28	0.93	72.73	69.94	2.79	57.14	49.98	7.16	40.00	39.49	0.51
16	91.43	91.06	0.37	84.21	83.26	0.95	72.73	69.94	2.79	57.14	49.97	7.17
32	95.52	95.39	0.13	91.43	91.05	0.38	84.21	83.21	1.00	72.73	69.93	2.80
64	97.71	97.57	0.14	95.52	95.36	0.16	91.43	91.03	0.40	84.21	83.19	1.02
128	98.84	98.70	0.14	97.71	97.53	0.18	95.52	95.29	0.23	91.43	91.01	0.42
256	99.42	99.28	0.14	98.84	98.69	0.15	97.71	97.52	0.19	95.52	95.38	0.14

For instance, when each bank is responsible for servicing two FIFO positions, maximal SMC performance exceeds that achieved in simulation by 7.16% for a four-bank memory system and 7.17% for an eight-bank system. When the depth-to-banks ratio is four, simulation results fall short of maximal by 2.79% for both the two- and four-bank systems, and 2.80% in the case of an eight-bank memory. This pattern emphasizes what we have already observed: the SMC's ability to amortize DRAM page-miss costs over a number of accesses that hit the current page decreases as the depth-to-banks ratio diminishes. In other words, *sufficiently deep FIFOs are essential to good performance*

Figure 4 and Table 2 give performance on the *daxpy* benchmark. Performance curves look much the same as for *copy*, exhibiting the same correlation between FIFO depth, number of memory banks, and performance. Non-SMC performance is uniformly low, as it is for all the benchmarks involving multiple vectors, because executing the accesses in their natural order doesn't take advantage of the DRAM's page mode. For each iteration of *daxpy* on a single-bank system, the first access to each of the *x* and *y* vectors incurs a DRAM page miss, but the write to *y* hits the current page. Since the cost of a page miss is four times that of a page hit, the computation takes $4 + 4 + 1 = 9$ cycles per iteration. This is $1/3$ the peak bandwidth, as illustrated by the dashed line in Figure 4(a).

.

Table 2: Performance Details for *daxpy*

FIFO depth	Percentage of Peak Bandwidth											
	1 Bank			2 Banks			4 Banks			8 Banks		
	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference
8	88.89	85.61	3.28	80.00	74.92	5.08	66.67	59.94	6.73	50.00	49.09	0.91
16	94.12	92.17	1.95	88.89	85.58	3.31	80.00	74.87	5.13	66.67	59.90	6.77
32	96.97	95.82	1.15	94.12	92.07	2.05	88.89	85.48	3.41	80.00	74.90	5.10
64	98.46	97.70	0.76	96.97	95.69	1.28	94.12	91.92	2.20	88.89	85.48	3.41
128	99.22	98.59	0.63	98.46	97.55	0.91	96.97	95.50	1.47	94.12	91.91	2.21
256	99.61	98.86	0.75	99.22	98.32	0.91	98.46	97.29	1.17	96.97	95.41	1.56

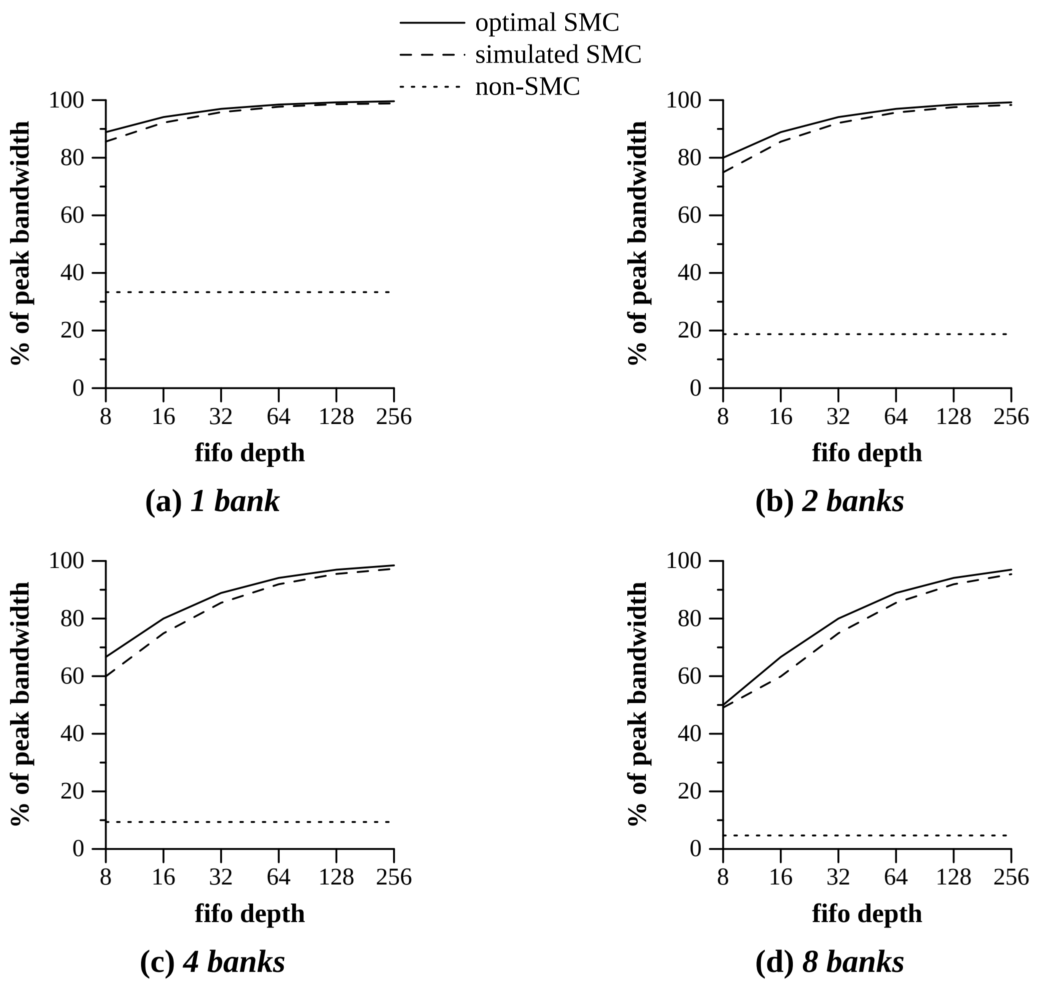


Figure 4 Analytic versus Simulated Performance for *daxpy*

Table 3 through Table 6 give the performance details *hydro*/*tridiag*, *scale*, *swap*, and *vaxpy*. As illustrated in Figure 5 through Figure 8, performance curves for these benchmarks are remarkably similar. The sole exception is the *scale* benchmark, which accesses only one vector. Performance for this kernel is nearly 100% of the peak system bandwidth in all cases, as depicted in Figure 6. In all but one instance (see Table 5 for *swap* using a depth-to-banks ratio of 2), simulation performance differs from the maximum attainable SMC performance by less than 5% of the peak system bandwidth.

.

Table 3: Performance Details for *hydro* and *tridiag*

FIFO depth	Percentage of Peak Bandwidth											
	1 Bank			2 Banks			4 Banks			8 Banks		
	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference
8	80.00	79.94	0.06	66.67	64.90	1.77	50.00	49.95	0.05	33.33	28.57	4.76
16	88.89	88.78	0.11	80.00	79.21	0.79	66.67	64.14	2.53	50.00	49.92	0.08
32	94.12	93.97	0.15	88.89	88.53	0.36	80.00	79.18	0.82	66.67	64.22	2.45
64	96.97	96.75	0.22	94.12	93.85	0.27	88.89	88.50	0.39	80.00	78.98	1.02
128	98.46	98.11	0.35	96.97	96.62	0.35	94.12	93.68	0.44	88.89	88.47	0.42
256	99.22	98.62	0.60	98.46	97.86	0.60	96.97	96.38	0.59	94.12	93.64	0.48

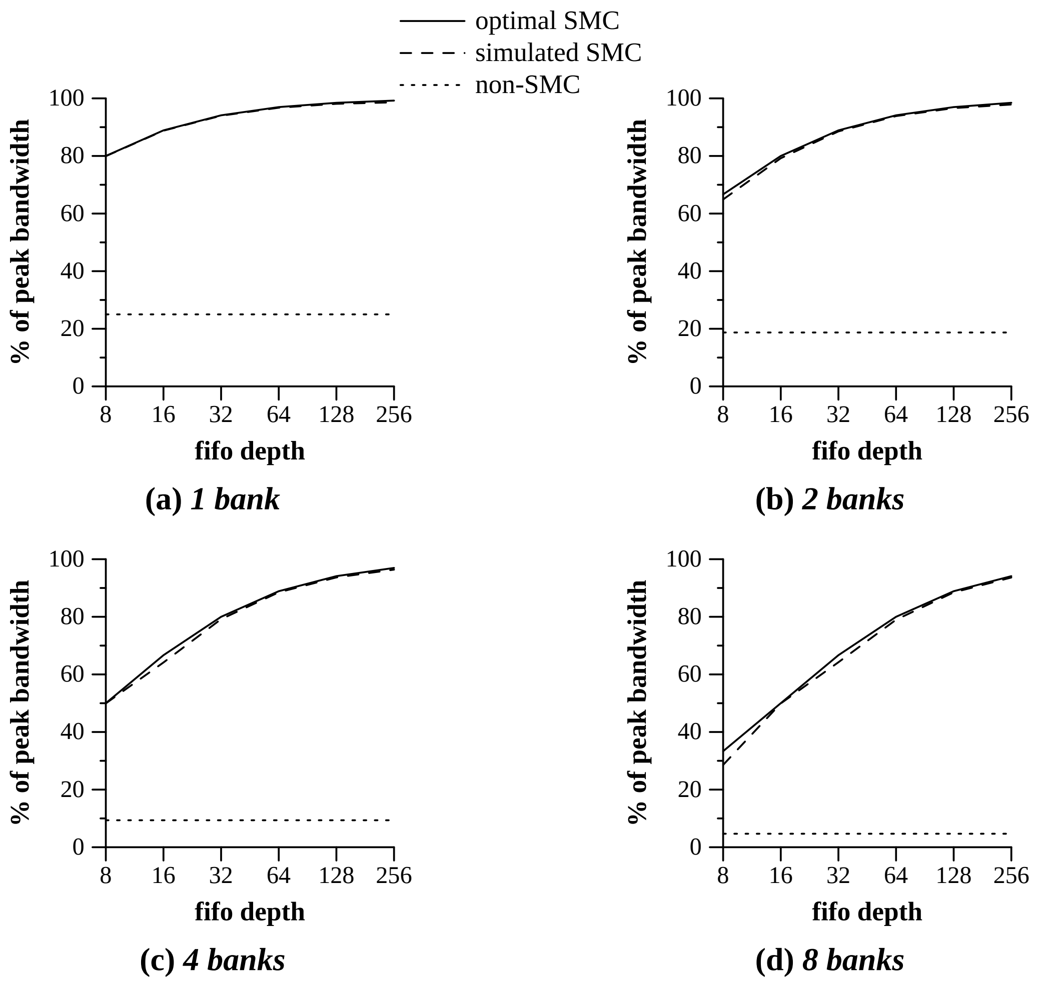


Figure 5 Analytic versus Simulated Performance for *hydro* and *tridiag*

.

Table 4: Performance Details for *scale*

FIFO depth	Percentage of Peak Bandwidth											
	1 Bank			2 Banks			4 Banks			8 Banks		
	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference
8	99.98	99.79	0.19	99.96	99.78	0.18	99.93	99.75	0.18	99.85	93.94	5.91
16	99.98	99.79	0.19	99.96	99.78	0.18	99.93	99.75	0.18	99.85	99.81	0.04
32	99.98	99.79	0.19	99.96	99.78	0.18	99.93	99.75	0.18	99.85	99.81	0.04
64	99.98	99.79	0.19	99.96	99.78	0.18	99.93	99.75	0.18	99.85	99.81	0.04
128	99.98	99.79	0.19	99.96	99.78	0.18	99.93	99.75	0.18	99.85	99.81	0.04
256	99.98	99.79	0.19	99.96	99.78	0.18	99.93	99.75	0.18	99.85	99.81	0.04

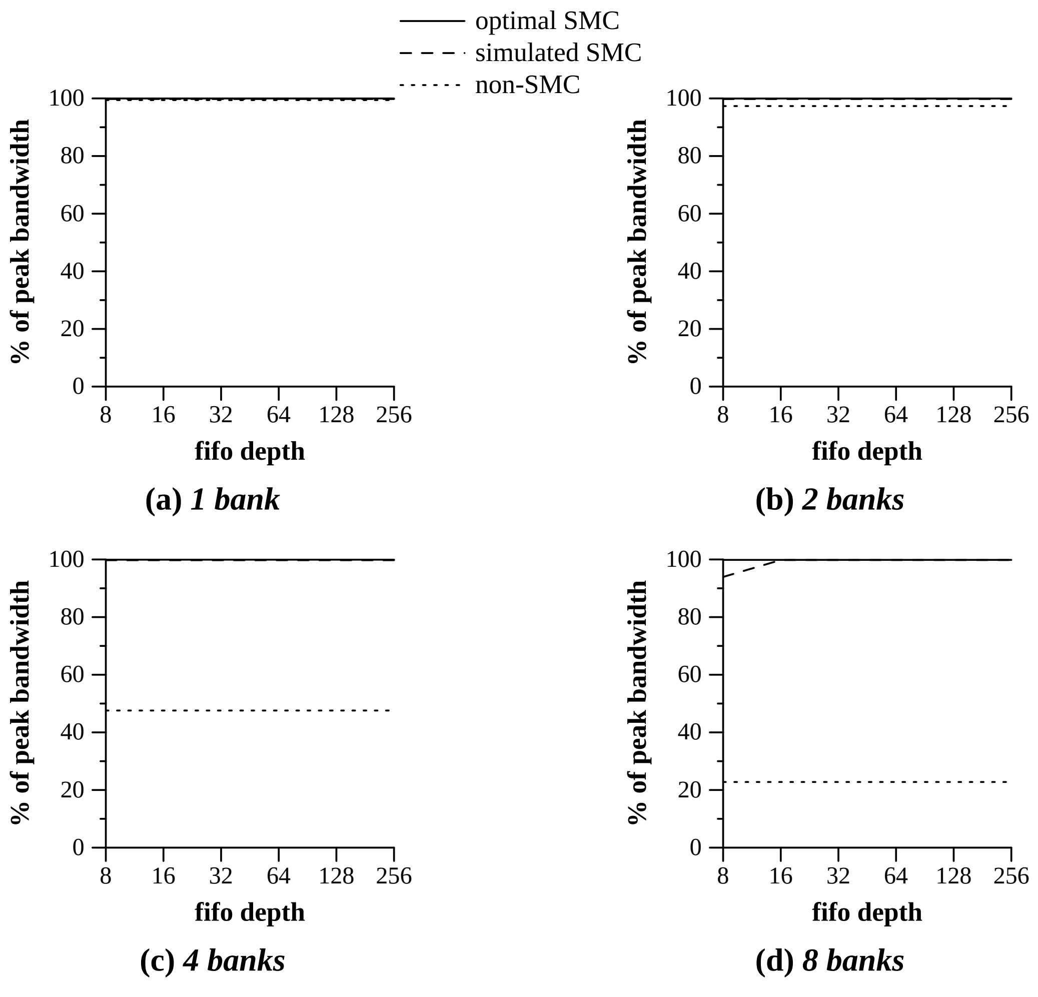


Figure 6 Analytic versus Simulated Performance for *scale*

Table 5: Performance Details for *swap*

FIFO depth	Percentage of Peak Bandwidth											
	1 Bank			2 Banks			4 Banks			8 Banks		
	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference
8	91.43	88.79	2.64	84.21	79.89	4.32	72.73	66.58	6.15	57.14	57.06	0.08
16	95.52	93.99	1.53	91.43	88.72	2.71	84.21	79.84	4.37	72.73	66.56	6.17
32	97.71	96.80	0.91	95.52	93.89	1.63	91.43	89.30	2.13	84.21	79.88	4.33
64	98.84	98.23	0.61	97.71	96.74	0.97	95.52	94.04	1.48	91.43	88.73	2.70
128	99.42	98.90	0.52	98.84	98.08	0.76	97.71	96.94	0.77	95.52	93.95	1.57
256	99.71	99.22	0.49	99.42	98.75	0.67	98.84	98.02	0.82	97.71	96.51	1.20

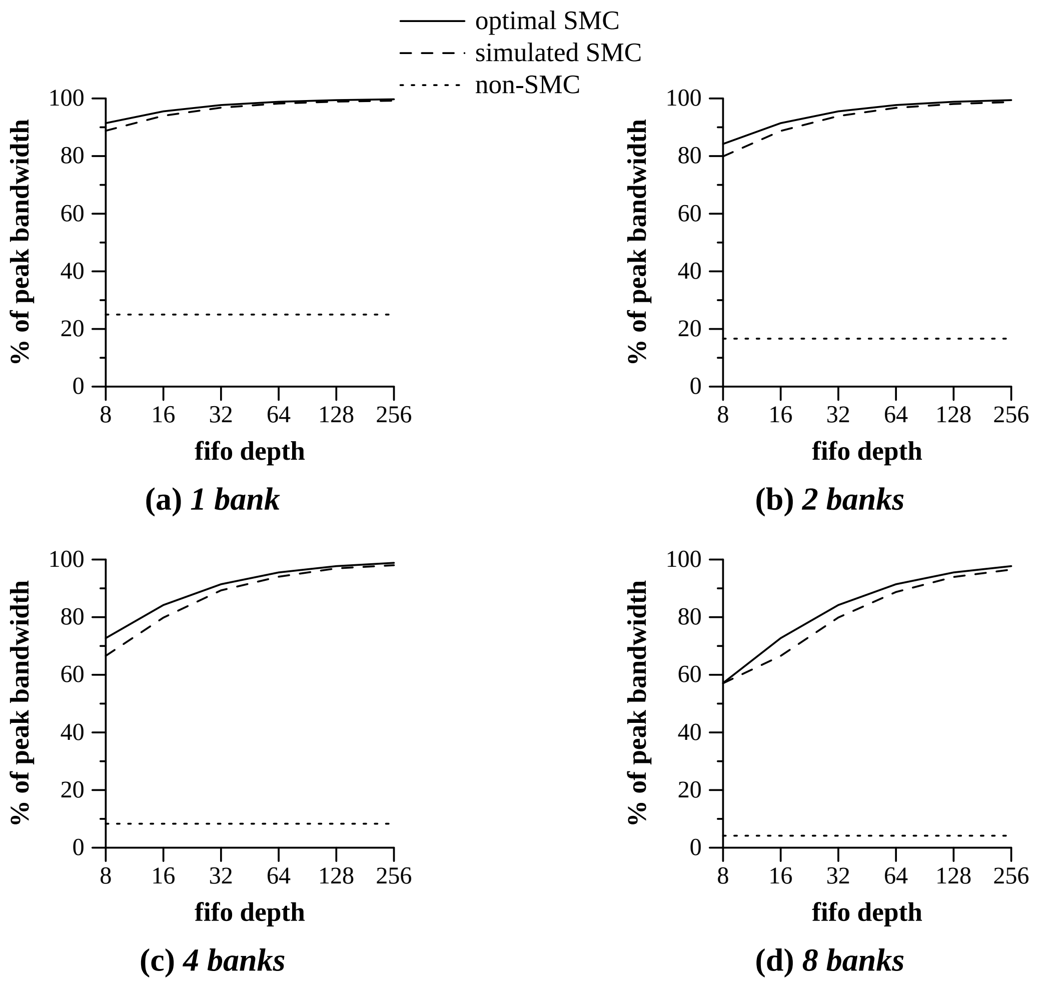
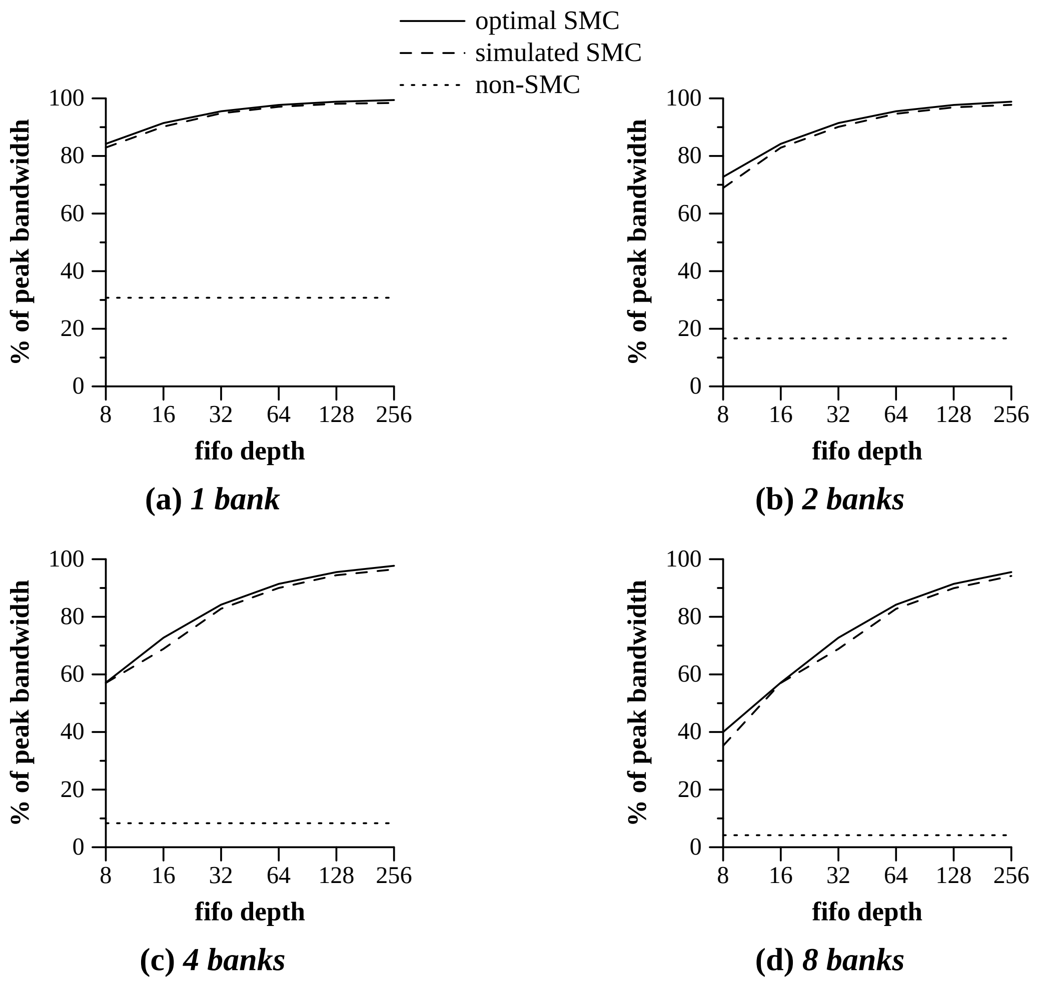


Figure 7 Analytic versus Simulated Performance for *swap*

Table 6: Performance Details for *vaxpy*

FIFO depth	Percentage of Peak Bandwidth											
	1 Bank			2 Banks			4 Banks			8 Banks		
	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference	Maximum	Simulated	Difference
8	84.21	82.92	1.29	72.73	68.89	3.84	57.14	57.05	0.09	40.00	35.23	4.77
16	91.43	90.19	1.24	84.21	82.88	1.33	72.73	68.86	3.87	57.14	57.03	0.11
32	95.52	94.81	0.71	91.43	90.11	1.32	84.21	82.81	1.40	72.73	68.85	3.88
64	97.71	97.08	0.63	95.52	94.65	0.87	91.43	90.01	1.42	84.21	82.76	1.45
128	98.84	98.13	0.71	97.71	96.88	0.83	95.52	94.44	1.08	91.43	89.93	1.50
256	99.42	98.41	1.01	98.84	97.75	1.09	97.71	96.47	1.24	95.52	94.18	1.34



(a) *1 bank* (b) *2 banks* (c) *4 banks* (d) *8 banks*

Figure 8 Analytic versus Simulated Performance for *vaxpy*

8. Conclusions

Here we have presented a bound on maximal SMC performance, and have demonstrated that the simulation behavior of our dynamic access-ordering heuristics approaches that bound. In the worst case, our simulations are delivering 7% or 8% less of the peak system bandwidth than the maximum achievable, and even these small degradations are only for very short FIFOs. For deeper FIFOs, our simulations deliver a percentage of peak bandwidth that is within 1 or 2 of the theoretical maximum attainable for an SMC.

References

[Don79] Dongarra, J.J., et. al., "Linpack User's Guide", SIAM, Philadelphia, 1979.

[Don90] Dongarra, J.J., DuCroz, J., Duff, I., and Hammerling, S., "A set of Level 3 Basic Linear Algebra Subprograms", ACM Trans. Math. Softw., 16:1-17, 1990.

[Gol93] Golub, G., and Ortega, J.M., *Scientific Computation: An Introduction with Parallel Computing*, Academic Press, Inc., 1993.

[Hen90] Hennessy, J., and Patterson, D., "Computer Architecture: A Quantitative Approach", Morgan Kaufmann, San Mateo, CA, 1990.

[IEEE92] "High-speed DRAMs", Special Report, IEEE Spectrum, vol. 29, no. 10, October 1992.

[Kat89] Katz, R., and Hennessy, J., "High Performance Microprocessor Architectures", University of California, Berkeley, Report No. UCB/CSD 89/529, August, 1989.

[Law79] Lawson, et. al., "Basic Linear Algebra Subprograms for Fortran Usage", ACM Trans. Math. Soft., 5, 3, 1979.

[McK93a] 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, August 1993.

[McK93b] McKee, S.A., Klenke, R.H., Schwab, A.J., Wulf, Wm.A., Moyer, S.A., Hitchcock, C., Aylor, J.H., "Experimental Implementation of Dynamic Access Ordering", University of Virginia, TR CS-93-42, August 1993. To appear in Proc. HICSS-27, Maui, HI, January 1994.

[McK93c] McKee, S.A., Moyer, S.A., Wulf, Wm.A., Hitchcock, C., "Increasing

Memory Bandwidth for Vector Computations", University of Virginia, TR CS-93-34, August 1993. To appear in Proc. Conf. on Prog. Lang. and Sys. Arch., Zurich, Switzerland, March 1994.

[McM86] McMahon, F.H., "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range", Lawrence Livermore National Laboratory, UCRL-53745, December 1986.

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

[Qui91] Quinnell, R., "High-speed DRAMs", EDN, May 23, 1991.

[Ram92] "Architectural Overview", Rambus Inc., Mountain View, CA, 1992.