Bounds on Memory Bandwidth in Streamed Computations

Sally A. McKee, Wm. A. Wulf, and Trevor C. Landon

University of Virginia, Charlottesville, VA 22903 USA

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, text searching, and scientific (vector) processing. This paper looks at streaming computations and derives analytic upper bounds on the bandwidth attainable from a class of access reordering schemes. We compare these bounds to the simulated performance of a particular dynamic access ordering scheme, the Stream Memory Controller (SMC). We are building the SMC, and where possible we relate our analytic bounds and simulation data to the simulation performance of the hardware. The results suggest that the SMC can deliver nearly the full attainable bandwidth with relatively modest hardware costs.

1. Introduction

As has become painfully obvious, processor speeds are increasing much faster than memory speeds. To illustrate the current problem, a 300 MHz DEC Alpha can perform 24 instructions in the time to complete a single memory access to a 40ns DRAM.

Those programs that are limited more by bandwidth than by latency are particularly affected by this growing disparity — these include vector (scientific) computations, multi-media (de)compression, encryption, signal processing, text searching, etc. Caching provides adequate bandwidth for portions of such programs, but not for the inner loops that linearly traverse vector-like, "stream" data. Each stream element is visited only once during lengthy portions of the computation, and this lack of temporal locality makes caching less effective than for other parts of the program.

In this paper we develop analytic models that bound the performance of *any* uniprocessor or symmetric multiprocessor memory system on streams. We present highlights of these results, comparing them to the performance of a scheme we have proposed for accessing stream data — the *Stream Memory Controller* (SMC) [McK94a, McK94b]. There are two independent comparisons: a bus-level simulation, and a gate-level simulation of the SMC's VHDL description. Both forms predict the SMC consistently delivers nearly the maximum attainable bandwidth determined by the analytic bounds. While not reported here, preliminary tests of the actual hardware being conducted as this paper is written appear to confirm these results.

The performance of most memory systems is dependent upon the order of the requests presented to it. A multi-bank system, for example, performs better if the accesses permits concurrency among the banks. Order matters at an even lower level too: most memory devices provide special capabilities that make some access sequences faster than others [IEE92,Ram92,Qui91]. For illustration we focus on one such capability, fast-page mode. These devices behave as if implemented with a single on-chip cache

line, or *page*. A memory access falling outside the address range of the current page is significantly slower than repeating accesses to the current page. Bandwidth can be increased by arranging requests to take advantage of such device capabilities.

Access ordering is any technique that changes the order of memory requests to increase bandwidth. Here we are specifically concerned with ordering vector-like *stream* accesses to exploit multi-bank systems using devices with special properties like page-mode. In this paper we buttress our previous results with both analytic models and (a few) gate-level simulations of the SMC being fabricated.

We first present the basic SMC architectures for uniprocessor and shared-memory multiprocessor systems. We then describe our multiprocessor task-scheduling strategy and how it affects memory performance. After explaining the assumptions underlying our analytic performance models and discussing the environment for our simulation experiments, we correlate the analytic performance curves with simulation results.

2. The SMC

There are many ways to approach the bandwidth problem, either in hardware or software. For instance, numerous designs of prefetching hardware have been proposed. These may prefetch into registers, cache, or special buffers [Bae91,Cal91,Chi94,Fu91, Gup91,Jou90,Kla91,Mow92,Skl92,Soh91]. Most of these schemes simply mask latency without increasing effective bandwidth. Such techniques are still useful, but they 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,Los92,Mea92]. Moyer's scheme unrolls loops and groups accesses to each stream, so that the cost of each DRAM page-miss can be amortized over several references to the same page. Lee develops subroutines to mimic Cray instructions on the Intel i860XR [Lee93]. His routine for streaming vector elements reads data in blocks (using non-caching load instructions) and then writes the data to a pre-allocated portion of cache. Meadows describes a similar scheme for the PGI i860 compiler [Mea92], and Loshin and Budge give a general description of the technique [Los92].

Register-level schemes are restricted by the size of the register file, and cache-level schemes potentially suffer from cache conflicts. Moreover, optimal orderings cannot be generated without the address alignment information usually available only at run-time. Nonetheless, these techniques are useful to the extent to which they can be applied. McKee and Wulf examine access-ordering in depth, developing performance bounds for these and other access-ordering schemes [McK95]. The limitations inherent in compile-time techniques motivate us to consider an implementation that reorders accesses dynamically. 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.

Our analysis is based on the simplified architectures of Figure 1 and Figure 2. In these systems, memory is interfaced to the processor through a 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, the MSU provides the same functionality and performance as a traditional memory controller.

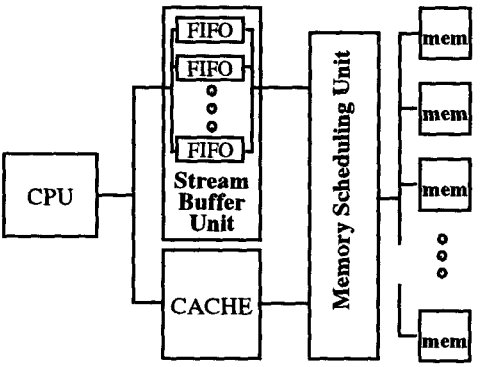


Figure 1 Uniprocessor SMC Organization

The MSU has full knowledge of all streams currently needed by the CPUs: using the base address, 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, such as 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) contains high-speed buffers for stream operands and provides memory-mapped 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.

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.

In the multiprocessor SMC system in Figure 2, all processors are interfaced to memory through a centralized MSU. The architecture is essentially that of the uniprocessor SMC, but with more than one CPU and a corresponding SBU for each. Note that since cache placement does not affect the SMC, the system could consist of a single cache for all processors or separate caches for each. Figure 2 depicts separate caches to emphasize the fact that the SBUs and cache reside at the same logical level of the memory hierarchy.

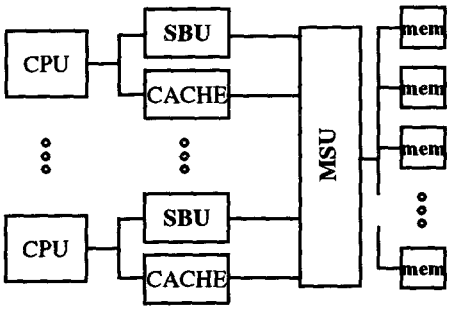
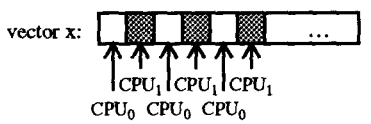


Figure 2 Symmetric Multiprocessor SMC Organization

3. Task Scheduling

The way in which a problem is partitioned for a multiprocessor system can have a marked effect on bandwidth. In particular, SMC performance is affected by whether the working sets of DRAM pages needed by different processors overlap during the course of the computation. If they overlap, the set of FIFOs using data from a page will be larger. With more buffer space devoted to operands from that page, more accesses can be issued to it in succession, resulting in greater bandwidth.

Here we focus on a scheduling model that distributes loop iterations among the CPUs, as in a FORTRAN DOALL. This parallelization scheme makes the *effective stride* at each of the *M* participating CPUs *M* times the original stride of the computation. If the number of memory banks is a multiple of the number of CPUs, this means that a different subset of banks will provide all the data for each CPU. Figure 3 illustrates the data distribution and code for this scheme. Since each of the *M* CPUs performs every *M*th iteration, for stride-1 vectors all processors use the same DRAM pages throughout most of the computation (obviously, if the processors proceed at different rates, some may cross page boundaries slightly sooner than others).



CPU$_0$'s code:
```
for (i = 0; i < L; i += 2) {
    /* operations on x[i]*/
}
```

CPU$_1$'s code:
```
for (i = 1; i < L; i += 2) {
    /* operations on x[i]*/
}
```

Figure 3 Data Distribution for a 2-CPU System

This model of scheduling maximizes the amount of DRAM page sharing, which in turn maximizes the SMC's ability to exploit memory bandwidth. We calculate the attainable bandwidth for an optimal data distribution, thus performance bounds derived for this scheduling model hold for other scheduling techniques. For details of SMC performance under other task-scheduling strategies, see our technical reports [McK94c, McK94d].

4. Modeling Assumptions

We have developed a number of reordering heuristics for the SMC, and we wish to evaluate their effectiveness. That was (and is) the motivation for the bounds derived here; however, even though our discussion is couched in terms of the SMC, our bounds apply to any scheme that performs batched ordering.

For the systems we consider, bandwidth is limited by how many page-misses a computation incurs. This means that we can derive a bound for *any* ordering algorithm by calculating the minimum number of page-misses, and we can use this bound to evaluate the performance of our heuristics. Similarly, we can calculate the minimum time for a processor to execute a loop by adding the minimum time the CPU must wait to receive all the operands for the first iteration to the time required to execute all remaining instructions.

This analysis provides us with two bounds on performance: the first gives asymptotic performance for very long vectors, and the second describes startup effects. The asymptotic model bounds bandwidth between the SMC and memory, whereas the startup-delay model bounds bandwidth between the CPUs and the SMC.

To make these bounds useful we want them to be upper bounds on what any real system can achieve; to that end we impose a number of constraints that real systems will not meet. We ignore bus turnaround delays and other external effects. We model the CPU as a generator of only 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. In calculating the number of page misses incurred by a multiple-stream computation, we assume that DRAM pages are infinitely large. In other words, we assume that misses resulting from crossing page boundaries are subsumed by the other misses calculated in our model. Finally, we derive our performance bounds by assuming that the SMC always amortizes page miss costs over as many accesses as possible: read FIFOs are completely empty and write FIFOs are completely full whenever the SMC begins servicing them.

As a practical consideration, we assume that the system is matched so that bandwidth between the CPUs and SMC equals the bandwidth between the SMC and memory; banks are assumed to be one word wide. The vectors we consider are of equal length and share no DRAM pages in common, and we assume a model of operation in which each CPU accesses its FIFOs in round-robin order, consuming one data item from each FIFO in each iteration. Each of these constraints tends to make the bound more conservative (larger) and hence harder to achieve in practice, but more useful as a yardstick for comparing access mechanisms.

We first look at how SMC startup costs impact overall performance, then we examine the limits of the SMC's ability to amortize page-miss costs as vector length increases asymptotically. We develop each of these models for uniprocessor SMC systems, then extend them to describe multiprocessor SMC performance.

5. Startup-Delay Models

Unlike the traditional performance concern over processor utilization, we focus on memory utilization for stream computations. Nonetheless, good overall performance requires that the processor(s) not be left unnecessarily idle.

Since we assume the bandwidth between the CPU and SMC equals that between the SMC and memory, optimal system performance allows each CPU to complete one memory access each bus cycle. Since the Memory Scheduling Unit attempts to issue as many accesses as possible to the current DRAM pages, most of our access-ordering heuristics tend to fill the currently selected FIFO(s) completely before moving on to service others. At the beginning of a computation on n streams, a CPU will stall waiting for the first element of the nth stream while the MSU fills the FIFOs for the first n-1 streams. By the time the MSU has provided all the operands for the first loop iteration, it will also have prefetched enough data for many future iterations, thus the computation can proceed without stalling the CPU again soon.

Deeper FIFOs cause the CPU to wait longer at startup, but if the vectors in the computation are sufficiently long, these delays are amortized over enough fast accesses to make them insignificant. Unfortunately, short vectors afford fewer accesses over which to amortize startup costs, thus the initial delays can represent a significant portion of the computation time.

To illustrate the problem, consider an SMC with FIFOs of depth f. If we disregard DRAM page misses, the total time for a computation is the time to fetch the first iteration's operands plus the time to finish processing all data. For a computation involving two read streams of length $l = f$, the CPU must wait f cycles (while the first FIFO is being filled) between reading the first operand of the first stream and the first operand of the second stream. According to our model (in which arithmetic and control are assumed to be infinitely fast), the actual processing of the data requires $2f$ cycles, one to read each element in each vector. For this particular system and computation, the time is at best $f + 2f = 3f$ cycles. This is only 66% of the optimal performance of $2f$ cycles (i.e., the minimum time to process all the stream elements). Figure 4 presents a time line of this example: the processor and memory both require the same number of cycles to do their work, but the extent to which their activities overlap determines the time to completion.

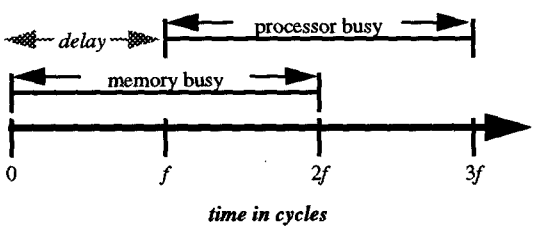


Figure 4 Startup Delay for 2 Read-Streams of Length f

In our analysis, a vector that is only read (or only written) consists of a single stream, whereas a vector that is read, modified, and rewritten constitutes two streams: a read-stream and a write-stream. Let s and s_{read} represent the total number of streams in a computation and the number of read-streams, respectively. The bandwidth limits caused by startup delays can then be described by:

$$\%\ \text{peak bandwidth} = \frac{s \times l \times 100}{f(s_{read} - 1) + (s \times l)} = \frac{s \times 100}{\left(\frac{f}{l}\right)(s_{read} - 1) + s} \quad (1)$$

Figure 5 illustrates these limits as a function of the log of the ratio of FIFO depth to vector length for a uniprocessor SMC system reading two streams and writing one. When vector length equals the FIFO depth ($log(f/l) = 0$), this particular computation can exploit at most 75% of the system bandwidth. In contrast, when the vector length is at least 16 times the FIFO depth ($log(f/l) = -4$), startup delays become insignificant, and attainable bandwidth reaches at least 98% of peak.

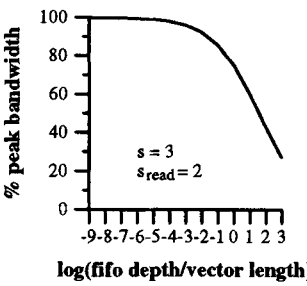


Figure 5 Performance Limits Due to Startup Delays

In a multiprocessor environment, we can bound the performance of the entire parallel computation by first calculating the minimum delay for the last processor to begin its share of the processing, and then adding the minimum time for that CPU to execute its remaining iterations. In developing these formulas, we assume all CPUs are performing the same operation, but are acting on different data. In our multiprocessor formulas, the length l reflects the portion of each vector being processed by a single CPU.

We can derive tighter bounds by tailoring our model to a particular SMC implementation. The way in which the MSU fills the FIFOs affects how long the CPUs must wait to receive the operands for their first iteration. If the MSU's ordering heuristic only services one FIFO at a time, then the last CPU must wait while the MSU fetches the read-streams for all other CPUs and all but one of its own read-streams. On the other hand, if the MSU can service more than one FIFO at a time, more than one CPU can start computing right away.

In the former case, when the MSU only services one FIFO at a time, the minimum number of cycles required to fill that FIFO is $1/N$ times the minimum for a uniprocessor system (because the bandwidth of the system is balanced, and there are now N CPUs that can each execute a memory reference per cycle). Let M represent the number of

CPUs participating in the computation. Then the CPUs are using M/N times the potential bandwidth, and the number of streams that must be fetched before the last CPU can start is $M \times s_{read}$ - 1. The startup-delay formula under these circumstances is:

$$\%\text{peak bandwidth} = \frac{s \times 100}{\left(\frac{1}{N}\right)\left(\frac{f}{l}\right)(Ms_{read} - 1) + s} \times \frac{M}{N} \tag{2}$$

For the latter case, let us assume that the MSU can perform accesses to M FIFOs at a time (one FIFO for each participating CPU). When $M = N$, the formula for startup delays is the same as for the uniprocessor SMC system (Equation 1). To see this, note that each CPU need only wait for all but one of its own read-streams to be fetched, and the average rate at which those FIFOs are filled will be one element per processor cycle. When $M < N$, the average time to fill a FIFO will be M/N times that for a uniprocessor, and the formula becomes:

$$\begin{aligned} \%\text{ peak bandwidth} &= \frac{s \times 100}{\left(\frac{M}{N}\right)\left(\frac{f}{l}\right)(s_{read} - 1) + s} \times \frac{M}{N} \\ &= \frac{s \times 100}{\left(\frac{1}{N}\right)\left(\frac{f}{l}\right)(Ms_{read} - M) + s} \times \frac{M}{N} \end{aligned} \tag{3}$$

The startup delays for the MSU servicing a single FIFO (Equation 2) and multiple FIFOs (Equation 3) differ only by a factor of M - 1 in the first term of the sum in the denominator. Thus Equation 3 also bounds bandwidth when the MSU fills one FIFO at a time; for simplicity, we use it as the basis for comparison with our simulation results.

6. Asymptotic Models

If a computation's vectors are long enough to make startup costs negligible, then the limiting factor becomes the number of fast accesses the SMC can make. The following models calculate the minimum number of DRAM page misses that a computation must incur — first for uniprocessors and then for multiprocessors.

6.1 Uniprocessor Models

The terms *stream* and *FIFO* will be used interchangeably, since each stream is assigned to one FIFO. For simplicity of presentation we refer to read-FIFOs unless otherwise stated; the analysis for write-FIFOs is analogous. We first present a model of small-stride, multiple-vector computations; we then extend this for single-vector or large-stride computations.

Multiple-Vector Computations

Let b be the number of interleaved memory banks, and let f be the depth of the FIFOs. Every time the MSU switches FIFOs, it incurs a page miss in each memory bank, thus the percentage of accesses that cause DRAM page misses is at least b/f for a stream whose stride is relatively prime to the number of banks. Strides not relatively prime to

the number of banks prevent us from exploiting the full system bandwidth since they don't hit all banks. In calculating performance for vectors with these strides we must adjust our formulas to reflect the percentage of banks actually used. We calculate the number of banks used as the total number of banks in the system divided by the greatest common denominator of that total and the vector stride: $b/gcd(b,stride)$. The fraction of accesses that miss the page is thus at least $b/(gcd(b,stride) \times f)$.

Let v be the number of distinct vectors in the computation, and let s be the number of streams (s will be greater than v if some vectors are both read and written). If the CPU accesses the FIFOs (in round robin order) at the same rate as the memory system, then while the MSU is filling a FIFO of depth f, the CPU will consume f/s more data elements from that stream, freeing space in the FIFO. While the MSU supplies f/s more elements, the CPU can remove $f/(s \times s)$, and so on. Thus the equation for calculating the miss rate, r, for single-access vectors is:

$$r = \frac{b}{gcd(b, stride)} \times \frac{1}{f\left(1 + 1/s + 1/s^2 + 1/s^3 + \ldots\right)} \tag{4}$$

In the limit, the series in the denominator converges to $s/(s - 1)$, and our formula reduces to:

$$r = \frac{b(s-1)}{gcd(b, stride) \times fs}.$$

The number of page misses for each vector is the same, but a read-modify-write vector is accessed twice as many times as a read-vector and requires two FIFOs, one for the read-stream and one for the write-stream. Note that for such vectors, using a clever reordering scheme, the *percentage* of accesses that cause page misses is *half* that of a read-vector. To conservatively bound the average DRAM page-miss rate for the entire computation, we amortize the per-vector miss rate over all streams. If we assume that none of the banks is on the correct page when the MSU changes FIFOs, then this average is $R = r \times (v / s)$. But if:

1. the MSU takes turns servicing each FIFO, providing as much service as possible before moving on to service another FIFO;
2) the MSU has filled all the FIFOs and must wait for the CPU to drain them before issuing more accesses; and
3) the first FIFO to be serviced during the next "turn" was the last to be serviced during the previous one,

then the MSU need not pay the DRAM page-miss overhead again at the beginning of the next turn. Thus the MSU may avoid paying the per-bank page-miss overhead for one vector at each turn. When we exploit this phenomenon, our average page-miss rate, R, becomes:

$$R = \frac{v-1}{s} \times r = \frac{v-1}{s} \times \frac{b(s-1)}{gcd(b, stride) \times fs} = \frac{b(s-1)(v-1)}{gcd(b, stride) \times fs^2} \tag{5}$$

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 page. The maximum achievable bandwidth for a computation is equal to the percentage of banks used, thus we must scale our bandwidth formula accordingly, dividing by the greatest common denominator of the total number of banks and the vector stride. The asymptotic bound on percentage of peak bandwidth for the computation is thus:

$$\%\ \text{peak bandwidth} = \frac{h \times 100}{(R \times m) + ((1-R) \times h)} \times \frac{1}{gcd(b, stride)} \tag{6}$$

Single-Vector and Large-Stride Computations

For a computation involving a single vector, only the first access to each bank generates a page miss. If we maintain our assumption that pages are infinitely large, all remaining accesses will hit the current page. In this case, the model produces a page-miss rate of 0, and the predicted percentage of peak bandwidth is 100. We can more accurately bound performance by considering the actual number of data elements in a page and calculating the precise number of page-misses that the computation will incur.

Likewise, for computations involving vectors with large strides, the predominant factor affecting performance is no longer FIFO depth, but how many vector elements reside in a page. The number of elements is the page size divided by the stride of the vector data within the memory bank, and the distance between elements in a given bank is the vector stride divided by the number of banks the vector hits. We refer to this latter value as the *effective intrabank stride*, or *EIS*:

$$\text{EIS} = \frac{stride}{gcd(b, stride)} \tag{7}$$

For example, on a system with two interleaved banks, elements of a stride-two vector have an EIS of 1, and are contiguous within a single bank of memory.

Decreasing DRAM page size and increasing vector stride affect SMC performance in similar ways. Let d be the number of data elements in a DRAM page. Then for computations involving either a single vector or multiple vectors with large EIS values, the average page-miss rate per FIFO is:

$$R = EIS/d \tag{8}$$

For single-vector computations or computations in which EIS/d is less than the FIFO depth, we use Equation 7 instead of Equation 4 to calculate R. The percentage of peak bandwidth is then calculated from Equation 5, as before. Note that neither FIFO depth nor the CPU's pattern of interleaving accesses affects performance for large-stride computations.

6.2 Multiprocessor Extensions

Given the similarity of the memory subsystems for the SMC organizations described in Section 2, we might expect a multiprocessor SMC system to behave much like a

uniprocessor SMC with a large number of FIFOs. For multiprocessor systems, though, some of the assumptions made in the uniprocessor models no longer hold. For instance, we can no longer assume that each read-vector occupies only one FIFO. The distribution of vectors among the FIFOs depends on how the workload is parallelized, since this affects the CPUs' pattern of DRAM page-sharing, which in turn affects performance. We bound multiprocessor SMC performance for all scheduling methods by calculating the minimum number of page misses for the extreme case when *all* CPUs share the same DRAM pages.

Recall that the system is balanced so that if each of N CPUs can consume a data item each cycle, the memory system provides enough bandwidth to perform N fast accesses in each processor cycle. Each CPU can only consume data from its set of FIFOs, while the MSU may arrange for all accesses to be for a single FIFO at a time: this means that the memory system can now fill a FIFO N times faster. Let M be the number of CPUs participating in the computation. When all CPUs use the same DRAM pages, we have essentially distributed each of our s streams over M FIFOs, which is analogous to using a single FIFO of depth $F = M \times f$ for each stream.

As before, we assume a model of computation in which each CPU accesses its FIFOs in round-robin order, consuming one data item from a FIFO at each access. It takes the MSU F/N cycles to supply F items for a stream. During this time, each CPU will consume F/Ns more data elements from this stream, for a total of MF/Ns freed FIFO positions. While the MSU is filling those FIFO positions (in MF/N^2s cycles), the CPU can remove M^2F/N^2s^2 more, and so on. Thus the page-miss rate of a vector is:

$$r = \frac{b}{gcd(b, stride)} \times \frac{1}{F\left(1 + \frac{M}{Ns} + \left(\frac{M}{Ns}\right)^2 + \left(\frac{M}{Ns}\right)^3 + \ldots\right)} \tag{9}$$

The equation for the average page-miss rate is:

$$R = \frac{r(v-1)}{s} = \frac{v-1}{s} \times \frac{b(Ns-M)}{gcd(b, stride) \times FNs} = \frac{b(Ns-M)\ (v-1)}{gcd(b, stride) \times FNs^2} \tag{10}$$

And the percentage of peak bandwidth is computed as in Equation 5:

$$\%\ \text{peak bandwidth} = \left(\frac{h \times 100}{(R \times m) + ((1-R) \times h)} \times \frac{1}{gcd(b, stride)}\right)$$

7. 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 CPUs; number of memory banks; DRAM speed and page size; benchmark kernel; 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

[McK94a,McK94b]. Complete shared-memory multiprocessor results can be found in [McK94c]. Since our concern here is to correlate the performance bounds 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. All simulation results here were generated using DRAM pages of 4K bytes.

Recall that in order to put as much stress as possible on the memory system, we model the processor as a generator of non-cached loads and stores of vector elements. Instruction and scalar data references are assumed to hit in the cache, and all stream references use non-caching loads and stores.

The simulations we discuss here focus on two kernels, the results for which define the ends of the performance spectrum with respect to our set of benchmarks: *scale*, which involves one vector (two streams); and *vaxpy*, which 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 kernels 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 [McK93a,McK94c]. Kernels are chosen, of course, because they are the portion of the applications that perform streamed accesses, which is the focus of this work; total system performance improvements obviously depend upon the fraction of time they spend in these kernels.

8. Comparative Results

All results are given as a percentage of the system's peak bandwidth, the bandwidth necessary to allow each processor to perform a memory operation each cycle. The vectors used for these experiments are 100, and 10,000 doublewords in length. Given the overwhelming similarity of the performance trends for most benchmarks and system configurations, we only discuss highlights of our results here. Although it is unlikely that system designers would build an SMC system with a FIFO depth less than the number of memory banks, we include results for such systems for completeness and for purposes of comparison.

Figure 6 represents the performance of a uniprocessor SMC system with two memory banks, depicting bandwidth as a function of FIFO depth. The graphs on the left show performance for *scale*, and those on the right are for *vaxpy*. The top graphs use 100-element, unit-stride vectors. The bottom graphs use stride-1 vectors of 10,000 elements.

Short vectors hinder the SMC's ability to amortize startup and initial page-miss costs. Even though *scale*'s simulation performance approaches 100% for both vector lengths, the percentage of peak bandwidth delivered for the vectors in Figure 6(a) is slightly lower than for those in Figure 6(c). Performance differences due to vector length are even more pronounced for multiple-vector computations. For the 100-element *vaxpy* computation in Figure 6(b), the startup-delay bound is the limiting performance factor.

Note that performance is constant for FIFO depths greater than the vector length. For longer vectors, as in Figure 6(d), startup-delays cease to impose significant limits to achievable bandwidth, and simulation performance approaches the asymptotic bound of over 97% of peak.

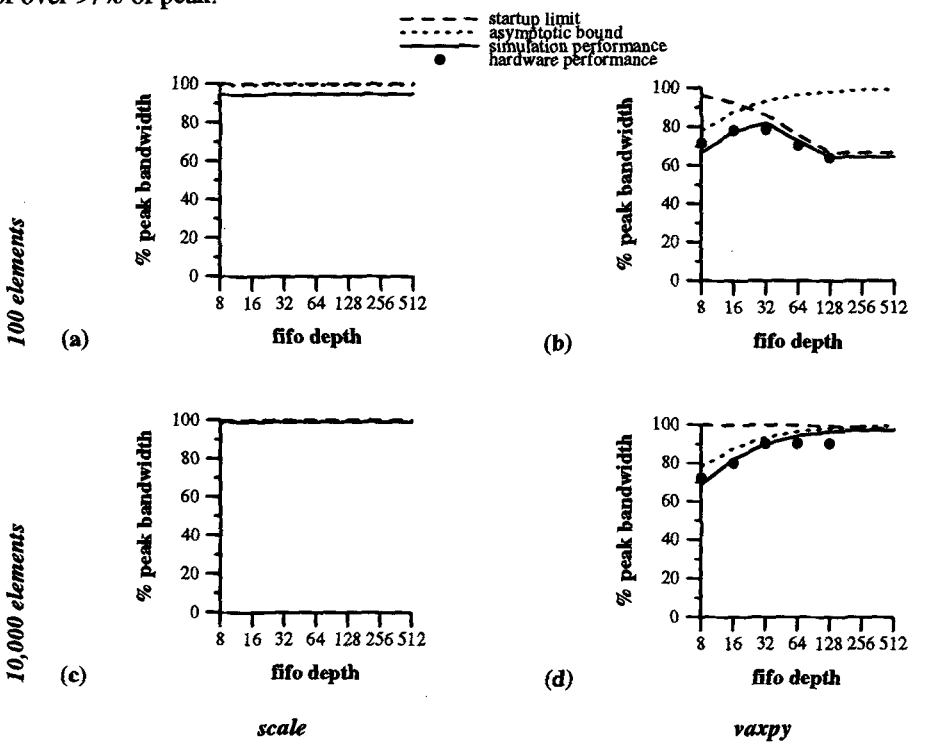


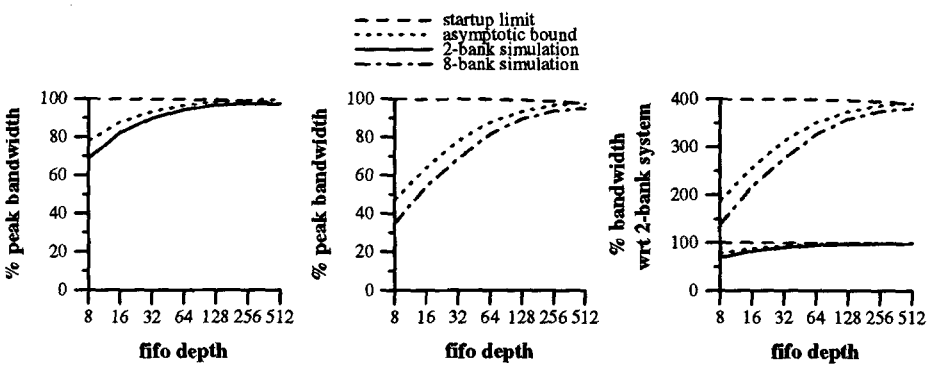
Figure 6 Uniprocessor SMC performance

The hardware data points in Figure 6 were generated via gate-level simulation of our initial implementation. The system parameters of the prototype differ slightly from the systems simulated; in particular, the hardware incurs extra delays (e.g. bus-turnaround) that have been abstracted out of our models, thus performance is limited to about 90% of the system peak. Nonetheless, this data gives us some indication of how actual SMC behavior relates to our models. It is still too early to make definitive claims, but the trends suggested in Figure 6 appear to agree with our other analysis and simulations.[1]

For deep FIFOs, if we increase the number of memory banks, we decrease the number of vector elements in each bank: doubling the number of banks affects performance much like halving the vector length. Alternatively, if the FIFO depth is small relative to the number of banks, increasing the number of banks further behaves like reducing the FIFO depth further since each FIFO holds items from more banks and this reduces the

1. These hardware simulations results are preliminary; we expect to have more data by the time of publication.

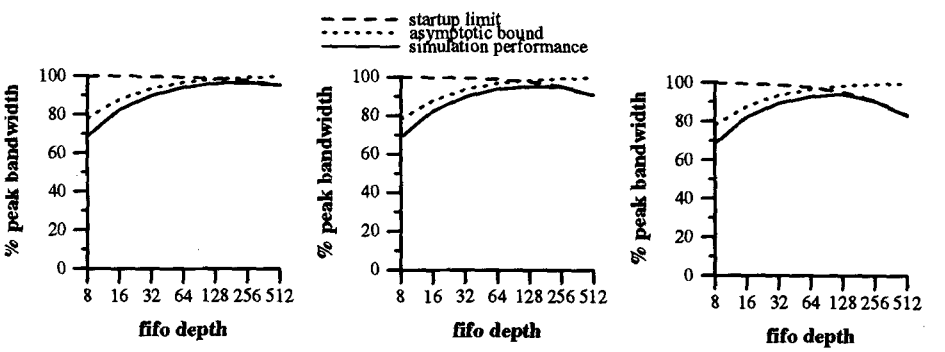
number of items from each DRAM page. Figure 7 demonstrates this phenomenon for stride-one *vaxpy* on 10,000-element vectors on a uniprocessor systems with two and eight banks. Decreasing the number of elements per bank limits the SMC's ability to amortize overhead costs, thus performance for systems with more banks is farther from the asymptotic limits. Note that systems with more banks deliver a smaller portion of a much *greater* bandwidth, as shown in Figure 7(c).



(a) *2 banks* (b) *8 banks* (c) *scaled bandwidth*

Figure 7 Uniprocessor *vaxpy* Performance for Increasing Banks

Figure 8 compares theoretical performance bounds to simulation results for our long-vector *vaxpy* computation on multiprocessor systems with two to eight CPUs. As the number of CPUs grows and the amount of data processed by each CPU decreases, performance becomes more limited by the startup-delay bound. For instance, this bound only begins to dominate performance at FIFO depths 128 and 256 for the 2-CPU system in Figure 8(a), but the crossover point between the startup-delay and the asymptotic bounds is between 64 and 128 for the 8-CPU system in Figure 8(c). All three systems deliver over 94% of peak for an appropriate choice of FIFO depth (in these cases 128).



(a) *2 CPUs, 4 banks* (b) *4 CPUs, 8 banks* (c) *8 CPUs, 16 banks*

Figure 8 Multiprocessor *vaxpy* performance

The graphs in Figure 8 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. Compilers can use the models presented here to calculate the optimal depth.

All examples thus far have used unit-stride vectors, but the same performance limits apply for vectors of any small stride. Figure 9 illustrates simulation results and performance limits for increasing strides on a uniprocessor SMC system with one bank, a FIFO depth of 256, and DRAM pages of 4Kb. We use the large-stride model from Section 6 to compute the asymptotic limits, since for these system parameters and strides, the number of elements in a page is never larger than the FIFO depth. Performance is constant for strides greater than 128, for at these strides only one element resides in any page.

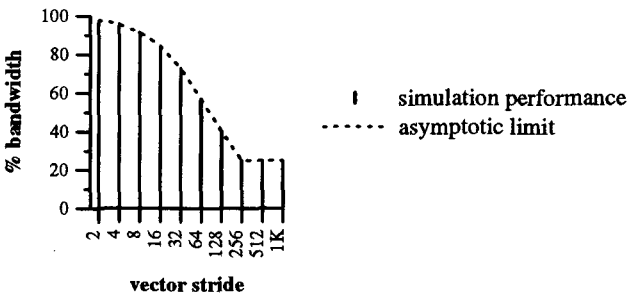


Figure 9 Asymptotic Limits for Increasing Strides

Figure 10 illustrates what happens when not all CPUs participate in a computation. If the MSU's ordering circuitry only services a single FIFO at a time, using fewer CPUs may optimize performance. For instance, by using one fewer CPUs for the task-scheduling scheme described here (in which each of M CPUs performs every Mth loop iteration), the effective stride of the computation becomes relatively prime to the number of memory banks. In such cases, the percentage of peak system bandwidth delivered becomes limited by the percentage of CPUs used, which lowers the startup-delay bound. The graph in Figure 10 shows SMC performance when only three CPUs of a four-CPU system are used to compute *vaxpy* on 10,000 element-vectors.

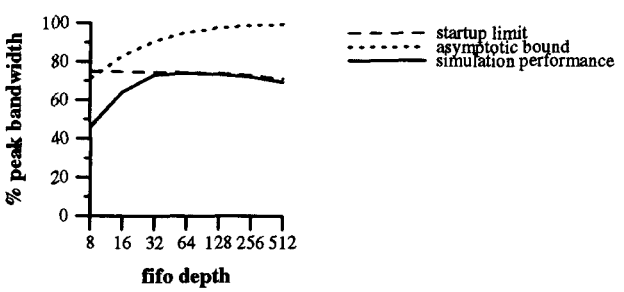


Figure 10 Using Only 3 CPUs of a 4-CPU System with 8 Banks

9. 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, we achieve high bandwidth relatively inexpensively.

Although our previous studies suggested good performance, we did not know how close our heuristic SMC algorithms were to optimal. Here we have described analytic models to bound the performance of both uniprocessor and symmetric multiprocessor SMC systems with memories comprised of multiple banks of page-mode DRAMs. Two different limits govern the percentage of peak bandwidth delivered:

- startup-delay bounds, or the amount of time a processor must wait to receive data for the first iteration of an inner loop; and

  asymptotic bounds, or the number of fast accesses over which the SMC can amortize DRAM page-miss costs.

Our analysis and simulation indicate that for sufficiently long vectors, appropriately deep FIFOs, and any of several selection heuristics, SMC systems can deliver nearly the full attainable memory system bandwidth.

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. Using the equations presented here, compilers can either compute optimal depth (if the vector lengths are known), or they can generate code to perform the calculation at run-time.

Acknowledgments

This work was supported in part by a grant from Intel Supercomputer Division and by NSF grants MIP-9114110 and MIP-9307626. Other members of the SMC team, past and present, are Assaji Aluwihare, Jim Aylor, Alan Batson, Charlie Hitchcock, Bob Klenke, Sean McGee, Steve Moyer, Chris Oliver, Bob Ross, Max Salinas, Andy Schwab, Chenxi Wang, Dee Weikle, and Kenneth Wright.

References

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

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

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

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

[Fu91] Fu, J.W.C., and Patel, J.H., "Data Prefetching in Multiprocessor Vector Cache Memories", Proc. 18th ISCA, May 1991.

[Gup91] Gupta, A., et. al., "Comparative Evaluation of Latency Reducing and Tolerating Techniques", Proc. 18th ISCA, May 1991.

[IEE92] "High-speed DRAMs", Special Report, *IEEE Spectrum*, 29(10), Oct. 1992.

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

[Kla91] Klaiber, A.C., and Levy, H.M., "An Architecture for Software-Controlled Data Prefetching", Proc. 18th ISCA, May 1991.

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

[Los92] Loshin, D., and Budge, D., "Breaking the Memory Bottleneck, Parts 1 & 2", Supercomputing Review, Jan./Feb. 1992.

[McK93a] McKee, S.A, "Hardware Support for Access Ordering: Performance of Some Design Options", Univ. of Virginia, Department of Computer Science, Technical Report CS-93-08, August 1993.

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

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

[McK94c] McKee, S.A., "Dynamic Access Ordering for Symmetric Shared-Memory Multiprocessors", Univ. of Virginia, Technical Report CS-94-14, April 1994.

[McK94d] McKee, S.A., "Dynamic Access Ordering: Bounds on Memory Bandwidth," Univ. of Virginia, Technical Report CS-94-38, Oct. 1994.

[McK95] McKee, S.A., and Wulf, Wm.A., "Access Ordering and Memory-Conscious Cache Utilization", Proc. High Performance Computer Architecture, Jan. 1995.

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

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

[Moy93] Moyer, S.A., "Access Ordering and Effective Memory Bandwidth", Ph.D. Thesis, Department of Computer Science, Univ. 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.

[Skl92] Sklenar, Ivan, "Prefetch Unit for Vector Operation on Scalar Computers", Computer Architecture News, 20(4), Sept. 1992.

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