

# Smarter Memory: Improving Bandwidth for Streamed References

Processor speeds are increasing so much faster than memory speeds that within a decade processors may spend most of their time waiting for data. Most modern DRAM components support modes that make it possible to perform some access sequences more quickly than others. The authors describe how reordering streams can result in better memory performance.

**Sally A. McKee**  
University of Utah

**Robert H. Klenke**  
Virginia Commonwealth University

**Kenneth L. Wright**  
IBM

**William A. Wulf**

**Maximo H. Salinas**

**James H. Aylor**

**Alan P. Batson**  
University of Virginia

Memory speeds have not kept up with processor speeds. More precisely, DRAM latency has not kept pace: Processor speeds have been increasing by at least 70 percent per year, while DRAM latency has improved only 7 percent annually. As a result, a contemporary superscalar 300-MHz DEC Alpha system with a 40-ns DRAM can perform at least 24 instructions in the time it takes to access its memory just once. In a few years, if current trends continue, the number of instructions per access could increase to a thousand.

Fortunately, memory bandwidth is another matter. Wider buses, multiple banks, more pins, the integrated circuit properties of DRAMs (such as static-column mode and on-chip cache), and the newer Rambus and Synchronous DRAM have all contributed to bandwidths that have scaled better than latency. A central problem for memory system designers is how to exploit this bandwidth to achieve lower latencies.

In this article, we describe a technique that can convert more than 90 percent of a memory system's bandwidth into low-latency accesses, at least for a particular class of computations.

The scheme nicely complements traditional caching in two ways: It handles frequently occurring memory reference patterns for which caches do not perform well; and by removing this problematic data from the cache, it makes the cache more effective for the remaining references.

## MEMORY ACCESS PATTERNS

The class of computations on which we focus involves access to *streams* of data. Scientific (vector) computations are one example of streaming, but streams are increasingly common in more general computations—for example, string processing, multimedia codecs, signal processing, and encryption. In all

these examples, each stream is linearly traversed, and each element is visited only a small number of times, often just once. Thus, while such streams have a high degree of spatial locality, they have almost no temporal locality.

Although stream accesses do not exhibit temporal locality, they have the charm of being completely predictable. Stream accesses are typically generated in loops; at the moment just before a loop executes it is possible to know the complete future history of the stream references. We characterize the reference pattern of each stream in terms of *base address* (the address of the first element in the stream), *stride* (the distance between successive stream elements), and *count* (the number of elements in the stream).

## Order-sensitive memory performance

The old notion that DRAM is random access—meaning insensitive to the order of accesses—is not really correct. (See the sidebar “DRAM Implementations.”) Consecutive accesses to the same row in page-mode memory, for example, are faster than random ones. Consecutive accesses that hit different banks in a multi-bank system allow concurrency and hence are faster than ones that hit the same bank. Similarly, for all the other newer memory chips, if the accesses are ordered properly we often get significantly better performance.

## Avoiding cache pollution

The most common method of using bandwidth to reduce latency is to lengthen the line size of the cache, thereby using the memory's bandwidth to fill several cache locations on each access. This technique works in some cases, but it has limitations. For instance, if the stream has a *nonunit stride*—streams in which  $a(i+1)$  doesn't immediately follow  $a(i)$  in memory—the cache will load data that won't be used, which actually reduces the useful bandwidth. Lengthening

## DRAM Implementations

The term DRAM—dynamic random access memory—is slightly misleading: It was coined to indicate that accesses to any random location require about the same amount of time, but most modern DRAMs provide special capabilities that make it possible to perform some access sequences faster than others.

Packaged DRAM chips contain an array of memory cells that store data as charge on capacitors: The DRAM interprets the presence or absence of charge in a capacitor as a binary 1 or 0. Charges must be refreshed periodically—dynamically—to compensate for the capacitors' natural tendency to discharge. The storage arrays are typically square, and each cell is connected to a row line and a column line. To select a bit, the DRAM splits the word address into two parts, with the row address transmitted first, followed by the column address.

DRAM access time is the latency between when a read request is initiated and when the data is available on the memory bus. Cycle time, however, is the minimum time between completion of successive requests. For sustained accesses, cycle time becomes the limiting performance factor. Most DRAM devices load an entire row (or page) of the memory array into a

bank of *sense amplifiers*, which behave much like a line of cache.

Fast-page-mode devices exploit this technique in two ways. Both the row and column addresses must be transmitted for the initial access, but only the column addresses (and accompanying column address strobes) are required for subsequent accesses to the page.

Fast-page mode also takes advantage of the fact that although a certain amount of time is needed to precharge and load the selected page before any particular column can be accessed, the page remains charged long enough for many other columns to be accessed as well.

### Rambus

Rambus is an interesting new bus-based memory technology. The initial DRAMs were capable of delivering a byte of data every 2 ns for a block of information up to 256 bytes. The most recent Rambus DRAMs double the memory channel width and increase its frequency to 400 MHz to yield a bandwidth of 1.6 GBytes per second. Like fast-page-mode DRAMs, Rambus devices use banks of sense amplifier latches to cache data. Unfortunately, these devices offer no performance benefit for random access patterns: The latency for

accesses that miss the two 1-Kbyte cache lines is 150 to 200 ns.

### Other devices

Other devices offer speed-optimizing features or exhibit novel organizations.

- Nibble mode outputs data from locations  $a$  through  $a+3$  after just location  $a$  is addressed.
- Static-column mode does not require a column address strobe each time the column is changed.
- Extended data out (EDO) DRAM adds a latch to the output of a fast-page-mode DRAM's sense amplifiers.
- Enhanced DRAM includes a small amount of SRAM cache on chip.
- Ralink reduces delays by using a ring topology instead of a bus.
- SDRAM designs latch information in and out under the control of the system clock.

The details of these DRAM implementations are not really important here; the key point is that the order of requests strongly affects the performance of all these memory devices. What is needed, then, is a way to take advantage of the faster access times as much as possible.

cache lines, therefore, increases effective bandwidth on unit-stride streams but decreases the cache hit rate for nonstreamed accesses. There is no perfect balance between these two disparate forms of access.

Our scheme is based on these observations; Figure 1 illustrates its main features. To avoid *polluting the cache* (filling the cache with data that will never be used), our method provides a separate buffer storage unit for streamed data; all stream data, and only stream data, use this buffer. To take advantage of the order sensitivity of the memory system, we implemented a scheduling unit that is capable of reordering accesses.

As a program is about to enter a loop that accesses one or more streams, compiler-generated code provides the scheduling unit with the base addresses, the number of elements, and the strides for any streams accessed in the loop body. Armed with this information, the memory scheduling unit (MSU) will reorder the requests so that even though the processor still issues requests to the stream buffer unit (SBU) in the natural order, the order in which associated requests are made to memory will maximize the use of its bandwidth. Finally, given that the polluting stream accesses are no longer affecting the cache, the cache can be designed more optimally for the remaining requests. The combination of the buffer and scheduling unit is called the stream memory controller (SMC).

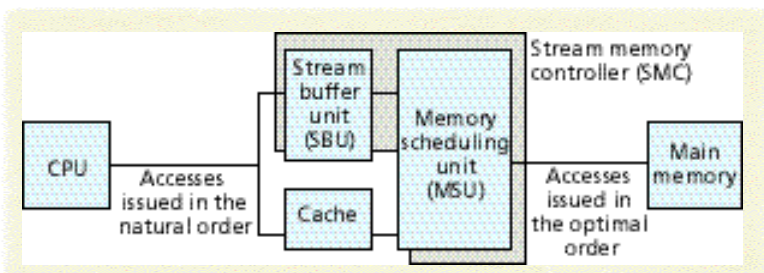


Figure 1. System structure overview.

### The SMC project

The remainder of this article describes our design and implementation. Issues we address include

- whether a compiler can detect streams and generate the appropriate code,
- what the programming model for accessing the buffer memory is,
- whether the buffer and scheduling units can be built so that they are simple and fast and do not slow down nonstream accesses,
- how much storage is needed in the buffer unit, and
- how performance compares with the more traditional cache-based design.

The results we present show that a relatively simple implementation can deliver in excess of 90 percent of

Figure 2. Sample code for dot product, for (a) the MIPS R3000 processor and (b) a hypothetical MIPS extension that includes a stream control unit.

do 10, i=1, 1000 10 s = s + a(i) * b(i)	
(a) MIPS	(b) MIPS with streaming
L63:	sin32i \$s0,\$11,\$9,4
lw \$2,(\$11)	sin32i \$s1,\$13,\$9,4
lw \$3,(\$10)	L63:
mul \$3,\$3,\$2	mul \$3,\$s0,\$s1
addu \$6,\$6,\$3	addu \$6,\$6,\$3
addu \$10,\$10,4	addu \$5,\$5,1
addu \$11,\$11,4	blt \$5,\$9,L63
addu \$5,\$5,1	
blt \$5,\$9,L63	

available memory bandwidth and that codes using our prototype consistently outperform pure cache implementations by factors of at least two to three. Moreover, our mechanism scales: As processor speeds and bandwidth increase further, we can continue to keep memory latencies low for streamed memory references.

### CACHES, VECTOR REGISTERS, AND RELATED ISSUES

The SMC is intended to complement, not replace, a cache. There is an interesting relationship between the SMC's function and the scatter-gather techniques that have been used since the 1960s to disassemble or assemble data items from nonadjacent memory locations.

For example, the vector register file of the Titan minisupercomputer<sup>1</sup> could be filled with data items from nonsuccessive memory locations. The use of an SMC-type device would materially improve the speed of such a system by reordering the memory accesses to provide maximum possible data bandwidth to the scatter-gather hardware.

For similar reasons, the use of an SMC device of the kind described here would significantly improve the performance of the various blocked prefetching schemes<sup>2</sup> by providing the requested data items at the best possible bandwidth from the memory system. (Prefetching initiates the load of data items from memory ahead of when they are needed in the computation.)

### PROGRAMMING MODEL AND COMPILATION

We have tried to heed the principal lesson from RISC design, namely to partition what is done at compile time and what is done at runtime, and particularly to keep runtime operations as simple and as regular as possible. Although there are several possible programming models for the stream buffer unit (SBU), the one we are currently investigating is that of a set of first-in first-out (FIFO) buffers, each managed by a control/status register. Once this register is initialized, the processor merely reads from (or writes to) the head of the queue to read (or write) the next data item in the stream. The act of accessing this location dequeues an input data item or queues an output data item.

From the memory system's perspective, these buffers need not be implemented as FIFOs. The memory scheduling unit (MSU) tries to fill the buffers in an order that maximizes memory bandwidth, which may require accessing the FIFOs' internal storage locations in an arbitrary order. Therefore, from the memory side, the buffers could appear to be a small addressable memory or register file. The MSU we built happens to access the buffers in FIFO order, which yields good performance under most circumstances.

Making the stream buffers behave like FIFOs simplifies the compilation problem. Figure 2 shows the code for dot product generated for the MIPS R3000 processor and a hypothetical MIPS extension that includes a stream control unit.

A fairly conventional optimizing compiler generated the original code, which has had strength-reduction applied to it. That is, the compiler recognizes that the computations of the addresses of a(i) and b(i), which are of the form  $a+i*4$ , actually do not have to perform the multiplication on each iteration. Instead, before the loop, a compiler-generated temporary location is initialized to the base address of the vector and is merely incremented by four bytes on each iteration.

Except for trivial differences, this is the same information needed by the streamed code shown in Figure 2b, which was generated by an experimental compiler built as part of our project.<sup>3</sup> At the point where the optimizer usually initializes a temporary storage location to the base address, the streaming compiler emits code to initialize the FIFO control register—in this case the sin32i (stream in 32-bit integer) instruction.

In those places where the conventional compiler loads a(i) and b(i), the streaming compiler references the head of the stream FIFOs (denoted as \$s0 and \$s1 here). This example demonstrates the feasibility of stream detection; the compilation process is not especially difficult. Using the streaming mechanism reduces the number of instructions in the inner loop, since the CPU no longer needs to compute the array addresses.

Although there is some similarity between streaming and vector load/store operations, compiling for streaming is substantially less complex. In particular, vectorizing compilers must test for a dependency between data generated on one iteration and used in a subsequent one; this dependency analysis is relatively expensive, but isn't needed here.

### AN EXPERIMENTAL IMPLEMENTATION

To demonstrate that the hardware requirements of the approach are reasonable and that it can perform at speed, we implemented two experimental versions in silicon. Each proof of concept is a single application-specific integrated circuit (ASIC) interfaced to an Intel i860 host processor.<sup>4</sup> We designed both versions of our SMC ASIC using VHDL for state machine specification, Mentor Graphics Corporation's Design Architect for

schematic capture, and Cascade Design Automation's Epoch tool for hardware synthesis. We selected the i860 because it was readily available and—more importantly—because it provides load/store instructions that bypass the cache. We use these noncaching instructions to access the stream buffers.

Figure 3 depicts the structure of the prototype system; it consists of an Intel i860 motherboard interfaced via an expansion connector to an SMC daughterboard. The motherboard contains an i860XP processor, a system boot erasable programmable ROM (EPROM), a memory controller optimized for cache-line fills, and 16 Mbytes of page-mode DRAM.

Since we did not have the option of implementing our own general-purpose processor, we were forced to implement the SMC off-chip. The packaging of the prototype is thus somewhat different from that suggested by the conceptual design illustrated in Figure 1, but as Figure 3 shows, the organization is logically similar. Memory references to locations higher than the first 16 Mbytes contained in the i860 motherboard are directed to the SMC daughterboard by the processor. The i860's normal load and store instructions, both cache-directed and cache-bypassing, work as usual.

In our experiments, we kept the actual i860 instructions of the benchmarks in low memory to make use of the cache. We kept all the data for the benchmarks in high memory on the daughterboard. The references to stream elements used noncached load/store i860 instructions via the memory-mapped SMC FIFOs.

Thus, in the prototype, the SMC cannot access the original 16 Mbytes of i860 motherboard memory, but the normal i860 cache and the SMC can both access the daughterboard memory. This limitation is merely an artifact of using the existing i860 board instead of integrating both CPU and SMC on a single chip. In an integrated implementation, all memory would be accessible from both the SMC and the cache.

### Prototype daughterboard

Each bank of the DRAM memory on the SMC daughterboard is composed of two 1 Mbit  $\times$  36 page-mode components with 1-Kbyte pages. These components have a minimum access time of 35 ns for memory locations in the currently open DRAM page. Accesses to locations outside the current page require 110 ns. This three-to-one ratio of page-hit time to page-miss time is typical of modern DRAM components. In the 40-MHz SMC application, page hits require two CPU cycles of 50 ns, while page misses take at least seven CPU cycles. Thus, with two interleaved DRAM memory banks, the SMC can deliver one data item per 25-ns processor cycle at peak bandwidth.

To meet timing and electrical constraints, we added a single-stage, bidirectional pipeline to the daughterboard. This was necessary because the processor takes approximately 14 ns to assert its address and cycle def-

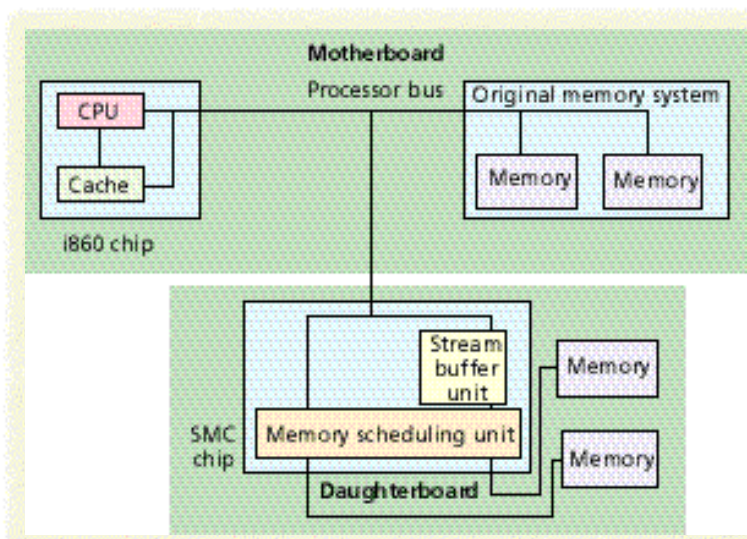


Figure 3. SMC prototype structure.

inition pins, and the signals take another 5 ns to propagate to the expansion connector, leaving less than 6 ns to latch data into or present data from the SMC.

In addition, the electrical specifications for expansion card connections call for signal-line lengths of less than one inch before the first level of logic on the daughterboard. The pipeline component (not shown in Figure 3) either

- latches the address, data, and cycle definition signals from the i860 and presents them to the SMC on the next clock cycle; or
- latches data from the SMC for use by the processor on the next cycle.

This off-chip implementation thus incurs pipeline delays in addition to extra time needed to switch the bus between reading and writing—delays that would not be present in an on-chip SMC. Nonetheless, as we will describe later, the performance of the prototype SMC represents a significant improvement over the performance of a non-SMC system for stream accesses.

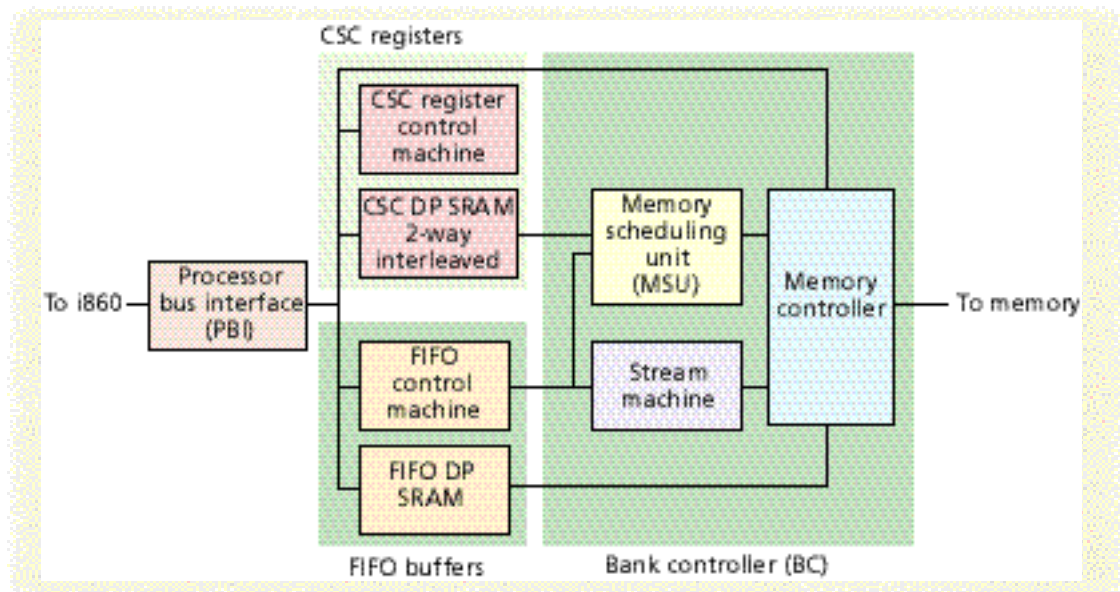
### VLSI implementation

The initial prototype is a 132-pin ASIC implemented in a 0.75-mm, three-level metal HP26B process fabricated through MOSIS, a fabrication service run by the Information Sciences Institute and the University of Southern California (<http://www.isi.edu/mosis>). The 40-MHz i860 host can initiate a bus transaction every other cycle, and *quadword loads* (that manipulate data items four times the word size of the machine) can allow the i860 to read 128 bits of data in two consecutive cycles. To match this, the SMC can deliver 64 bits of data every cycle.

We implemented the initial version of the SMC as a four-way bit-sliced system. We chose this organization over a full 64-bit-wide version because the latter's size would have been dominated by the I/O pads, making it too costly for a proof of concept.

Figure 4 shows the decomposition of each 16-bit SMC ASIC into four logical components:

Figure 4. SMC ASIC architecture.



- processor bus interface (PBI),
- command status/control (CSC) registers,
- FIFO buffers, and
- bank controller (BC).

The PBI state machine—illustrated on the left in Figure 4—provides the logic to interface with the i860 processor bus. The PBI manages the accesses to the CSC registers, the stream accesses to the memory-mapped FIFO heads, and the scalar (nonstream) accesses to the memory subsystem. The CPU transmits the base, length, and stride parameters for each stream by writing the CSC registers, which are implemented with dual-ported SRAM so that the CPU and BC can access them simultaneously.

The FIFOs buffer data between the processor and the memory. The buffer component is composed of two sections:

- dual-ported SRAM buffers used to implement the FIFOs, and
- the FIFO controller state machine used to generate the addresses for the MSU’s accesses to the FIFOs.

The FIFO controller logic provides signals to tell both the BC and the PBI how full or empty each FIFO is. The PBI uses these signals to determine when a given CPU access can be completed, and the BC uses them in deciding which memory access to perform next. The BC logic handles the interface to the interleaved memory system and fills or drains the FIFOs as required. The stream machine section of the BC generates memory addresses for all stream elements. The BC also provides support for scalar accesses to the SMC memory.

The first version of the SMC includes four FIFOs that are 16 64-bit double words deep and can each be set to read or write. We designed and built a second version implementing adjustable-depth FIFOs that are software programmable. In both versions, the MSU implements a simple ordering policy: The BC considers each FIFO

in round-robin order, continually performing accesses until the current FIFO is filled (or drained) before moving on to the next. The BC makes the decision about which FIFO to service next concurrently with memory accesses for the current FIFO. Despite its simplicity, this ordering strategy works well in practice; for uniprocessor systems, simulations demonstrate that its performance is competitive with that of more sophisticated policies. We need more intelligent schemes to achieve good performance on computations involving streams with strides that do not hit all memory banks and on multiprocessor systems in general.<sup>5</sup>

Additional details of the design, implementation, and testing of the initial SMC ASIC and daughter-board can be found on the SMC Web page<sup>6</sup> or in the article “Design and Evaluation of Dynamic Access Ordering Hardware.”<sup>7</sup>

## EXPERIMENTAL RESULTS

For expediency, we chose to build our initial, proof-of-concept hardware with four FIFOs that are each only 16 elements deep. For a production SMC system, the question of how large the FIFOs should be and how many FIFOs should be implemented needs more careful analysis. We employed several kinds of models to refine and evaluate such design decisions:

- Analytic models let us derive performance bounds;
- a behavioral-level, functional simulator made it possible to explore the large SMC design space quickly; and
- gate-level simulation of the hardware description verified that the synthesized chip would meet its specifications.

We used these models to analyze different scenarios. To illustrate how SMC performance changes as system parameters are varied, we present only a small fraction of these results here. We then present detailed measurements for the first SMC design we fabricated.

(Other publications contain more results, along with discussions of how our approach to the memory bandwidth problem relates to those approaches others have taken.<sup>5,7</sup>) To facilitate comparison, we present results for all systems as a percentage of peak system bandwidth, which we define here as the bandwidth required to provide the CPU with one memory access per processor cycle.

The experimental results presented here are for unit-stride vectors of equal length. (The article by Sally McKee and others<sup>7</sup> shows results for nonunit strides and illustrates that SMC performance, relative to cache performance, is even better than for unit-stride vectors.) The vectors share no DRAM pages in common and are aligned to begin in the same bank. We use four benchmark kernels, listed in Table 1. Daxpy and Copy are from the basic linear algebra subroutines,<sup>8</sup> and Tridiag is a tridiagonal Gaussian elimination fragment from the Livermore Fortran Kernels (see [http://www.llnl.gov/asci\\_benchmarks/asci/limited/lfk/README.html](http://www.llnl.gov/asci_benchmarks/asci/limited/lfk/README.html)). Vaxpy denotes a vector operation that occurs in matrix-vector multiplication by diagonals. We are concerned with the access patterns to memory, not with the details of the kernel computations. The access patterns in our kernels are representative of those found in many codes. For instance, Copy uses a common pattern found in string processing and multimedia applications involving JPEG operations.

In all our models, we assume the system is matched so that bandwidth between the processor and SMC equals the bandwidth between the SMC and memory. To put as much stress as possible on the memory system, we assume the processor only generates stream accesses. That is, we assume all computation to be infinitely fast and all instruction and scalar data references to hit in-cache.

The memory systems we model consist of interleaved banks of page-mode DRAMs, where each page is 1 Kbyte and the DRAM page-miss cycle-time is roughly five times that of a page hit. These numbers correspond to the memory parameters of our prototype hardware (facilitating comparison of modeled and measured results), but the performance for other parameter values is not qualitatively different.

### Functional simulation

Figure 5 illustrates the results of our functional simulation. The simulated SMC had four FIFOs, the same number as in the prototype hardware and enough to provide a FIFO for each stream in the benchmarks identified in Table 1. The curves in each graph show the predicted SMC performance as a function of FIFO depth for vectors of length 128 and 8,192 and memories with one, two, four, and eight banks. For deep FIFOs and the long vectors of the graphs (b), (d), (f), and (h), the SMC delivers over 90 percent of the peak system bandwidth for all benchmarks and systems modeled.

Table 1. Benchmark algorithms.

Kernel	Operation	Types of streams
Copy	for (i = 0; i < N; i++) y[i] = x[i];	x: read y: write
Daxpy	for (i = 0; i < N; i++) x[i] = z[i] * (y[i] - x[i-1]);	x: read y: read and write
Tridiag	for (i = 1; i < N; i++) x[i] = z[i] * (y[i] - x[i-1]);	z,y: read x: write <sup>†</sup>
Vaxpy	for (i = 0; i < N; i++) y[i] = a[i] * x[i] + y[i]	a,x: read y: read and write

<sup>†</sup>x[i] is retained in a register for use as x[i-1] on the next iteration.

We built the MSU so that it fills an entire FIFO at a time, which lets it exploit fast-page mode as much as possible, but it also creates a start-up cost for using the SMC. For computations that read more than one stream, the processor must wait for the first element of the last read-stream while the MSU fills the FIFOs for all other read-streams. By the time the MSU has provided all operands for the first loop iteration, it will have fetched data for many future iterations, so the processor won't quickly stall again.

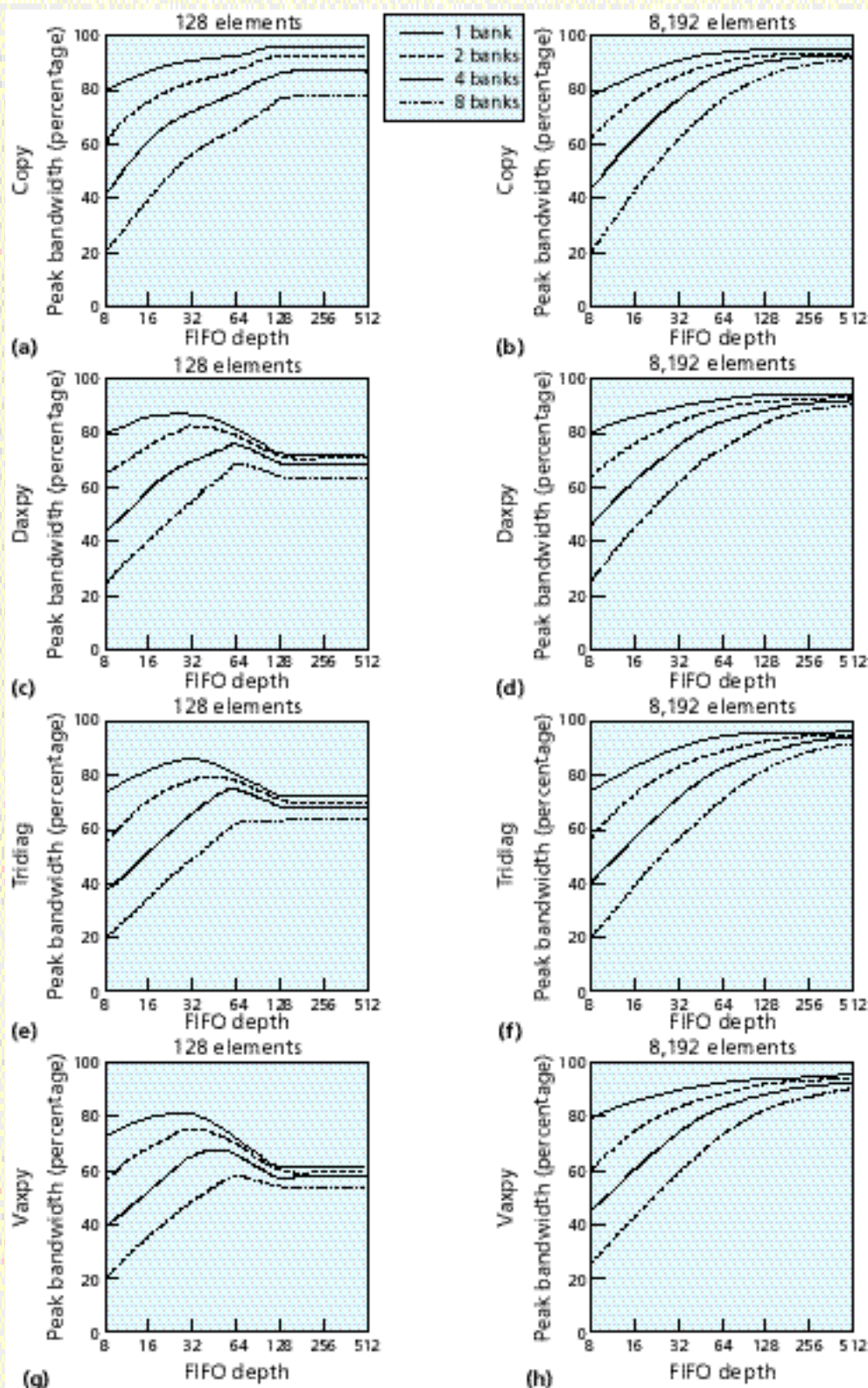
Even though deeper FIFOs allow the MSU to get more data from a DRAM page each time it is made current, they cause the processor to wait longer to complete the first loop iteration. The graphs (a), (c), (e), and (g)—illustrating FIFO depth of 128 in Figure 5—show the net effect of these competing factors. Short-vector computations have fewer total accesses over which to amortize start-up and page-miss costs.

For these computations, initial delays can represent a significant portion of the computation time; this is easy to see in the performance curves for the short-vector computations that read two or more streams (Daxpy, Tridiag, and Vaxpy). The Copy kernel incurs no initial delay, since it only reads one stream. The curves flatten to the right because performance remains constant once the FIFO depth exceeds the vector length. In contrast to the short-vector computations on the left, the longer vector computations in the right column allow start-up and page-miss costs to be amortized much more effectively. For these, initial delays have very little effect on overall performance.

These results emphasize the importance of being able to tailor the FIFO depth to each computation. The second version of the prototype hardware supports FIFOs up to 128 elements and allows the depth to be selected at runtime. We have derived equations that the compiler can use to generate code that selects an appropriate depth.<sup>5</sup>

Note that performance for systems with multiple banks appears to be noticeably lower than that for systems with a single bank. Doubling the number of memory banks halves the number of vector elements in each bank, and having fewer elements in a bank limits the SMC's ability to amortize page-miss and start-up costs (much as short vectors do). To look at

Figure 5. Modeled SMC performance for four FIFOs. These graphs show the predicted SMC performance as a function of FIFO depth for vectors of length 128 and 8,192, and memories with one, two, four, and eight banks. The graphs (a), (c), (e), and (g) illustrate FIFO depth of 128 for the four algorithms, while graphs (b), (d), (f), and (h) illustrate FIFO depth of 8,192.



it another way, if we assume that total system bandwidth scales with the number of banks, then systems with more banks actually deliver a somewhat smaller percentage of a much larger total bandwidth. The net effect is still a significant increase in bandwidth. To

illustrate this, Figure 6a shows SMC performance relative to peak system bandwidth for the Copy benchmark using long vectors on single-bank and eight-bank systems, and Figure 6b illustrates the absolute bandwidths in the two cases.

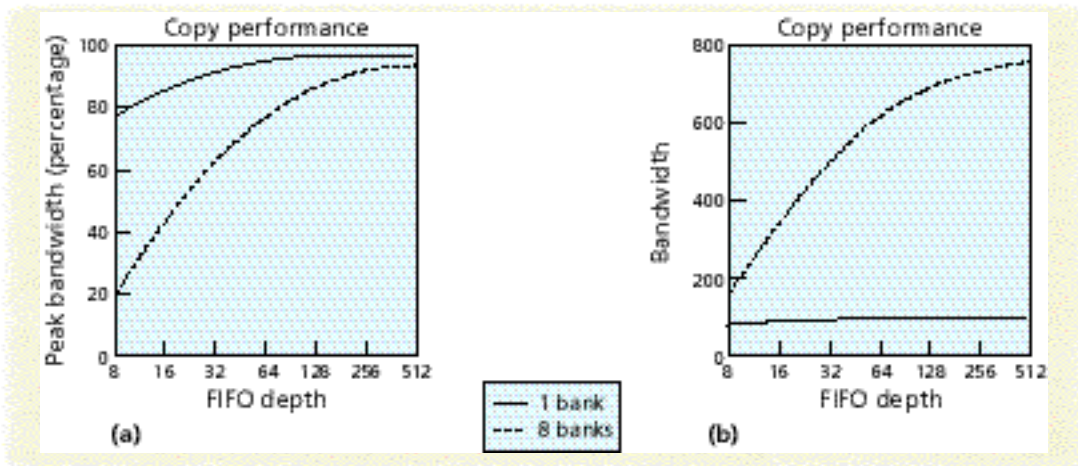


Figure 6. Copy performance for (a) relative and (b) absolute bandwidths.

### Gate-level simulation

In addition to the bus-level, functional simulations described above, we performed a number of hardware gate-level simulations to verify the behavior of our design prior to synthesis. In addition, we augmented the initial SMC hardware description to support the four FIFOs with a programmable depth of from 8 to 128 elements, in powers of two, and simulated its performance for medium and long vectors.

Figure 7 illustrates how the hardware and functional simulation results compare to those of the functional simulator for the Vaxpy kernel on vectors of length 100 and 10,000 and a memory system with two interleaved banks (as in our prototype SMC system). The dashed top line shows the performance bounds defined by the SMC start-up cost and the minimum number of DRAM page misses for this computation. The solid lines again represent our functional simulation results. The dots indicate the results of our gate-level simulations.

The dotted and long-dash lines show performance when using normal caching load and noncaching load instructions to access stream data in the i860 motherboard's cache-optimized memory. These lines represent the maximum effective bandwidth measured for each computation. Obviously, these performances have nothing to do with FIFO depth, but we represent them with lines on these graphs for purposes of comparison.

Performing the computation in the natural order with caching accesses yields only 27 percent of the system's peak bandwidth. In contrast, the effective bandwidth delivered by the SMC for these computations is between 2.4 and 3.3 times that—from 64 to 90 percent of peak. In general, the disparity between SMC and cache performance becomes even more dramatic when performing nonunit stride computations, since each cache-line fill would fetch unneeded data.

### Hardware measurements

We performed numerous experiments on the initial prototype system to evaluate its performance. For these experiments, we carefully coded three versions of each benchmark in i860 assembler:

- one using the SMC,
- one using caching accesses, and

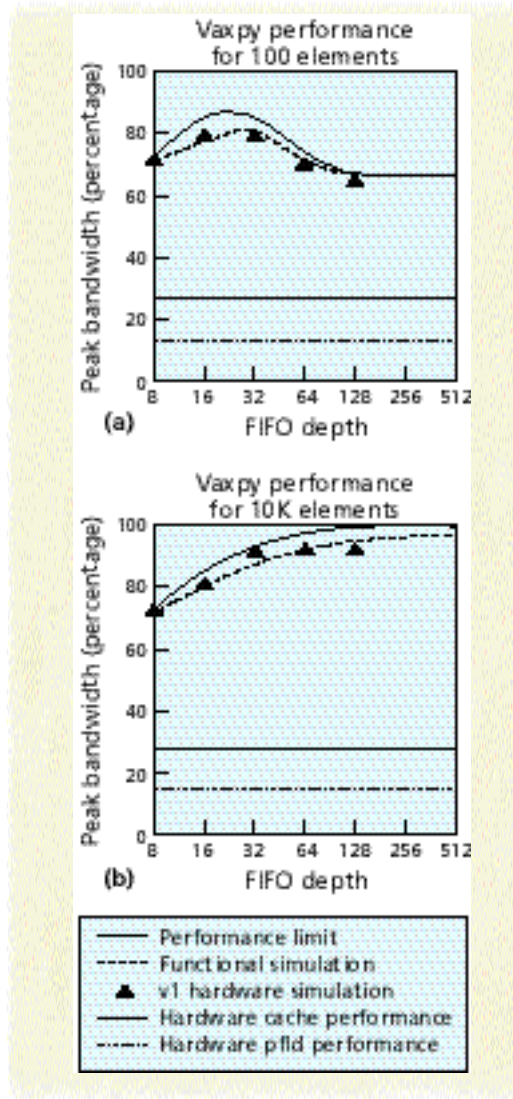
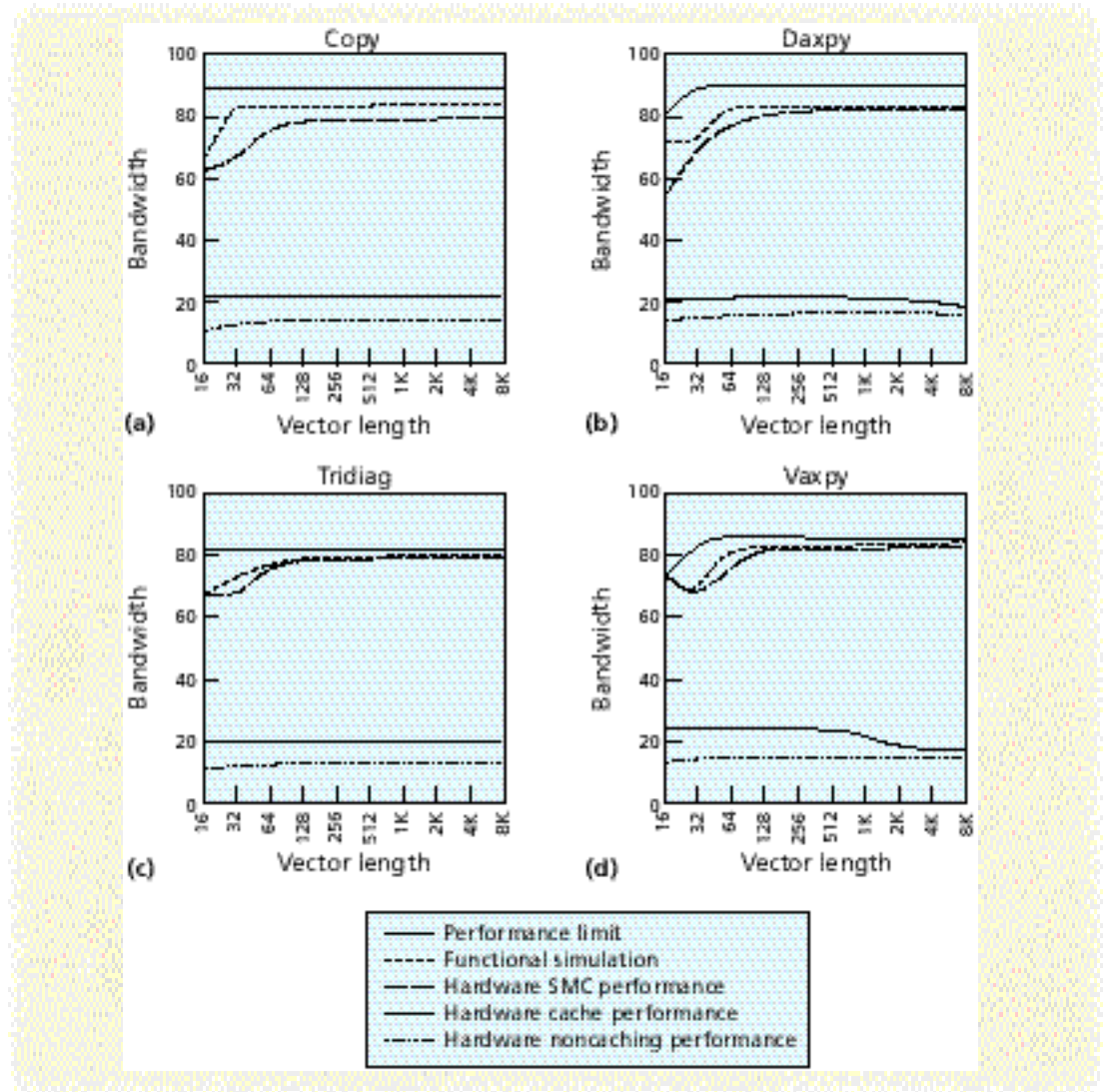


Figure 7. Vaxpy performance for increasing FIFO depth for (a) 100 elements and (b) 10,000 elements.

- one using noncaching accesses without the SMC.

As before, in order to put as much stress as possible on the memory system, we assume arithmetic computation to be abstracted out of each kernel and to be infinitely fast. We execute each loop once before beginning

Figure 8. SMC performance for increasing vector lengths for (a) Copy, (b) Daxpy, (c) Tridiag, and (d) Vaxpy.



our measurements so that the loop instructions will be resident in the i860's cache.

Figure 8 presents these results. Since our initial prototype has a fixed FIFO depth, these graphs illustrate effective memory bandwidth as a function of vector length. For short vectors, it is easy to see the effects of the start-up delay. The agreement between the simulated and measured performance for vectors of 128 or more elements is striking: They differ by 5 percent of peak bandwidth or less. As in Figure 7, the long-dashed curves represent the performance of the i860's cache, and the dotted curves indicate performance when the i860's noncaching, pipelined loads are used.

The patterns of the performance curves for the other benchmarks are similar. Variations in the processor's reference sequence have little effect on the SMC's ability to use bandwidth. Even an SMC with only a small amount of buffer space can consistently deliver over 80 percent of the peak system bandwidth and over 90 percent of the attainable bandwidth. SMCs with deeper FIFOs can exploit nearly the full system bandwidth for long-vector computations.

Matching the performance of the memory system to that of the processor will remain one of the most challenging architectural problems for the near future. Obviously, for those parts of a program that exhibit a high degree of both spatial and temporal locality, the answer lies in bigger, smarter, and multilevel caches.

However, for at least one pattern of accesses that do not exhibit temporal locality, namely streams, we have additional information that should be exploited. For these patterns we have perfect knowledge—at least in principle—of the future. We have shown that a compiler can extract from source code the requisite information about streamed accesses and that relatively modest amounts of buffer storage and control logic can achieve more than 90 percent of the attainable bandwidth, converting that bandwidth into effective low-latency accesses by the processor and yielding overall improvements of a factor of two to three over cache-only schemes. Moreover, the results are relatively independent of benchmark, alignment, and stride. We have not yet measured the improvement in cache performance from removing the polluting

stream references, but there is clearly a positive effect.

We recognize that data consistency is an issue when a particular piece of data could exist in both a stream FIFO and in the cache. This situation can be avoided in a single-processor system, since the compiler can recognize any such data items and cause them to be uniformly loaded as stream elements. The solution is not so straightforward for multiprocessor systems, however, and needs to be addressed. ❖

---

#### Acknowledgments

This work was supported in part by an Intel grant and by National Science Foundation grants MIP-9114110 and MIP-9307626. Assaji Aluwihare, Ben Clark, Charlie Hitchcock, Sung Hong, Trevor Landon, Sean McGee, Mike Nahas, Chris Oliver, Bob Ross, Andy Schwab, Adam Szymkowiak, Chenxi Wang, and Dee Weikle all made significant contributions to the SMC project.

---

#### References

1. T. Diede et al., "The Titan Graphics Supercomputer Architecture," *Computer*, Sept. 1988, pp. 13-30.
2. S.P. VanderWeil and D.J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques," *Computer*, July 1997, pp. 13-30.
3. M.E. Benitez and J.W. Davidson, "Code Generation for Streaming: An Access/Execute Mechanism," *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 132-141.
4. *i860 XP Microprocessor Data Book*, Intel Corp., Santa Clara, Calif., 1991.
5. S.A. McKee, *Maximizing Memory Bandwidth for Streamed Computations*, doctoral thesis, Dept. of Computer Sci., Univ. of Virginia, 1995.
6. SMC Web site, <http://www.cs.virginia.edu/~wm/smc.html>.
7. S.A. McKee et al., "Design and Evaluation of Dynamic Access Ordering Hardware," *Proc. ACM SIGARCH Int'l Conf. Supercomputing*, ACM Press, New York, 1996, pp. 125-132.
8. J.J. Dongarra et al., "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software*, Mar. 1990, pp. 1-17.

**Sally A. McKee** is a research assistant professor of computer science at the University of Utah. Her current interests include the design of efficient, adaptable memory systems, and performance modeling and analysis. McKee received a BA from Yale University, an MSE from Princeton University, and a PhD from the University of Virginia, all in computer science.

**Robert H. Klenke** is an associate professor of electrical engineering at Virginia Commonwealth Univer-

sity. His research interests include system-level modeling, hardware description languages, parallel algorithms for automatic test pattern generation, and high-speed digital design. Klenke received a BS in electrical engineering from the Virginia Military Institute and an MS and PhD in electrical engineering from the University of Virginia.

**Maximo H. Salinas** is a senior scientist at the Center for Semicustom Integrated Systems at the University of Virginia. His current interests include computer architecture, VLSI, digital design methodology development, and microelectromechanical systems (MEMS). Salinas received a BS from MIT and an MS from the University of Virginia, both in electrical engineering. He is a PhD candidate in electrical engineering at the University of Virginia.

**Kenneth L. Wright** is a senior associate engineer at IBM. His current interests include software engineering, system-level hardware verification, and formal verification. Wright received an MS in electrical engineering from the University of Virginia.

**William A. Wulf** is AT&T professor of engineering and applied science in the Department of Computer Science at the University of Virginia. He is presently on leave while serving as president of the National Academy of Engineering. His current interests include national science policy, undergraduate computer science curriculum reform, computer security, hardware-software codesign, and computer architecture and performance analysis. Wulf received a BS in engineering physics, an MS in electrical engineering—both from the University of Illinois at Urbana-Champaign—and a PhD in computer science from the University of Virginia.

**James H. Aylor** is a professor and chairman of the Department of Electrical Engineering and former director of the Center for Semicustom Integrated Systems at the University of Virginia. He is the Associate Editor-in-Chief of *Computer*. His current interests include system-level modeling, concurrent error detection, automatic test pattern generation, hardware description languages, and VLSI system design. Aylor received a BS, an MS, and a PhD in electrical engineering from the University of Virginia.

**Alan P. Batson** is a professor of computer science at the University of Virginia. His current interests include design and performance of computer systems. Batson received a PhD in physics from the University of Birmingham, UK.

Contact McKee at [mckee@cs.utah.edu](mailto:mckee@cs.utah.edu) and the other authors at {klenke, msalinas, wright, wulf, jha, apb}@virginia.edu.