Dynamic Access Ordering for Streamed Computations

Sally A. McKee, *Member*, *IEEE Computer Society*, William A. Wulf, *Fellow*, *IEEE*, James H. Aylor, *Fellow*, *IEEE*, Robert H. Klenke, *Senior Member*, *IEEE*, Maximo H. Salinas, *Member*, *IEEE Computer Society*, Sung I. Hong, and Dee A.B. Weikle, *Member*, *IEEE Computer Society*

Abstract—Memory bandwidth is rapidly becoming the limiting performance factor for many applications, particularly for streaming computations such as scientific vector processing or multimedia (de)compression. Although these computations lack the temporal locality of reference that makes traditional caching schemes effective, they have predictable access patterns. Since most modern DRAM components support modes that make it possible to perform some access sequences faster than others, the predictability of the stream accesses makes it possible to reorder them to get better memory performance. We describe a Stream Memory Controller (SMC) system that combines compile-time detection of streams with execution-time selection of the access order and issue. The SMC effectively prefetches read-streams, buffers write-streams, and reorders the accesses to exploit the existing memory bandwidth as much as possible. Unlike most other hardware prefetching or stream buffer designs, this system does not increase bandwidth requirements. The SMC is practical to implement, using existing compiler technology and requiring only a modest amount of special-purpose hardware. We present simulation results for fast-page mode and Rambus DRAM memory systems and we describe a prototype system with which we have observed performance improvements for inner loops by factors of 13 over traditional access methods.

Index Terms—Memory systems architecture, memory latency, memory bandwidth, memory access ordering, memory access scheduling.

1 Introduction

Processor speeds are increasing much faster than memory speeds, thus memory latency and bandwidth are rapidly becoming the limiting performance factors for many applications. This work addresses the memory bandwidth problem for an important class of applications: those whose inner loops linearly traverse streams of vector-like data, i.e., structured data having a known, fixed displacement between successive elements. Because they execute sustained accesses, these *streamed computations* are limited more by bandwidth than by latency. Examples of these kinds of programs include vector (scientific) computations, multimedia applications, compression and decompression, encryption, signal processing, image processing, text searching, some graphics applications, and DNA sequence matching, to name a few. We often couch our discussion in terms of scientific computation, but our results are applicable to a much wider class of applications.

Caching has long been used to bridge the gap between microprocessor and DRAM performance, but, as the bandwidth problem grows, the effectiveness of the technique is rapidly diminishing [5], [52]. Even if the addition of cache memory is a sufficient solution for general-purpose scalar computing (and even some portions of vector-oriented computations), its general effectiveness for vector processing is questionable. The vectors used in streamed computations are normally too large to cache and each element is visited only once during lengthy portions of the computation. This lack of temporal locality of reference makes caching less effective than it might be for other parts of the program. In addition to traditional caching, other proposed solutions to the memory bandwidth problem range from software prefetching and iteration space tiling, to prefetching or nonblocking caches, unusual memory systems (such as those with prime [23] or pseudorandom [46] interleavings), and address transformations (such as skewing [24]). These solutions generally presume that memory components require about the same time to access any random location, an assumption that does not hold for modern DRAMs. Memory systems can be made more efficient by exploiting the modes and features of current DRAM devices.

- *S.A. McKee is with the School of Computing, University of Utah, 50 S. Central Campus Dr. #3190, Salt Lake City, UT 84112. E-mail: sam@cs.utah.edu.*
- *W.A. Wulf and D.A.B. Weikle are with the Department of Computer Science, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA 22903-2442. E-mail: {wulf, daw4q}@cs.virginia.edu.*
- *J.H. Aylor and M.H. Salinas are with the Department of Electrical Engineering, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA 22903-2442. E-mail: {jha, msalinas}@virginia.edu.*
- *R.H. Klenke is with the Department of Electrical Engineering, Virginia Commonwealth University, 601 W. Main St., Room 222, PO Box 843072, Richmond, VA 23284-3072. E-mail: rhklenke@vcu.edu.*
- *S.I. Hong is with Lockheed Martin Federal Systems, 9500 Godwin Dr., Manassas, VA 20110. E-mail: sung.hong@lmco.com.*

Manuscript received 9 Apr. 1999; accepted 4 Sept. 1999.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 109578.

2 DRAM Basics

Before discussing our technique for improving memory system performance, we first review the basic operation of dynamic memory devices. Since DRAM storage cell arrays are typically rectangular, a data access sequence consists of a row access (RAS, or *row address strobe* signal) followed by one or more column accesses (CAS, or *column address strobe* signal). During RAS, the row address is presented to the DRAM. In *fast page mode* (or just *page mode*), data in the storage cells of the decoded row are moved into a bank of *sense amplifiers* (or a *page buffer*), which serves as a row cache. During CAS, the column address is decoded and the selected data is read from the sense amps. Once the sense amps are precharged and the selected page (row) is loaded, the page remains charged long enough for many columns to be accessed. Consecutive accesses to the current row—called *page hits* or *row hits*—require only a CAS, allowing data to be accessed at the maximum frequency.

Although the memory core of Rambus DRAMs is similar to that of other current DRAMs, the architecture and interface are unique. An RDRAM is actually an interleaved memory system integrated onto a single chip. The interface provides separate pins for row address, column address, and data and the pipelined microarchitecture supports up to four outstanding requests. By transferring 16 bits of data on each edge of the 400MHz interface clock, even a single Direct RDRAM chip can yield up to 1.6 Gbytes/sec in bandwidth. All 64 Mbit RDRAMs incorporate at least eight independent banks of memory. Some RDRAM cores incorporate 16 banks in a "double bank" architecture, but two adjacent banks cannot be accessed simultaneously, making the total number of independent banks effectively eight [44], [45].

The key point is that the order of requests strongly affects the performance of all these memory devices. Request order is important on another level: Accesses to different banks can be performed faster than successive accesses to the same bank. In addition, the order of reads with respect to writes affects bus utilization: Every time the memory controller switches between reading and writing, a *bus turnaround delay* must be introduced to allow data traveling in the opposite direction to clear.

3 Access Ordering

A comprehensive solution to the memory bandwidth problem should exploit the richness of the full memory hierarchy, both its architecture and its component characteristics. One way to do this is via *access ordering*—changing the order of memory requests to increase memory system performance. For applications that perform vector-like, *streaming* memory accesses, for instance, effective bandwidth can be increased by reordering the requests to take advantage of device properties such as page mode, internal banking, and pipelined interfaces. The notion that performance of memory-intensive applications can be improved by reordering memory requests is not new, but our work is unique in the combination of how and when that ordering is applied. Access ordering systems can be broadly classified by three key components:

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

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

3.1 Compile-Time Schemes

Benitez and Davidson [4] detect streams at compile time and Moyer [40] derives access-ordering algorithms relative to a precise analytic model of memory systems. 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's subroutines to mimic Cray instructions on the Intel i860XR include another purely compile-time approach: He treats the cache as a pseudo "vector register" by reading vector elements in blocks (using no-caching load instructions) and then writing them to a preallocated portion of cache [32]. Meadows et al. describe a similar scheme for the Portland Group International i860 compiler [39] and Loshin and Budge give a general description of the technique [31]. The benefits of this kind of (CT, CT, CT) access ordering can be dramatic: We measured the time to load a single vector via Moyer's and Lee's schemes on a node of an iPSC/860, observing performance improvements between about 40 to 450 percent over cache-line fills, depending on the stride of the vector [41].

Traditional caching and cache-based software prefetching techniques (including the compiler-directed prefetching of Callahan et al. [13], Mowry et al. [37], and Klaiber and Levy [30], and the software-controlled caches of Cheriton et al. [16]) may also be considered (CT, CT, CT) schemes. The compiler detects streams, determines the order of the memory accesses, and decides where in the instruction stream the accesses are issued. Alexander et al.'s compiler optimizations for wide-bus machines [1] and Davidson and Jinturkar's memory-access coalescing [20] also fall into the (CT, CT, CT) category, as do schemes that prefetch into registers or into a special *preload buffer* (as in Chen et al.'s technique for register preloading [14] and hardware support for loop-based preloading [8]). The "ordering" in the latter prefetching schemes is simply the processor's natural access order for the computation. All prefetching techniques attempt to overlap memory latency with computation, which can lead to significant performance increases. Most such techniques can be rendered more effective by combining them with an access-ordering scheme to exploit architectural and device characteristics of the underlying memory system.

The purely compile-time approach can be augmented with an enhanced memory controller that provides buffer space and that automates vector prefetching, producing a (CT, CT, RT) system. Doing this relieves register pressure

and decouples the sequence of accesses generated by the processor from the sequence observed by the memory system: The compiler determines a sequence of vector references to be issued and buffered, but the actual access issue is executed by the memory controller. Schemes that decouple the issuing of the memory accesses from the processor's instruction execution without performing sophisticated access scheduling can be considered (CT, CT, RT) schemes. For instance, Chiueh [10] proposes a programmable prefetch engine that fetches vector data for the next loop iteration. This data is stored in a special buffer, the *Array Register File*, until the corresponding iteration is executed, at which point the prefetched data is transferred to cache. Using a separate prefetch buffer avoids cache conflicts between the current and future working sets of vector data, but not between the vectors and the scalar data that they may displace. The scheme has a limited *prefetch distance*, the time between a prefetch operation and the corresponding load instruction. Furthermore, it assumes that all memory accesses take about the same amount of time, making no attempt to improve effective bandwidth by reordering vector accesses.

3.2 Run-Time Schemes

The (CT, CT, CT) and (CT, CT, RT) solutions are static in the sense that the order of references seen by the memory is determined at compile time. *Dynamic* access ordering systems determine the interleaving of a set of references at run-time, either by introducing logic into the memory controller, by executing code to decide the reference pattern, or by some combination of the two. The benefits of compile-time ordering schemes can be substantial, but their performance is below those of dynamic schemes. The compiler cannot generate the optimal access sequence without the address alignment information that is usually only available at run time. For instance, on systems with page mode DRAMs, the compiler cannot determine where stream data crosses DRAM page boundaries.

For a dynamic (CT, RT, RT) system, stream descriptors are developed at compile time and sent to the memory controller at run time, where the order of memory references is determined dynamically and independently. Determining access order dynamically allows the controller to optimize behavior based on run-time interactions. Valero et al. propose efficient hardware to dynamically avoid bank conflicts in vector processors by accessing vector elements out of order, analyzing this system first for single vectors [50] and then extending the work for multiple vectors [51]. Del Corral and Llaberia analyze a related hardware scheme for avoiding bank conflicts among multiple vectors in complex memory systems [17]. These access ordering schemes focus on vector computers whose memory systems are composed of SRAM components, which have uniform access time.

Current approaches most closely related to ours are the Command Vector Memory System proposed by Corbal et al. [9] and the Impulse Adaptable Main Memory Controller being developed by Carter et al. [11]. Corbal et al.'s system exploits parallelism and locality of reference to improve effective bandwidth for vector accesses on out-of-order vector processors with dual-banked SDRAM memories. Instead of sending individual requests to specific devices, this approach broadcasts commands requesting multiple, independent words. The memory subsystem orders requests to each dual-banked device, attempting to overlap precharge operations to each internal SDRAM bank with access operations to the other. This system buffers stream data in vector registers within the CPU.

The Impulse memory controller increases processor cache and memory bus utilization by dynamically remapping physical memory. Impulse enables several optimizations that let the application control how, when, and where its data are loaded into the on-chip caches: gathering sparse data into dense cache lines, tiling and recoloring data structures without copying, and mapping noncontiguous physical pages to a single TLB entry [49], [11]. The compiler or application programmer inserts system calls to remap data structures, making this a (CT, RT, RT) approach. Impulse prefetches and buffers data within the memory controller until the CPU requests them, avoiding cache pollution. Most of the approaches outlined in this section do little to improve memory performance for sparse data structures, but Impulse remaps irregular or strided data so that elements occupy dense regions of cache. Mathew et al. describe a subcomponent of the Impulse controller, the Parallel Vector Access Unit [38]. This unit operates on vector commands and exploits SDRAM device characteristics to gather strided data efficiently. Other kinds of access ordering are under investigation.

Fully dynamic (RT, RT, RT) systems implement access ordering without compiler support by augmenting the previous controller with logic to decide what to fetch and when. Whether or not such a scheme is superior to a (CT, RT, RT) system depends on the relative quality of the compile-time and run-time algorithms for deciding the access pattern, the extent to which prefetching is exploited (that is, whether or not there is a limited prefetch distance), and the relative hardware costs.

Baer and Chen [2], Fu and Patel [22], and Sklenar [48] have proposed (RT, RT, RT) "vector prefetch units" that induce stream parameters at run-time. The cache-based sequential hardware prefetching of Dahlgren et al. [19] eliminates the need for detecting strides dynamically. The prefetch distance of these run-time techniques is generally limited to a few loop iterations (or a few cache lines) and the prefetched data may replace other needed data or may be evicted before it is used. None of these schemes orders accesses to fully exploit the underlying memory architecture. The lookahead technique proposed by Bird and Uhlig [7] uses a *Bank Active Scoreboard* to order accesses dynamically to avoid bank contention, but does not try to exploit device characteristics such as page mode.

Palacharla and Kessler [43] investigate code restructuring techniques to exploit an (RT, RT, RT) unit-stride *read-ahead* stream buffer and page mode memory devices on the Cray T3D [29]. The read-ahead mechanism operates like Jouppi's proposed stream buffers [27]: On a cache miss, the memory controller first performs the cache-line fill, then the read-ahead hardware automatically prefetches the next consecutive cache line into a stream buffer inside the memory controller. If the next cache miss hits in the stream

buffer, the entire line is transferred to cache and the next cache line of data is prefetched to the buffer. If the next cache miss also misses in the stream buffer, the buffer's contents are discarded, the desired line is fetched for the cache, and the subsequent line is prefetched. In Palacharla and Kessler's approach, the order in which vectors are fetched is decided at compile-time, but they avoid cache conflicts by determining at run-time the amount of each vector to fetch at once. They measure a performance improvement of up to 75 percent in two, three, and four-stream examples and Brooks demonstrates a factor of 13 improvement in T3D performance after applying access ordering to a 3×3 matrix multiplication routine used in Quantum Chromo Dynamics codes [6]. These performance benefits are substantial, but this approach offers little flexibility: "Programming" the streaming mechanism amounts to rearranging the source code to present the hardware with an appropriate sequence of addresses. Effectively exploiting these stream buffers thus requires significant modifications to the source program.

Palacharla and Kessler also investigate the use of a set of stream buffers as a replacement for secondary cache [42] (similar to what was implemented in the Cray T3D and T3E multiprocessors [47]). This scheme generally increases cache hit rates for the benchmarks simulated, but these improvements come at the expense of increased main memory bandwidth requirements. Even with a filter to reduce the number of false stream accesses, these stream buffers require as much as 45-50 percent extra bandwidth for a few of the scientific benchmarks studied and 25 percent or more extra bandwidth for more than half of them. Farkas et al. mitigate this problem with an incremental prefetching technique that reduces stream buffer bandwidth consumption by 50 percent without decreasing performance [21].

In addition to implementing read-ahead stream buffers, the Cray T3E multiprocessor augments the memory interface of the DEC 21164 microprocessor with a large set of explicitly managed, memory-mapped, external registers called *E-registers* [47]. The E-registers can be programmed to perform vector *get* or *put* operations to transfer eight words with arbitrary stride between nodes. The large number of E-registers allows gets and puts to be highly pipelined and the bus interface allows up to four properly aligned get/put commands to be issued each two-cycle bus transaction. Gather operations use strided vector gets to load contiguous E-registers, which are then loaded "broadside" in cache-line increments (i.e., the width of the bus) into registers on the processor chip. Since E-register data is accessed via I/O space loads, vector elements gathered this way must be copied to local memory to be cacheable.

Contention for resources can offset the benefits of any prefetching scheme. Approaches that prefetch into general-purpose registers suffer from register pressure and those that prefetch into cache without remapping data suffer from cache conflicts. How much these factors affect performance depends on the system and workloads in question, but the detrimental effects can be significant. To put the cache-interference problem in perspective for image processing applications, Impulse-style remapping to remove cache conflicts accounts for a speedup of 180 percent (and improved TLB performance yields an even greater overall speedup) on a ray tracing benchmark and reduces memory stall time by almost a factor of five (from 16 million cycles to 3.4 million cycles) on an image filtering benchmark [53].

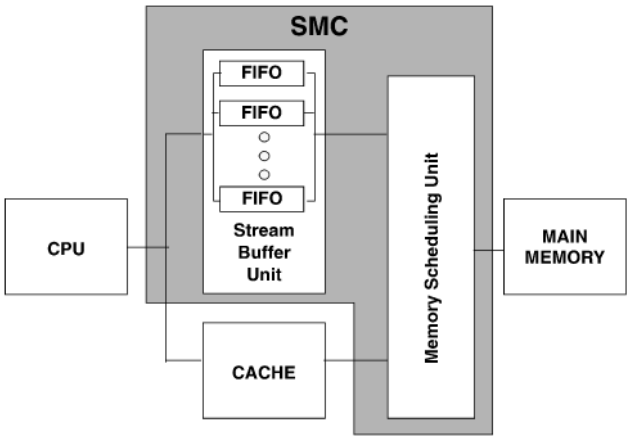

Fig. 1. Stream memory controller organization.

4 The Stream Memory Controller

Unique to our work is the premise that access ordering should be:

- performed to exploit both memory system architecture and device component capabilities, and
- done at run-time, when more information is available on which to base scheduling decisions.

4.1 Architecture

We describe our approach—a (CT, RT, RT) system in our taxonomy—based on the simplified architecture of Fig. 1. In this system, the compiler must detect the presence of streams (as in [4]) and arrange to transmit information about them (i.e., base address, stride, length, data size, and whether the stream is being read or written) to the hardware at run-time. The dynamic access ordering hardware then prefetches the read operands, buffers the write operands, and reorders the accesses to get better memory system performance.

Our dynamic access ordering hardware, called a *Stream Memory Controller* (SMC), is logically divided into two components: a *Stream Buffer Unit* (SBU) and a *Memory Scheduling Unit* (MSU). The MSU is a controller through which memory is interfaced to the CPU. It includes logic to issue memory requests and to determine the order of requests during streaming computations. For nonstream accesses, the MSU provides the same functionality and performance as a traditional memory controller. As with the stream-specific parts of the MSU, the SBU is not on the critical path to memory and the speed of nonstream accesses is not adversely affected by its presence.

The MSU has full knowledge of all streams currently needed by the CPU: Using the base address, stride, and vector length, it can generate the addresses of all elements in a stream. It also knows the details of the memory architecture, including interleaving and device characteristics. The access-ordering circuitry uses this information to

issue requests for individual stream elements in an order that attempts to maximize memory system performance.

The Stream Buffer Unit contains high-speed buffers for stream operands and provides memory-mapped control registers that the processor uses to specify stream parameters. By memory mapping the control registers and FIFO heads, we avoid having to modify the processor's instruction set.

Even though the hardware depicted in Fig. 1 does not cache stream data, the dynamic access ordering described here is equally valid for memory systems that can perform intelligent access ordering and/or prefetching and buffering within the memory controller, even if they always transmit data in cache-line increments to a processor chip with a traditional cache hierarchy. Application-driven remapping of physical addresses at the memory controller [49], [11] can be used to give programs more control over how stream data is cached. In the SMC system described here, stream data can be cached by copying it to a portion of memory that has been preallocated in cache, as in Lee's subroutines [32]. *Fundamentally, the decision whether or not to cache stream data is orthogonal to the problem of designing an efficient memory controller for modern DRAMs.*

4.2 Programming Model and Compilation

We have tried to heed the principal lesson from RISC designs, namely to wisely partition what is done at compile time and what is done at run time—and particularly to keep run-time operations as simple and regular as possible. Although there are several possible programming models for the Stream Buffer Unit, the one we chose is that of a set of FIFOs, each of which is 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 (write) the next data item in the stream. The act of accessing this location dequeues an input datum or enqueues an output datum.

Conceptually, these buffers need not be implemented as FIFOs from the memory system's perspective. The Memory Scheduling Unit 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. From the memory side, the buffers could appear to be a small addressable memory, or register file. For a system with an out-of-order or speculative processor, the stream buffers would need to appear as addressable memories from both sides, since the CPU need not access elements in stream order.

Making the stream buffers behave like FIFOs simplifies the compilation problem. Fig. 2 shows the code for dot product as generated for, Fig. 2a, a MIPS microprocessor [36] and, Fig. 2b, a hypothetical MIPS extension that includes a stream control unit. The original code in Fig. 2a was generated by a fairly conventional optimizing compiler and has had strength-reduction applied to it. The compiler recognizes that the computations of the addresses of $a(i)$ and $b(i)$ (which are of the form $a + i \times 4$) do not have to actually perform the multiplication on each iteration. Outside the loop, a compiler-generated temporary location is initialized to the base address of the vector. This

```
      do 10, i=1, 1000
10    s = s + a(i) * b(i)
```

```
L63:  lw    $2,($11)
      lw    $3,($10)
      mul   $3,$3,$2
      addu  $6,$6,$3
      addu  $10,$10,4
      addu  $11,$11,4
      blt   $11,$4,L63
```

(a)

```
      sin32i  $s0,$11,$9,4
      sin32i  $s1,$13,$9,4
L63:  mul     $3,$s0,$s1
      addu    $6,$6,$3
      addu    $5,$5,1
      blt     $5,$9,L63
```

(b)

Fig. 2. Example code for dot product. (a) Normal MIPS. (b) MIPS with streaming.

temporary is then merely incremented by four bytes on each loop iteration.

Except for trivial differences, this is the same information needed by the streamed code shown in Fig. 2b, which was generated by an experimental compiler built as part of this research project [4]. At the point where the usual optimizer initializes a temporary storage location to the base address, the streaming compiler emits code to initialize the FIFO control register—the *sin32i* ("stream in 32-bit integer") instruction in this case. 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). The key point in this example is that it demonstrates the feasibility of stream detection: The compilation process is not especially difficult. As a beneficial side effect, the number of instructions in the inner loop is reduced because 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 relatively expensive dependency analysis is not needed here. Many recurrence dependences can be broken by streaming [4] and the compiler can insert run-time checks that trigger execution of a nonstreaming version of the loop when true dependences exist.

5 Methodology

The SMC is only intended to speed up the inner loops of streamed computations, so we use benchmark kernels to evaluate our design. The impact of dynamic access ordering on whole program performance is being studied as part of the Impulse project [11]. Fig. 3 lists the kernels used to generate the results presented here. *daxpy*, *copy*, and *scale* are from the BLAS (Basic Linear Algebra Subroutines) [18], and *tridiag* is a tridiagonal gaussian elimination fragment, the fifth Livermore Loop [35]. *vaxpy* denotes a "vector axpy" operation that occurs in matrix-vector multiplication by diagonals: A vector a multiplied by a vector x plus a vector y.

For our purposes, the actual computation in these loops is unimportant; we focus instead on the access pattern and the lengths of the streams. These kernels represent the access patterns frequently found in real codes. For instance,

kernel	operation	access modes
copy	for (i = 0; i < L * S; i += S) y[i] = x[i];	x: rd y: wr
daxpy	for (i = 0; i < L * S; i += S) y[i] += a * x[i];	x: rd y: rd-mod-wr
scale	for (i = 0; i < L * S; i += S) x[i] = a * x[i];	x: rd-mod-wr

swap	for (i = 0; i < L * S; i += S) {
	reg = x[i];
	x[i] = y[i];
	y[i] = reg;
	}
		x: rd-mod-wr
		y: rd-mod-wr

tridiag	for (i = 0; i < L * S; i += S) x[i] = z[i] * (y[i] - x[i-1]);	x: wr y: rd z: rd
vaxpy	for (i = 0; i < L * S; i += S) y[i] += a[i] * x[i];	a: rd x: rd y: rd-mod-wr

Fig. 3. Benchmark kernel access patterns.

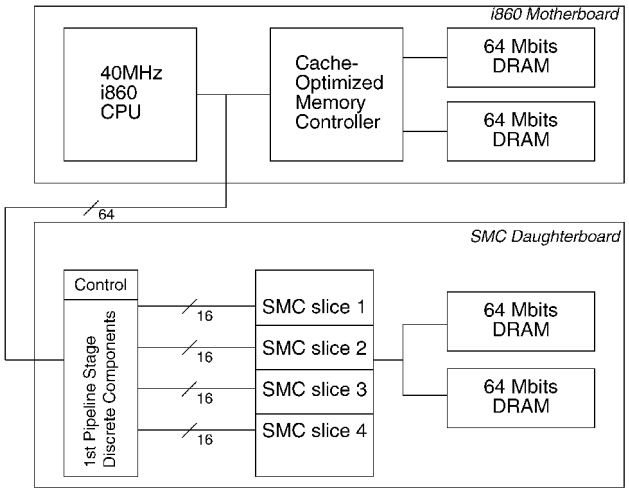

Fig. 4. SMC system architecture.

copy and *scale* are the memory access patterns of JPEG and MPEG operations in multimedia applications. To a first-order approximation, MPEG video encoding is simply JPEG coding on each successive frame. The JPEG coding algorithm includes several steps that either manipulate the entire image or operate on smaller blocks of data, giving rise to two basic stream lengths. The first stream size is related to the size of the image and can range from five thousand to about two million, depending on the video resolution. The second stream size is that of a block, which is either 64 or 100.

To illustrate the effectiveness of our approach, we present two categories of results. The first explores performance for systems with our prototype's organization: a uniprocessor system with an external SMC chip managing an interleaved memory of fast-page mode DRAMs. These results include analytic bandwidth bounds, functional simulation results, and measured hardware performance. The second category explores a wider design space and includes analytic and simulation results for systems with SMCs integrated onto processor chips: We examine a range of fast-page mode DRAM memory systems similar to the prototype, as well as two single-chip Direct Rambus memory systems.

We present our results as a percentage of peak bandwidth or that which would be achieved if the CPU could complete one memory access each processor cycle. The vectors we consider are of equal length, unit stride, share no DRAM pages in common, and are aligned to begin in the same bank, unless otherwise noted. To put as much stress as possible on the memory system, arithmetic computation is assumed to be infinitely fast and is abstracted out of each kernel. For the hardware results, we execute each loop prior to beginning our measurements so that the experiment can run entirely out of the instruction cache. The i860's eight Kbyte data cache is two-way set associative, write-back, and write-around, with pseudorandom replacement and 32-byte lines. In our experiments, all SMC stream references use noncaching loads and stores.

6 Results

6.1 Performance for a Separate SMC ASIC

Our proof-of-concept Stream Memory Controller system is implemented as a single, semi-custom VLSI integrated circuit interfaced to a 40MHz Intel i860 host processor [26]. We chose the i860 for its ready availability and because it provides load/store instructions that bypass the cache. 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 of Fig. 1, but the organization is logically the same.

The results in this subsection describe the performance of an off-chip SMC system with the architecture depicted in Fig. 4. The i860 motherboard is interfaced via an expansion connector to an SMC daughterboard. The motherboard contains an i860XP processor, its eight Kbyte data cache, a system boot EPROM, a memory controller optimized for cache-line fills, and 16 Mbytes of fast-page mode DRAM. The daughterboard contains the four bit-sliced VLSI ASICs that constitute the SMC, its memory subsystem, and a pipeline stage needed to meet timing and line-length constraints.

Each bank of DRAM memory on the daughterboard is composed of two 32 Mbit 60 nsec page-mode components with 1 Kbyte pages. The minimum cycle time for fast page-mode accesses is 35 nsec and random accesses require 110 nsec. Wait states make the SMC's observed access time for sustained accesses 50 nsec (two CPU cycles) for page hits and 175 nsec for page misses (seven CPU cycles, including the time to precharge and set up the new DRAM page). Since there are two interleaved banks of memory, the SMC can deliver one 64-bit data item every 25 nsec processor cycle for streams with relatively prime strides. This matches the consumption rate of the 40MHz i860 host processor: It can only initiate a new bus transaction every other clock cycle, but quadword instructions allow the processor to read 128 bits of data in two consecutive clocks.

The processor takes about 14 nsec to assert its address and cycle definition pins and the signals take another five nsec to propagate to the expansion connector. This

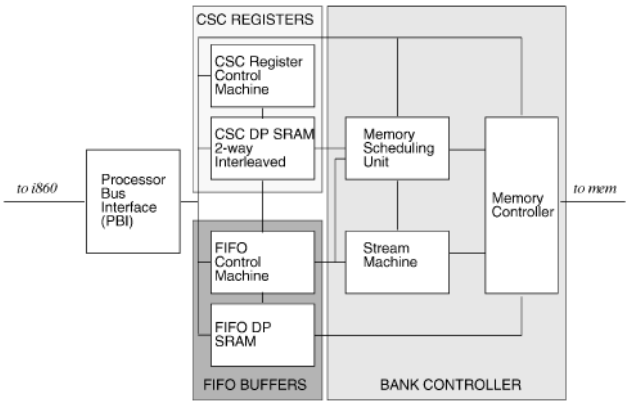

Fig. 5. SMC ASIC architecture.

leaves less than six nsec in the current cycle 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. In light of these two constraints, we added a single-stage, bidirectional pipeline to the daughter board; this component 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. Our off-chip implementation thus incurs pipeline delays in addition to bus turnaround delays when switching between reading and writing. The latter delays would not be present in the on-chip SMCs described in Section 6.2 or in a system that doubles the bus width and then drives alternate halves, as in Mathew et al.'s Parallel Vector Access Unit [38]. Nonetheless, the performance of our prototype SMC represents a significant improvement over the performance of a non-SMC system for stream accesses.

Our prototype Stream Memory Controller is a 132-pin ASIC implemented in a 0.75 mm, three-level metal HP26B process fabricated through MOSIS. We chose a four-way bit-sliced organization over a full 64-bit wide version to avoid being severely pad-limited in size. Fig. 5 illustrates the decomposition of each 16-bit SMC ASIC into four logical components: the Processor Bus Interface (PBI), the Command Status and Control (CSC) registers, the FIFO Buffers, and the Bank Controller (BC).

The PBI state machine shown at the left of Fig. 5 provides the logic necessary to interface the SMC with the i860 processor bus. The PBI manages accesses to the CSC registers, stream accesses to the memory mapped FIFO heads, and nonstream (scalar) accesses to the memory subsystem. The CPU transmits the base, length, and stride parameters for each stream by writing the CSC registers. These registers are implemented with dual-ported SRAM, allowing both the CPU and the BC to access them simultaneously.

The FIFOs buffer data between the processor and the memory and can be accessed by both simultaneously. The buffer component is broken down into two sections: the dual-ported SRAMs used to implement virtual FIFOs and the FIFO controller state machine that generates the addresses for Memory Scheduling Unit (MSU) accesses to the FIFOs. The FIFO controller logic provides signals conveying "fullness" information for each FIFO to both the BC and the PBI. The PBI uses these signals to determine when a given access can be completed and the BC uses them to decide which memory access to perform next. The BC logic handles the interface to the interleaved memory system and fills or drains the FIFOs. The BC also provides support for scalar accesses to the SMC daughterboard memory.

This version of the SMC is 52 square millimeters and about 150,000 transistors, with an estimated power dissipation of 1.14 watts. It includes four software-programmable FIFOs that can each be set to read or write and whose depth can be adjusted to powers of two from eight to 128 double-word elements. The prototype's Memory Scheduling Unit implements a very simple ordering policy: The BC considers each FIFO in round-robin order, performing as many accesses as it can for the current FIFO before moving on to the next. Despite its simplicity, this ordering strategy works well in practice. For uniprocessor systems, its simulation performance is competitive with that of more sophisticated policies. More intelligent schemes are required to achieve uniformly good performance on streams whose strides do not hit all memory banks and on multiprocessor systems in general [34].

6.1.1 Effective Bandwidth for Long-Stream Computations

Fig. 6 illustrates the measured performance of our prototype system on each of the benchmark kernels with vectors of 16 to 8K elements and with the FIFO depth set at 16. These graphs show the percentage of the peak system bandwidth exploited for each benchmark. The short-dashed lines labeled "limit" indicate the combined effect of two performance bounds: SMC startup costs and unavoidable page misses and bus-turnaround delays (derivations of performance bounds are given elsewhere [34]). The long-dashed lines indicate the performance of our software simulations and the solid lines indicate the performance of our prototype hardware. The dotted lines indicate the performance measured when using caching load instructions to access the stream data in the i860's own cache-optimized memory and the dot-dash lines indicate the performance measured when using the i860's noncaching pipelined floating point load (*pfld*) instruction. These performances have nothing to do with FIFO depth, but we represent them with lines on these graphs for purposes of comparison. We unroll each loop eight times for the cache and SMC experiments, but the *pfld* results represent the natural access order for the computation. Unrolling and grouping reads and writes minimizes the number of bus transitions between reading and writing, amortizing turnaround delays on the bus between the CPU and the SBU over several accesses.

The cache performance numbers presented here in some sense represent a lower bound since the i860's data cache has relatively short lines. Most cache controllers, including the i860's, exploit memory device characteristics only within a single cache line, but caches with longer lines can deliver better effective bandwidth for small-stride

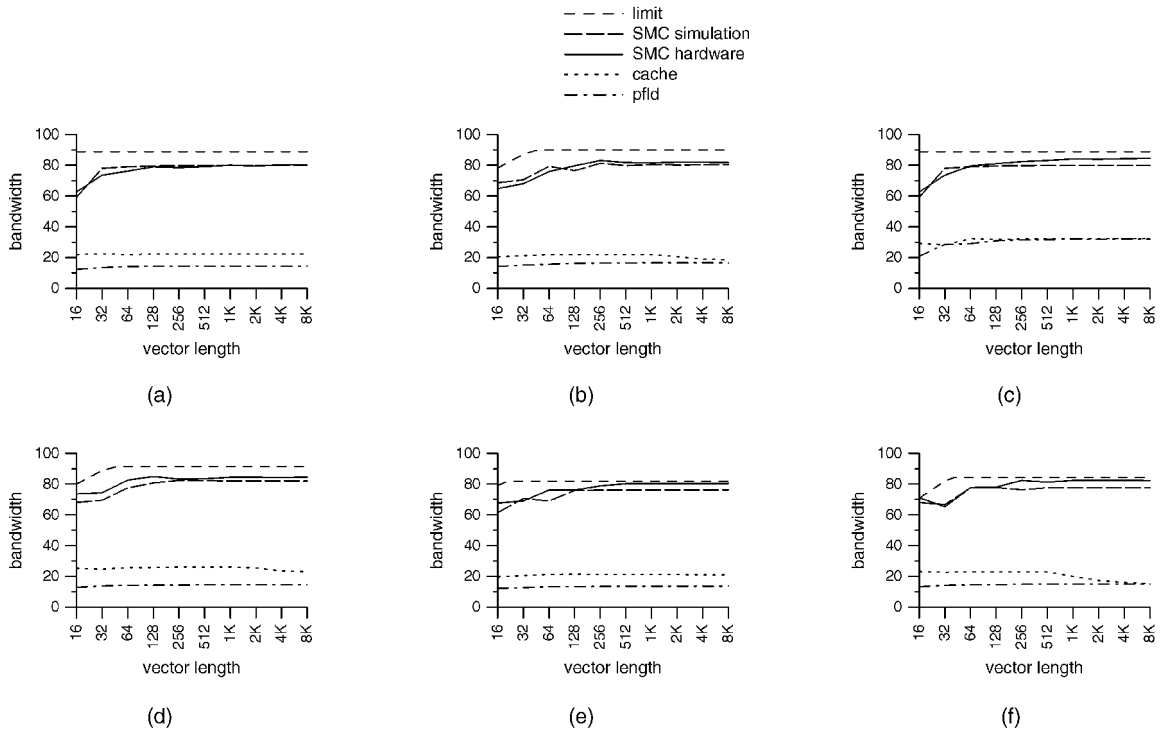

Fig. 6. Percentage of peak bandwidth for 16-deep FIFOs and varying vector lengths. (a) *copy*. (b) *daxpy*. (c) *scale*. (d) *swap*. (e) *tridiag*. (f) *vaxpy*.

streams. For other strides (unless physical addresses are remapped to increase locality), caching streams wastes bus bandwidth and cache space by loading unneeded data. We do not measure those effects here, but they have been addressed elsewhere: McKee and Wulf's analytic models and measurements explore effective bandwidth for caching accesses under a variety of cache-controller assumptions [41] and Carter et al.'s empirical results quantify the negative effects of cache pollution and wasted bus bandwidth for scientific applications [11].

These experiments measure the efficiency with which the memory subsystem transfers data to and from the processor, not the execution time of the loop. Compiler optimizations like prefetching are orthogonal to this analysis—they might change when a datum is loaded with respect to its use, but the same amount of data is transferred, regardless. Note that software prefetching may offer little benefit for bandwidth-limited loops if there is insufficient computation between accesses to mask the memory latency: The memory system would quickly become saturated.

The effective bandwidth delivered by the SMC for these kernels is between 2.14 and 3.75 times that delivered by cache-line fills. Performing the computation with caching accesses yields less than 32 percent of the system's peak bandwidth for all access patterns. For the two multiple-vector kernels that both read and write the same vector (*daxpy* and *vaxpy*), cache performance falls off when vector length exceeds the cache size and modified cache lines are written to memory as they are evicted. The i860's write-back operation when dirty lines hit the current DRAM page is more efficient than its cache-line fill, as evidenced by the absence of a drop in cache performance for *scale* on long vectors. This kernel's cache performance would rival the SMC's if the i860's cache controller could take more advantage of page mode for the read accesses.

When noncaching instructions are used in the natural order of the computation, performance is generally even worse than when using caching loads. The exception to this is *scale*, results for which are shown in Fig. 6c. This kernel operates on a single vector and so the accesses always hit the open page.

Variations in the processor's reference sequence have little effect on the SMC's ability to improve bandwidth, as evidenced by the similarity of the performance curves for different benchmarks. The slight dips in the SMC performance curves at 32-element vectors for the *tridiag* and *vaxpy* kernels in Fig. 6 occur because of an interaction between the number of streams, the vector length, and the FIFO depth. Exactly when the DRAM page misses happen depends on all these parameters and the shorter the vectors are, the greater the impact each page miss has on overall performance. Plotting points for vectors of every length reveals a saw-tooth shape, the "teeth" of which get smaller as vector length grows and page misses are amortized over more accesses. Fig. 7 shows this detail for the *copy* kernel.

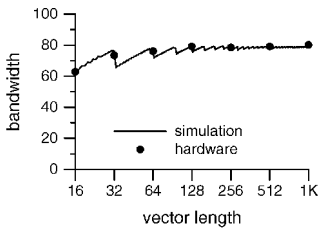

Fig. 7. Detailed *copy* performance for 16-deep FIFOs and varying vector lengths.

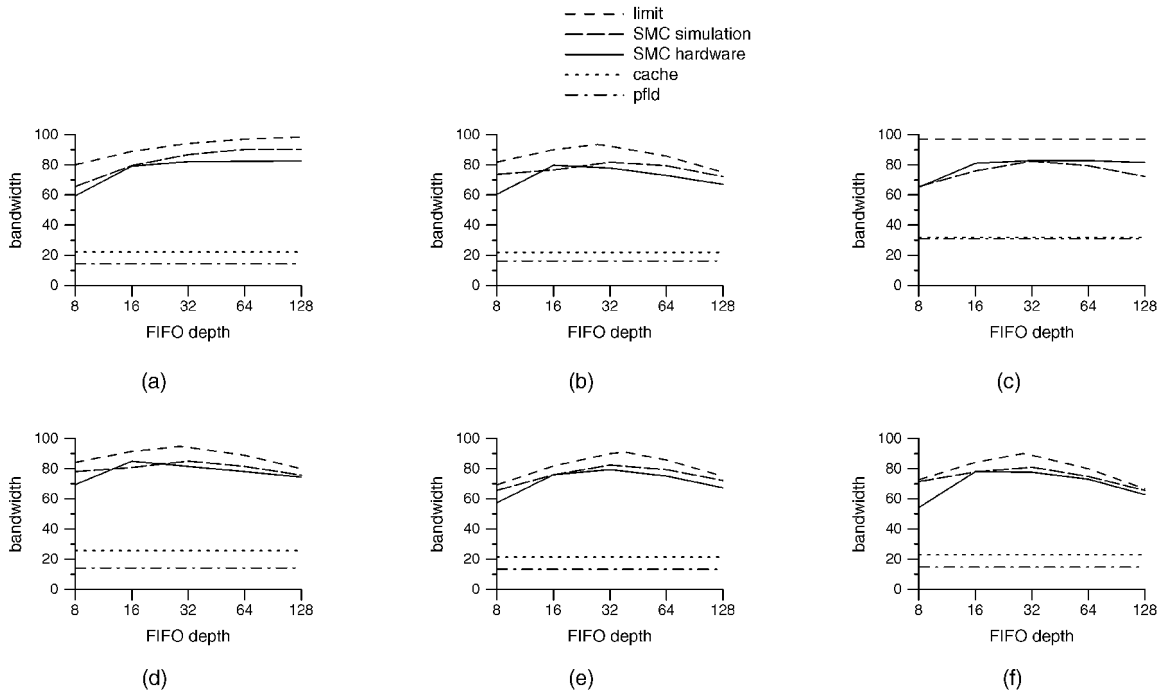

Fig. 8. Percentage of peak bandwidth for 128-element vectors and varying FIFO depths. (a) *copy*. (b) *daxpy*. (c) *scale*. (d) *swap*. (e) *tridiag*. (f) *vaxpy*.

6.1.2 FIFO Depth and Attainable Bandwidth

Exploiting page mode as much as possible 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 sth stream while the MSU fills the FIFOs for the first $s-1$ streams. By the time the MSU has provided all operands for the first loop iteration, it will have prefetched data for many future iterations, so the processor will not stall again soon. Nonetheless, even though deeper FIFOs allow the MSU to get more data from a DRAM page each time it is loaded into the sense amps, they cause the processor to wait longer at startup. The graphs in Fig. 8 illustrate the net effect of these competing factors for our benchmark kernels on 128-element vectors. The legend is the same as for Fig. 6. The descending portions of the short-dashed line labeled "limit" show the performance bounds defined by the startup cost. Short-vector computations have fewer total accesses over which to amortize startup and page-miss costs. For these loops, initial delays can represent a significant portion of the computation time. This is easy to see in the performance curves for the kernels that read two or more streams (*daxpy*, *swap*, *tridiag*, and *vaxpy*). The *copy* and *scale* kernels incur no initial delay since they read only one stream. If we were to plot performance for deeper FIFOs, these portions of the curves would be flat: Effective bandwidth remains constant once FIFO size exceeds vector length.

These results illustrate the importance of choosing an appropriate FIFO depth for each computation. Fortunately, the compiler can use the equations for the startup delay bound and the page miss/bus-turnaround bound to generate code that selects the FIFO depth at run time. The heuristic of choosing the FIFO depth closest to the intersection of the two performance limits gives good results in all of our thousands of simulation experiments [34].

The 2K-element vectors used to generate the results depicted in Fig. 9 allow startup and page-miss costs to be amortized much more effectively than the 128-element vectors of Fig. 8. For the longer vectors, initial delays have very little effect on overall performance for the prototype SMC's range of FIFO depths. Performance of this off-chip implementation of the SMC reaches a maximum of about 90 percent of peak system bandwidth, regardless of the computation parameters. This limit reflects the cost of transferring data across chip boundaries.

The disparity between SMC and cache performance is even more dramatic for nonunit stride computations, where each cache-line fill fetches unneeded data. For computations using the i860's pipelined, noncaching load, performance for nonunit strides is about half that for unit strides since quadword instructions can no longer be used to access two elements every two cycles. In contrast, SMC performance is relatively insensitive to changes in vector stride as long as the stride hits all memory banks and is small relative to the page size. For instance, SMC performance for vectors with stride five (the smallest stride for which only

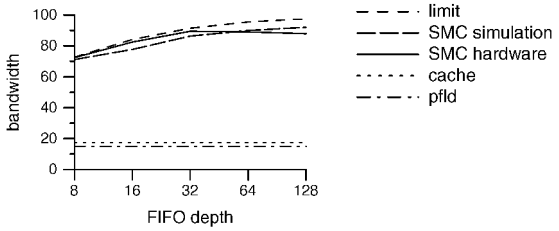

Fig. 9. Percentage of peak bandwidth for *vaxpy* with 32-deep FIFOs and longer vectors.

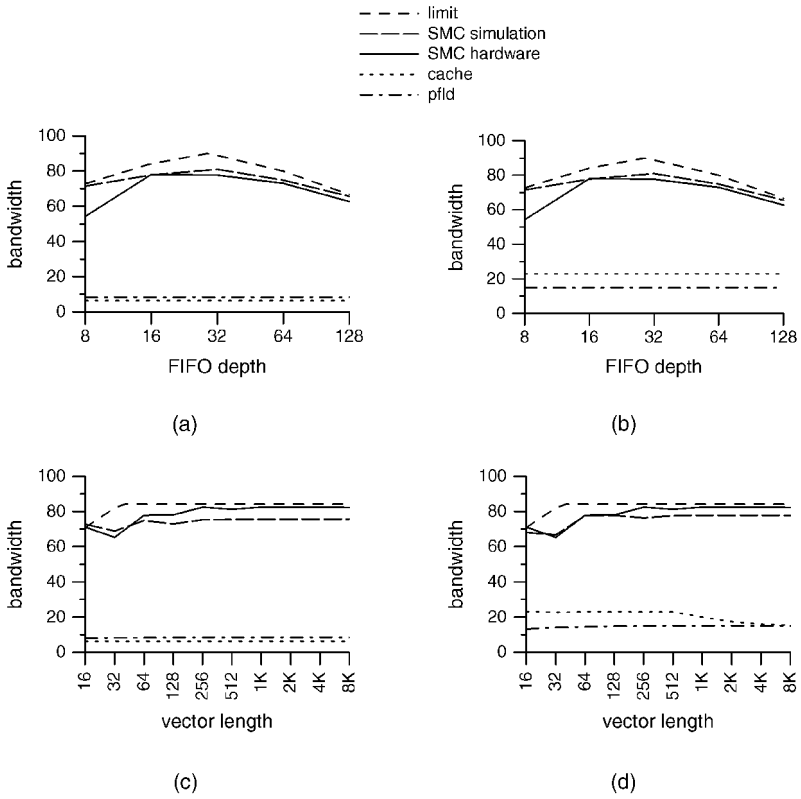

Fig. 10. Percentage of peak bandwidth for *vaxpy*'s access pattern and varying vector strides. (a) 128-element, stride-5 vectors. (b) 128-element, stride-1 vectors. (c) Stride-5 vectors and 16-deep FIFOs. (d) Stride-5 vectors and 16-deep FIFOs.

one element resides in each cache line) on the *vaxpy* access pattern is nearly identical to that for unit stride, whereas the caching load and *pfld* performances decrease by factors of 3.7 and 1.8, respectively. Fig. 10 illustrates this, both as FIFO depth changes (Fig. 10a, Fig. 10b) and as vector length grows (Fig. 10c, Fig. 10d). For these vectors, the SMC delivers between 10.4 and 13.2 times the effective bandwidth of cache-line fills.

As the results in this section illustrate, even an SMC with only a small amount of buffer space (16 elements) can consistently deliver over 80 percent of the peak system bandwidth for all but the shortest vectors. When we take each kernel's inherent bandwidth limits into account, these SMC performances represent between 89 and 98 percent of the attainable bandwidth for vectors over 128 elements. With FIFO depths set at only 32 elements, our system delivers its maximum possible performance on vectors of only 2K elements. Deeper FIFOs yield even better performance for computations on longer vectors.

6.2 Performance for an Integrated CPU and SMC

The last subsection demonstrated the high correlation of our functional simulation results with the performance of our prototype hardware. We also used our simulation model to investigate other system organizations that integrate the SMC into the processor chip, where the stream buffers enjoy the same access times as the on-chip cache. This section presents analytic and simulation results for memory systems composed of up to eight interleaved banks of fast-page mode DRAM and for systems consisting of a single Direct RDRAM.

6.2.1 An Integrated SMC for Interleaved Page Mode Memory Systems

For the fast-page mode systems we examine, other parameters (such as the DRAM page size and the hit/miss cost ratio) are the same as in the prototype system. In Section 6.1, we unrolled loops to amortize bus turnaround delays between the CPU and the SBU over several accesses. This optimization is unnecessary for an on-chip SMC, where the bus width is not limited by the number of available pins.

Fig. 11 shows comparative results for each kernel on vectors of 4K elements for a range of FIFO depths and memory organizations. Since this organization does not suffer the delays inherent in an off-chip memory controller, the SMC can exploit nearly the full system bandwidth for sufficiently deep FIFOs. Vectors of length 4K are long enough to reap most of the SMC's benefit: simulation results for vectors of 16K elements differ by less than 3 percent of peak bandwidth.

By increasing the number of memory banks, we decrease the number of vector elements in each bank, which limits the SMC's ability to amortize page-miss and startup costs. This is particularly evident in the performance of organizations with shallow FIFOs (eight or 16 elements) and a higher degree of interleaving. For instance, the percentage of peak bandwidth delivered for *vaxpy* by an eight-bank SMC system with FIFOs set at eight elements in Fig. 11f is only 35.1 percent of that delivered by a similar single-bank system. This may seem counter-intuitive at first, but systems with more memory banks require deeper FIFOs to deliver good performance. If we assume that total system

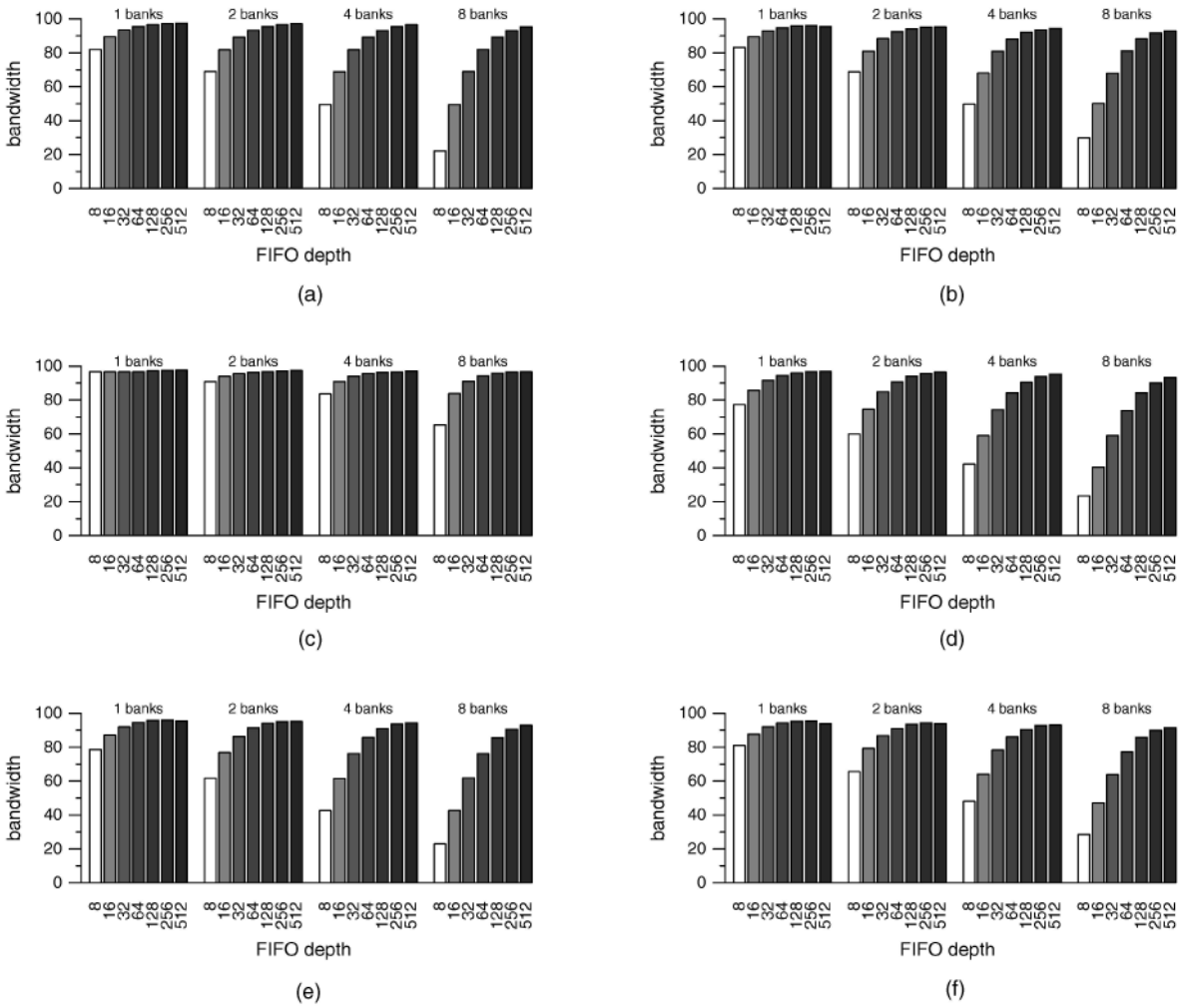

Fig. 11. Percentage of peak bandwidth for long vectors and an integrated SMC. (a) *copy*. (b) *daxpy*. (c) *scale*. (d) *swap*. (e) *tridiag*. (f) *vaxpy*.

bandwidth scales with interleaving, the eight-bank system delivers a smaller percentage of a much larger bandwidth. To put this in perspective, Fig. 12 illustrates how these absolute bandwidths relate to each other.

6.2.2 An Integrated SMC for Two Rambus Memory Systems

The RDRAM's separate pins for row address, column address, and data allow each bank's sense amplifiers to be independently opened, accessed, and precharged [45]. For example, one bank's page can be left open while accessing another bank's sense amps. This independence permits a number of precharge policies. In a *closed-page* policy, the sense amps are always precharged after a data access (or burst of accesses) to a bank. In an *open-page* policy, the sense amps are left open—unprecharged—after a data access to a bank. The fast-page mode systems described above use an open-page policy for stream accesses. A closed-page policy makes more sense when successive accesses are expected to be to different pages and an open-page policy makes more sense if successive accesses are likely to be to the same page.

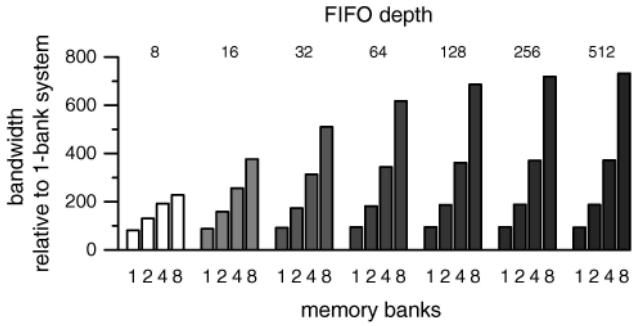

Fig. 12. Percentage of peak bandwidth for *vaxpy* relative to a single-bank system.

We examine two memory configurations, one using a closed-page policy and cache-line interleaving (CLI) so that successive cache lines reside in different RDRAM banks and one using an open page policy and page interleaving (PI) so that switching banks only happens when accessing an RDRAM page different from the last. These two configurations represent two extreme points of the design space for RDRAM memory systems and are both employed in real system designs [44]. Henceforth, references to CLI systems imply a closed-page policy and references to PI systems imply an open-page policy.

In our experiments, we model memory systems composed of Direct RDRAMs with eight independent banks

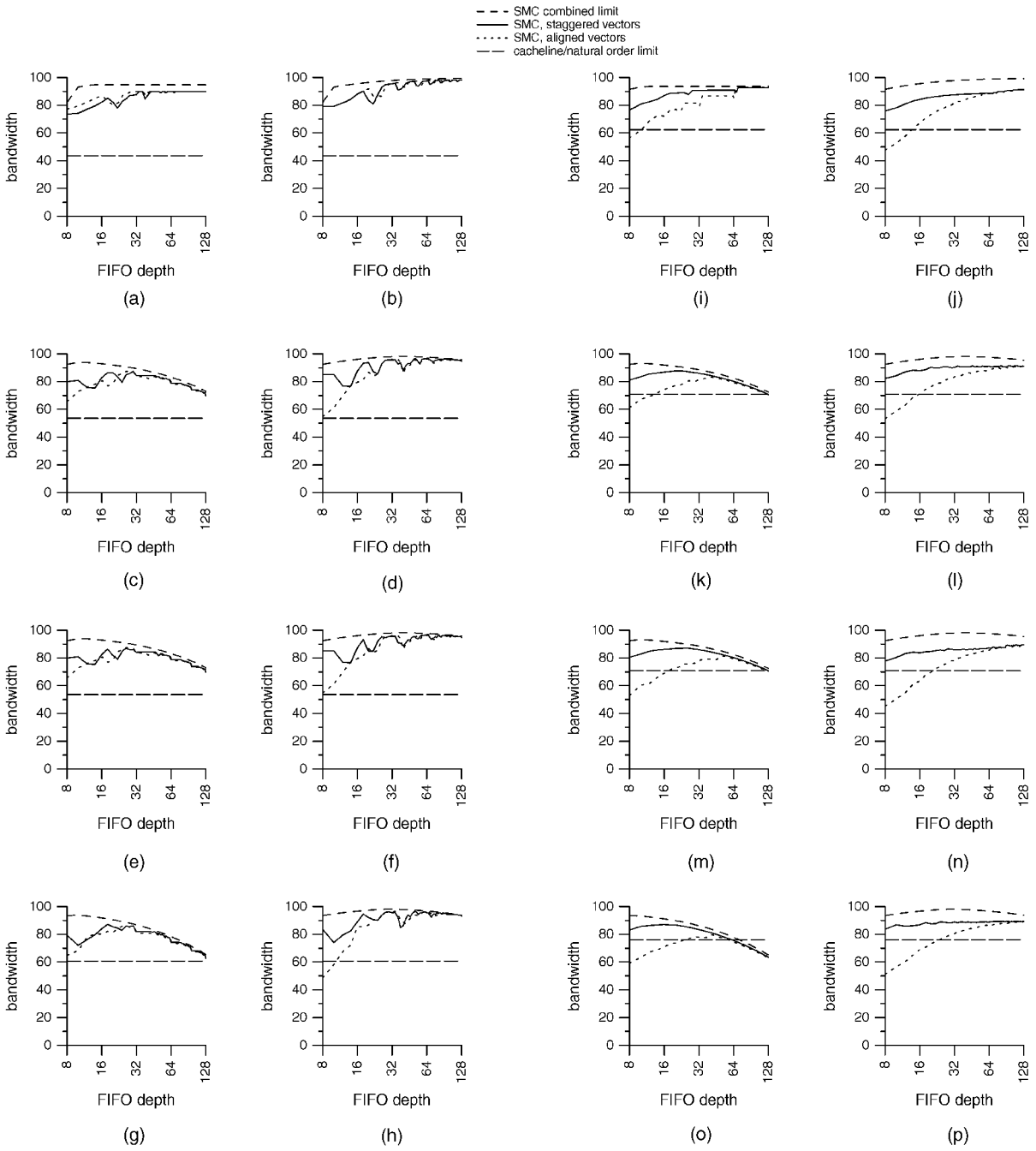

Fig. 13. Percentage of peak bandwidth for direct RDRAM systems. (a) CLI *copy* 128. (b) CLI *copy* 1K. (c) CLI *daxpy* 128. (d) CLI *daxpy* 1K. (e) CLI *tridiag* 128. (f) CLI *tridiag* 1K. (g) CLI *vaxpy* 128. (b) CLI *vaxpy* 1K. (i) PI *copy* 128. (j) PI *copy* 1K. (k) PI *daxpy* 128. (l) PI *daxpy* 1K. (m) PI *tridiag* 128. (n) PI *tridiag* 1K. (o) PI *vaxpy* 128. (p) PI *vaxpy* 1K.

and 1 Kbyte pages (128 64-bit words). The vectors we use are either 128 or 1,024 64-bit elements in length and are unit-stride unless otherwise stated. All communication to and from an RDRAM is performed using packets and each command or data packet requires four 2.5 nsec clock cycles to transfer. Two 64-bit stream elements fit in a data packet and four elements fit in a cache line. We consider FIFO depths from eight to 128 stream elements and we used the same, simple ordering scheme as in our prototype, fast-page mode system. These experiments assume no particular processor model. Results are relative to the Direct RDRAM's maximum bandwidth.

Fig. 13 illustrates our results for each of the four multivector benchmark kernels and the two memory interleaving schemes. The columns on the left, Fig. 13a, Fig. 13b, Fig. 13c, Fig. 13d, Fig. 13e, Fig. 13f, Fig. 13g, Fig. 13h, give CLI system performance and those on the right, Fig. 13i, Fig. 13j, Fig. 13k, Fig. 13l, Fig. 13m, Fig. 13n, Fig. 13o, Fig. 13p, give PI system performance. Graphs in a given row show results for the same benchmark. The first and third columns represent performance for benchmarks with vectors of length 28 and the second and fourth columns represent performance for benchmarks with vectors of length 1,024. The long-dashed lines in these

graphs indicate the maximum bandwidth that can be exploited when accessing streams via cache-line fills. This limit is optimistic in that it ignores potential delays from writing dirty lines back to memory and assumes an optimal data placement such that the computation encounters no bank conflicts. The dashed lines labeled "SMC, max conflict" show simulated SMC performance when the vectors are aligned to cause bank conflicts and the solid lines labeled "SMC, min conflict" illustrate results for a more advantageous data placement. The dashed lines represent the attainable bandwidth defined by two analytic performance bounds, one modeling the influence of the startup delay on performance and one modeling the maximum bandwidth for computations with longer vectors. These equations differ significantly from those for fast-page mode DRAM systems [25]. In an RDRAM system, the precharges to one bank can be overlapped with accesses to another and, thus, bus-turnaround delays—and not page-miss overheads—become the limiting performance factor.

Our results for loading stream data in the natural order in cache-line increments are lower than the 95 percent efficiency rate that Crisp reports [15]. This difference arises because we model streaming kernels on a memory system composed of a single RDRAM device, whereas Crisp's experiments model more random access patterns on a system with many devices. For kernels with four or fewer streams, we find that effective bandwidth is limited to less than 76 percent for PI systems and less than 61 percent for CLI systems. Maximum effective bandwidth increases with the number of streams in the computation: Loops with more streams exploit the Direct RDRAM's available concurrency better by enabling more pipelined loads or stores to be performed between each bus-turnaround delay. A computation on eight, independent, unit-stride streams (seven read-streams and one write-stream, aligned in memory so that there are no bank conflicts between cache-line fills) can exploit up to 88.68 percent and 76.11 percent of peak bandwidth for stride-one vectors on a PI and CLI system, respectively. When the vector stride increases to four or more—so that three-fourths of the data in each cache line goes unused—this performance drops to 22.17 percent and 19.03 percent of peak bandwidth. Even though PI organizations perform better than CLI organizations for streaming, they should perform much worse than CLI for more random, nonstream accesses, where successive cache-line fills are unlikely to be to the same RDRAM page.

Although performance for accessing cache lines in the computation's natural order is sensitive to the number of streams in the computation, the performance for the SMC is uniformly good, regardless of the number of streams in the loop or the order in which the processor accesses them. An SMC always beats using natural-order cache-line fills for CLI memory organizations and an SMC with deep FIFOs on unit-stride, long-vector computations delivers between 2.11 (for *vaxpy*) and 2.94 (for *copy*) times the maximum potential performance of the traditional approach. The improvement is smaller for shorter vectors or shallower FIFOs, particularly for an unfavorable vector alignment. For PI organizations and appropriate FIFO depths, an SMC still beats the natural order every time, although the improvements here

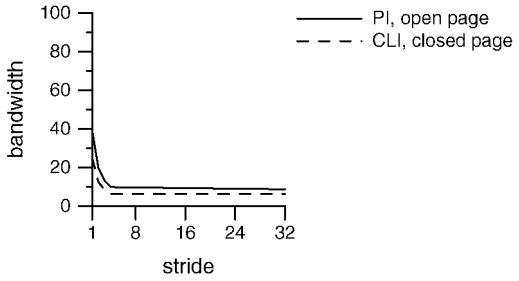

Fig. 14. Effective bandwidth for cache line fills.

are smaller than for CLI systems. In contrast to our analytic performance bounds for fast-page mode systems, the RDRAM bounds do not help in calculating appropriate FIFO depths for a computation: The best FIFO depth must be chosen experimentally.

Vector alignment has little impact on effective bandwidth for SMC systems with CLI memory organizations, as evidenced by the nearly identical performances for the simulations of maximal and minimal numbers of bank conflicts on systems with FIFOs deeper than 16 elements. A larger performance difference arises between the maximum and minimum bank-conflict simulations for SMC systems with PI memory organizations and FIFO depths of 32 elements or fewer. With deep FIFOs (64-128 elements) and long vectors, the SMC can deliver good performance even for a suboptimal data placement, yielding over 89 percent of the attainable bandwidth for all benchmarks.

For computations on shorter streams, the SMC can deliver nearly the effective bandwidth defined by the startup-delay bound. In fact, SMC performance approaches the bandwidth limits in all cases except for the 1K-element vector kernels on PI systems with an open-page policy. This difference in the SMC's ability to exploit available bandwidth results from our simple MSU scheduling policy. When the MSU's current request misses the RDRAM page, it must initiate a precharge operation before it can access the data. This means that the first access of each stream incurs a precharge delay, as does every access that crosses an RDRAM page boundary and every access to a busy bank (where the precharge cannot be overlapped with other activity). Furthermore, when we switch pages, the MSU must issue a row-activation command packet in addition to the column access packet. These overhead costs occur frequently and thus have a significant impact on long-stream performance. A scheduling policy that speculatively precharges a page and issues a row-activation command before the stream crosses the page boundary would mitigate some of these costs, as would an MSU that overlaps activity for another FIFO with the latency of the precharge and row activate commands.

As stride increases, any RDRAM controller becomes hampered in its ability to deliver peak performance. Data types and strides that do not densely occupy the 128-bit data packets can only exploit a fraction of the RDRAM's bandwidth. To put this in perspective, Fig. 14 shows the maximum percentage of peak bandwidth that cache-line fills in the computation's natural order can deliver when reading single streams. The solid and dotted lines indicate performance bounds for CLI and PI systems, respectively.

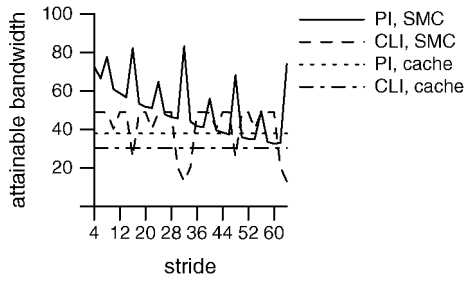

Fig. 15. Attainable bandwidth for *vaxpy* pattern and varying strides.

The effective bandwidth drops as vector stride increases up to the cache-line size and, once the stride exceeds the number of words in the cache line, the performance limits remain constant. For these larger strides, cache-line fills in the natural order only deliver 10 percent or less of the Direct RDRAM's potential bandwidth.

Increasing the number of non-unit-stride streams accessed increases the potential to exploit the parallelism supported by the Direct RDRAM interface, just as it did for unit-stride streams of Fig. 13. To illustrate this, Fig. 15 compares performances for the access pattern of the *vaxpy* kernel on vectors of length 1,024. The FIFO depth in these experiments is 128 elements. The y axis in Fig. 15 indicates the percentage of attainable bandwidth which, for non-unit-strides, is 50 percent of the peak system bandwidth. Performance for the SMC systems is sensitive to the stride of the computation, which determines the number of bank conflicts that the MSU will suffer. For PI systems and computations with strides over about 40, using cache-line fills in the computation's natural order may beat using an SMC with the current, simplistic reordering scheme. For smaller strides, and for some advantageous strides larger than 40, the SMC delivers significantly better performance than the cache can—up to 2.2 times the maximum effective bandwidth of the traditional memory controller. For CLI systems, the SMC delivers up to 1.6 times the bandwidth of the traditional approach, but performs worse for strides that are multiples of 16. Using cache-line fills for these strides is likely to create more cache conflicts because the vectors leave a larger footprint, but measuring the negative performance impact of these conflicts is beyond the scope of this study.

7 Data Coherence

The addition of the Stream Memory Controller using a noncaching path to main memory introduces the problem of data coherence between cache and the Stream Buffer Unit or between separate FIFOs in the SBU. Some mechanism must ensure coherence between the different components in the memory hierarchy, either by simply mapping stream pages as noncacheable or providing more sophisticated support from the compiler, the hardware, or some combination thereof. The design presented here assumes that coherence is maintained by software that guarantees stream data are not resident in cache when the SMC is active, perhaps by flushing the cache before streaming starts (which is only cost-effective for long streams). A snooping mechanism could be used to update or invalidate stale data, but this would require modifications to the SBU design and would add to the hardware cost of the SMC. Given that supporting out-of-order access to stream elements by the processor requires similar modifications and requires abandoning the simple FIFO SBU model, a hardware coherence mechanism might be attractive in that context.

The most effective solutions to the coherence problem will likely involve a combination of hardware and software. Programmable caches allow the compiler to manage coherence through software. This requires at least two operations: *invalidate* and *post* (which copies a value back to main memory). Cytron et al. [12] develop algorithms to determine when a cached value must update its shared variable or when a cached value is potentially stale. Their work shows how automatic techniques can effectively manage software-controlled caches.

Some decisions that cannot be made at compile-time can be made dynamically. For instance, the compiler could generate two versions of a loop body and insert run-time checks to determine which one to execute, avoiding streaming if there were potential aliasing problems. Another possibility is to allow programmer directives to specify whether streaming is safe for a given vector. These last two solutions can be used to avoid data dependences (and thus coherence problems) between streams within the SMC.

8 Switching Contexts

The additional hardware in SMC systems introduces a potentially large amount of state per process. If the SMC is only used by one process at a time, then there is no need to save its state when the operating system switches contexts. If the SMC is shared, then the two main questions to address are:

- How much state should be (or must be) saved? and
- When should (must) it be saved?

One solution is simply to discard data in read FIFOs since they can be refetched the next time the process runs. Other strategies become possible if the operating system need not implement precise interrupts for context switches. For instance, the SMC could be instructed to stop prefetching stream operands, but execution of the process could continue until at least one of the read FIFOs is drained.

Data in write FIFOs must be flushed to memory before a new process begins writing data to the SMC, but these writes could be overlapped with the loading of the new process's context. Alternatively, shadow write buffers could be added to hold the data being flushed, allowing the new process to use the SMC sooner. Whether or not the expense of such a scheme would be justified is an open question. Of course, the state of each FIFO (current address, operand count remaining, stride) must be saved as well. The SMC could even be used for saving and restoring the contexts themselves.

Another option is to extend virtual memory management techniques to include the SMC context and to do demand paging. Whenever a process with an invalid context mapping attempts to access the SMC, a page fault is

generated and a device driver saves the current device state, loads the new state, and validates the context mapping of the new process. This is similar to the approach taken in Kilgard et al.'s implementation of OpenGL direct rendering [28]. Mainwaring and Culler's network interface also performs demand paging of process contexts, maintaining a cache of the most recently used contexts within the controller [33]. Some network interface controllers (e.g., Myrinet [3]) already include the necessary extra hardware to manage contexts. Adopting this approach for the SMC would greatly increase the hardware requirements, but might represent an attractive, scalable solution in a system with a large transistor budget.

9 Conclusions

By combining compile-time detection of streams with execution-time selection of the access order and issue, we achieve near-optimal bandwidth for vector-like accesses relatively inexpensively. The SMC can even deliver vector-like memory performance for streamed computations whose data recurrences prevent vectorization [4]. This technique can complement more traditional cache-based schemes and help alleviate the memory bottleneck. We have studied dynamic access ordering within the context of uniprocessor systems with fast-page mode DRAM and Direct Rambus DRAM memories, but the technique may also be applied to other systems in terms of both computing platform and memory technology. For instance, our investigations indicate that the SMC concept can be effectively applied to shared-memory multiprocessor systems, but that a more complex ordering strategy is required for such systems to achieve uniformly high performance [34].

The dynamic access ordering hardware described here is both feasible and efficient to implement: It neither increases the processor's cycle time nor lengthens the path to memory for non-stream accesses. The hardware complexity is a function of the number and size of the stream buffers (implemented as FIFOs) and SMC placement (whether or not it is integrated into the processor chip). Our prototype ASIC for fast-page mode DRAM memories uses about 150,000 transistors, a modest number compared to the many millions used in current microprocessors. The SMC's control state machines are relatively small and, thus, the transistor count should scale approximately linearly with the maximum FIFO depth. Using commercially available memory parts and only a few hundred words of buffer storage, our prototype demonstrates that an SMC system can deliver nearly the full memory system bandwidth. Moreover, it does so without heroic compiler technology.

When accessing streams on single-device Direct Rambus memory systems, we find that a page-interleaved memory organization can deliver higher effective bandwidth for cache-line fills than a cache-line interleaved organization, although the latter should deliver better performance for nonstream accesses. By adding hardware support for streaming in the form of an SMC, we improve the performance of both memory organizations, allowing computations on long streams to utilize nearly all of the available memory bandwidth. The Direct RDRAM SMC systems described here implement the same, simple scheduling scheme that our fast-page mode DRAM systems use. More sophisticated access ordering mechanisms warrant further study to evaluate the robustness of their performances and the complexity of the corresponding hardware.

Acknowledgments

This work was supported in part by US National Science Foundation awards MIP-9114110 and MIP-9307626 and by a grant from Intel Corporation. Intel also contributed the i860 processor and board. The Oregon Graduate Institute of Science and Technology provided resources and partial support for Sally McKee during part of this research. The authors thank Assaji Aluwihare, Alan Batson, Ben Clark, Trevor Landon, Sean McGee, Chris Oliver, Bob Ross, Adam Szymkowiak, and Kenneth Wright for their many contributions to this project.

References

[1] M.J. Alexander, M.W. Bailey, B.R. Childers, J.W. Davidson, and S. Jinturkar, "Memory Bandwidth Optimizations for Wide-Bus Machines," *Proc. IEEE 26th Hawaii Int'l Conf. Systems Sciences (HICSS-26),* pp. 466-475, Jan. 1993. (Incorrectly published under M.A. Alexander et al.).

[2] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proc. Supercomputing '91,* pp. 176-186, Nov. 1991.

[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W.-K. Su, "Myrinet—A Gigabit-per-Second Local-Area Network," *IEEE Micro,* vol. 15, no. 1, pp. 29-36, Feb. 1995.

[4] M.E. Benitez and J.W. Davidson, "Code Generation for Streaming: An Access/Execute Mechanism," *Proc. Fourth Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 132-141, Apr. 1991.

[5] D. Burger, J.R. Goodman, and A. Kägi, "The Declining Effectiveness of Dynamic Caching for General-Purpose Microprocessors," Technical Report 1261, Univ. Wisconsin, 1995.

[6] J. Brooks, "Single PE Optimization Techniques for the Cray T3D System," *Proc. First European T3D Workshop,* Sept. 1995.

[7] P.L. Bird and R.A. Uhlig, "Using Lookahead to Reduce Memory Bank Contention for Decoupled Operand References," *Proc. Supercomputing '91,* pp. 187-196, Nov. 1991.

[8] W.Y. Chen, R.A. Bringmann, S.A. Mahlke, R.E. Hank, and J.E. Sicolo, "An Efficient Architecture for Loop Based Data Preloading," *Proc. IEEE/ACM 25th Int'l Symp. Microarchitecture,* pp. 92-101, Dec. 1992.

[9] J. Corbal, R. Espasa, and M. Valero, "Command Vector Memory Systems: High Performance at Low Cost," *Proc. 1998 Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 68-77, Oct. 1998.

[10] T.-C. Chiueh, "Sunder: A Programmable Hardware Prefetch Architecture for Numerical Loops," *Proc. Supercomputing '94,* pp. 488-497, Nov. 1994.

[11] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama, "Impulse: Building a Smarter Memory Controller," *Proc. Fifth Ann. Symp. High Performance Computer Architecture,* pp. 70-79, Jan. 1999.

[12] R. Cytron, S. Karlovsky, and K.P. McAuliffe, "Automatic Management of Programmable Caches," *Proc. 1988 Int'l Conf. Parallel Processing,* pp. 229-238, Aug. 1988.

[13] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proc. Fourth Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 40-52, Apr. 1991.

[14] W.Y. Chen, S.A. Mahlke, and W.W. Hwu, "Tolerating Data Access Latency with Register Preloading," *Proc. 1992 Int'l Conf. Supercomputing,* pp. 104-113, Sept. 1992.

[15] R. Crisp, "Direct Rambus Technology: The New Main Memory Standard," *IEEE Micro,* vol. 17, no. 6, pp. 18-28, Nov./Dec. 1997.
[16] D.R. Cheriton, G.A. Slavenburg, and P.D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," *Proc. 13th Ann. Int'l Symp. Computer Architecture,* Dec. 1986.
[17] A.M. del Corral and J.M. Llaberia, "Access Order to Avoid Inter-Vector Conflicts in Complex Memory Systems," *Proc. Ninth Int'l Parallel Processing Symp.,* 1995.
[18] J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Math. Software,* vol. 16, no. 1, pp. 1-17, Mar. 1990.
[19] F. Dahlgren, M. Dubois, and P. Stenstrom, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems,* vol. 6, no. 7, pp. 733-746, July 1995.
[20] J.W. Davidson and S. Jinturkar, "Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses," *Proc. SIGPLAN '94 Conf. Programming Language Design and Implementation,* pp. 186-195, June 1994.
[21] K.I. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "Memory-System Design Considerations for Dynamically-Scheduled Processors," *Proc. 24th Ann. Int'l Symp. Computer Architecture,* pp. 133-143, June 1997.
[22] J.W.C. Fu and J.H. Patel, "Data Prefetching in Multiprocessor Vector Cache Memories," *Proc. 18th Ann. Int'l Symp. Computer Architecture,* pp. 54-65, May 1991.
[23] Q.S. Gao, "The Chinese Remainder Theorem and the Prime Memory System," *Proc. 20th Ann. Int'l Symp. Computer Architecture,* pp. 337-340, May 1993.
[24] D.T. Harper III and J.R. Jump, "Vector Access Performance in Parallel Memories Using a Skewed Storage Scheme," *IEEE Trans. Computers,* vol. 36, no. 12, pp. 1,440-1,449, Dec. 1987.
[25] S.I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor, and W.A. Wulf, "Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory," *Proc. Fifth Ann. Symp. High Performance Computer Architecture,* pp. 80-89, Jan. 1999.
[26] Intel Corp., *i860 64-bit Microprocessor Programmer's Manual,* 1990.
[27] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," *Proc. 17th Ann. Int'l Symp. Computer Architecture,* pp. 364-373, May 1990.
[28] M.J. Kilgard, D. Blythe, and D. Hohn, "System Support for OpenGL Direct Rendering," *Proc. Graphics Interface '95,* May 1995.
[29] R.K. Koeninger, M. Furtney, and M. Walker, "A Shared-Memory MPP from Cray Research," *Digital Technical J.,* vol. 6, no. 2, pp. 8-21, 1994.
[30] A.C. Klaiber and H.M. Levy, "An Architecture for Software-Controlled Data Prefeching," *Proc. 18th Ann. Int'l Symp. Computer Architecture,* pp. 43-53, May 1991.
[31] D. Loshin and D. Budge, "Breaking the Memory Bottleneck, Parts 1 & 2," *Supercomputing Review,* Jan./Feb. 1992.
[32] K. Lee, *The NAS860 Library User's Manual,* NASA Ames Research Center, Mar. 1993.
[33] A.M. Mainwaring and D.E. Culler, "Design Challenges of Virtual Networks: Fast, General-Purpose Communication," *Proc. 1999 Conf. Principles and Practice of Parallel Programming,* pp. 119-130, May 1999.
[34] S.A. McKee, "Maximizing Memory Bandwidth for Streamed Computations," PhD thesis, School of Eng. and Applied Science, Univ. of Virginia, May 1995.
[35] F.H. McMahon, "The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range," Technical Report UCRL-53745, Lawrence Livermore Nat'l Laboratory, Dec. 1986.
[36] MIPS Technologies, Inc., *MIPS R10000 Microprocessor User's Manual, Version 2.0,* Dec. 1996.
[37] T.C. Mowry, M.S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proc. Fifth Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 62-73, Oct. 1992.
[38] B.K. Mathew, S.A. McKee, J.B. Carter, and A. Davis, "Parallel Access Ordering for SDRAM Memories," *Proc. Sicth Ann. Symp. High Performance Computer Architecture,* pp. 39-48, Jan. 2000.
[39] L. Meadows, S. Nakamoto, and V. Schuster, "A Vectorizing Software Pipelining Compiler for LIW and Superscalar Architectures," *Proc. RISC '92,* pp. 331-343, 1992.
[40] S.A. Moyer, "Access Ordering Algorithms and Effective Memory Bandwidth," PhD thesis, School of Eng. and Applied Science, Univ. of Virginia, May 1993.
[41] S.A. McKee and W.A. Wulf, "Access Ordering and Memory-Conscious Cache Utilization," *Proc. First Ann. Symp. High Performance Computer Architecture,* pp. 253-262, Jan. 1995.
[42] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," *Proc. 21st Ann. Int'l Symp. Computer Architecture,* pp. 24-33, May 1994.
[43] S. Palacharla and R.E. Kessler, "Code Restructuring to Exploit Page Mode and Read-Ahead Features of the Cray T3D," Cray Research Internal Report, Feb. 1995.
[44] Rambus, Inc., "Direct Rambus Technology Overview," 1997. http://www.rambus.com/html/documentation.html.
[45] Rambus, Inc., "64M/72M Direct RDRAM Data Sheet,"DL 0035-00.c0.5.28, Mar. 1998. http://www.rambus.com/html/documentation.html.
[46] B.R. Rau, "Pseudo-Randomly Interleaved Memory," *Proc. 18th Ann. Int'l Symp. Computer Architecture,* pp. 74-83, May 1991.
[47] S. Scott, "Synchronization and Communication in the T3E Multiprocessor," *Proc. Seventh Symp. Architectural Support for Programming Languages and Operating Systems,* pp. 26-36, Oct. 1996.
[48] I. Sklenar, "Prefetch Unit for Vector Operation on Scalar Computers," *Computer Architecture News,* vol. 20, no. 4, pp. 31-37, Sept. 1992.
[49] M.R. Swanson, L.B. Stoller, and J.B. Carter, "Increasing TLB Reach Using Superpages Backed by Shadow Memory," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* pp. 204-213, June 1998.
[50] M. Valero, T. Lang, J.M. Llaberia, M. Peiron, E. Ayguade, and J.J. Navarro, "Increasing the Number of Strides for Conflict-Free Vector Access," *Proc. 19th Ann. Int'l Symp. Computer Architecture,* pp. 372-381, May 1992.
[51] M. Valero, T. Lang, M. Peiron, and E. Ayguade, "Conflict-Free Access for Streams in Multi-Module Memories," Technical Report UPC-DAC-93-11, Universitat Politecnica de Catalunya, Barcelona, Spain, 1993.
[52] W.A. Wulf and S.A. McKee, "Hitting the Wall: Implications of the Obvious," *Computer Architecture News,* vol. 23, no. 1, pp. 20-24, Mar. 1995.
[53] L. Zhang, J.B. Carter, W.C. Hsieh, and S.A. McKee, "Memory System Support for Image Processing," *Proc. 1999 Int'l Conf. Parallel Architectures and Compilation Techniques,* pp. 98-107, Oct. 1999.

Sally A. McKee received a BA degree from Yale University in 1985, an MSE degree from Princeton University in 1990, and a PhD degree from the University of Virginia in 1995, all in computer science. She is a research assistant professor of computer science at the University of Utah. Her current interests in computer architecture include performance modeling and analysis and the design of efficient, adaptable memory systems. She is a member of the IEEE Computer Society and the ACM.

William A. Wulf received his BS degree in engineering physics in 1961 and his MS degree in electrical engineering in 1963, both from the University of Illinois. He received a PhD degree in computer science from the University of Virginia in 1968. He is AT&T Professor of Engineering and Applied Science in the Department of Computer Science at the University of Virginia. He is currently on leave while serving as president of the National Academy of Engineering in Washington, D.C. His current interests include national science policy, undergraduate computer science curriculum reform, computer security, harware-software codesign, and computer architecture and performance analysis. He is a member of the National Academy of Engineering, a fellow of the ACM, a fellow of the IEEE, and a member of the American Academy of Arts and Sciences.

James H. Aylor received the BS, MS, and PhD degrees in electrical engineering from the University of Virginia in 1968, 1971, and 1977, respectively. He is a professor and chairman of the Department of Electrical Engineering and director of the Center for Semicustom Integrated Systems at the University of Virginia. His current interests include system-level modeling, concurrent error detection, automatic test pattern generation, hardware description languages, and very large scale integration system design. He is a fellow of the IEEE.

Robert H. Klenke received his BS degree in electrical engineering from the Virginia Military Institute in 1982 and his MS and PhD degrees in electrical engineering from the University of Virginia in 1989 and 1993, respectively. He is currently an associate professor of electrical engineering at the Virginia Commonwealth University. His research interests include system level modeling, hardware description languages, parallel algorithms for automatic test pattern generation, and high speed digital design. He is a senior member of the IEEE and a member of the IEEE Computer Society, Tau Beta Pi, and Eta Kappa Nu.

Maximo H. Salinas received the BS degree from the Massachusetts Institute of Technology in 1984 and the MS degree from the University of Virginia in 1990, both in electrical engineering. He is currently completing a PhD degree in electrical engineering at the University of Virginia on the modeling of computer instruction set architectures. He is a senior scientist at the Center for Semicustom Integrated Systems at the University of Virginia. His current interests are in the areas of computer architecture, VLSI, digital design methodology development, and microelectromechanical systems (MEMS). He is a member of the IEEE Computer Society, the Association for Computing Machinery, Eta Kappa Nu, and Tau Beta Pi.

Sung (Tony) I. Hong received his MS degree in electrical engineering from the University of Virginia in 1998. He currently works for Lockheed Martin Federal Systems in Manassas, Virginia.

Dee A.B. Weikle is a PhD student at the University of Virginia, where her research centers on memory systems architecture and the development of analytic approaches to complex memory hierarchy design and analysis. She received a BS degree in electrical engineering from Rice University in 1985 and an MS degree in computer science from the University of Virginia in 1996. She is a member of the IEEE Computer Society and the ACM.