Design and Evaluation of Dynamic Access Ordering Hardware

Sally A. McKee, Assaji Aluwihare, Benjamin H. Clark, Robert H. Klenke, Trevor C. Landon,
Christopher W. Oliver, Maximo H. Salinas, Adam E. Szymkowiak, Kenneth L. Wright,
Wm.A. Wulf, James H. Aylor

University of Virginia
Charlottesville, VA 22903
{mckee | bhc2f | klenke | cwoliver | msalinas | aes2n | wright | wulf | jha}@virginia.edu

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 caches 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 and evaluate a Stream Memory Controller system that combines compile-time detection of streams with execution-time selection of the access order and issue. The technique is practical to implement, using existing compiler technology and requiring only a modest amount of special-purpose hardware. With our prototype system, we have observed performance improvements by factors of 13 over normal caching.

1. INTRODUCTION

As has become painfully obvious, processor speeds are increasing much faster than memory speeds. To illustrate the current problem, consider the multiprocessor Cray T3D [Cra95]. The peak Dynamic Random Access Memory (DRAM) read bandwidth for each 150MHz DEC Alpha processor [DEC92] of this machine is 320 Mbytes/sec, or about one 64-bit word per four clock cycles. Unfortunately, the actual bandwidth may be as low as 28 Mbytes/sec — in other words, the processors can perform up to 42 instructions in the time it takes to read a single DRAM location. As Jeff Brooks explains, "the rates you see in [a T3D] application depend highly on how the memory is referenced" [Bro95].

This variance in performance occurs because the T3D's DRAMs can perform some access sequences faster than others. Even though the term "DRAM" was coined to indicate that accesses to any "random" location require about the same amount of time, most modern DRAMs provide special capabilities that result in non-uniform access times. For instance, nearly all current DRAMs (including the T3D's) implement a form of *fast-page mode* operation [Qui91,IEEE92].

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 in two ways, addressing both convenience and speed. 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. The observed access time for page misses is typically three to five times as long as page hits. For instance, on a Cray T3D system with 16Mbit DRAMs, each page is 2048 Kbytes, page hits take about 4 cycles (26ns), and the additional overhead for page misses takes about 15 cycles more (for about 125ns in all) [Pal95,Bro95]. Although the terminology is similar, DRAM pages should not be confused with virtual memory pages.

The order of requests strongly affects the performance of other common devices that offer speed-optimizing features (nibble-mode, static column mode, or a small amount of SRAM cache on chip) or exhibit novel organizations (Ramlink and the new synchronous DRAM designs) [IEEE92]. With an advertised bandwidth of 500 Mbyte/s, Rambus DRAMS are another interesting new memory technology [Ram92]. These bus-based systems are capable of delivering a byte of data every 2ns for a block of information up to 256 bytes long. Like page-mode DRAMs, Rambus devices use banks of sense amplifier latches to "cache" data on chip, and good performance requires exploiting these cache lines and burst-mode transfers. In fact, these devices offer no performance benefit for random access patterns: the latency for accesses that miss the cache lines is 150-200ns. For interleaved memory systems, the order of requests is important on another level, as well: accesses to different banks can be performed in parallel, and thus happen faster than successive accesses to the same bank. In general, memory system performance falls significantly short of the maximum whenever accesses are not issued in an appropriate order.

Caches can help bridge the processor-memory performance gap for parts of programs that exhibit high locality of reference, but many computations do not reuse data soon enough to derive much benefit from caching. Codes that linearly traverse long streams of vector-like data are particularly bandwidth-limited. Unfortunately, this comprises a large and important class of computations, including scientific vector processing, digital signal processing, multi-media (de)compression, text searching, and some graphics applications, to name a few. We describe a system that addresses the memory bandwidth problem for such *streaming computations* by dynamically reordering stream accesses to exploit memory system architecture and device features. The technique is practical to implement, using existing compiler technology and requiring only a modest amount of special-purpose hardware.

We built a "proof of concept" dynamic access ordering system composed of a 40MHz i860 host processor and a daughterboard

To Appear in Proc. International Conference on Supercomputing, Philadelphia, PA, May 25-28, 1996.

containing a 132-pin ASIC and associated DRAM memory. In this paper, we describe and evaluate our implementation of this hardware. Initial experiments with the system have yielded performance improvements by factors of 2-13 over normal caching.

2. THE STREAM MEMORY CONTROLLER

We describe our approach based on the simplified architecture of Figure 1. In this system, the compiler must detect the presence of streams (as in Benitez and Davidson's streaming algorithms [Ben91]) 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.

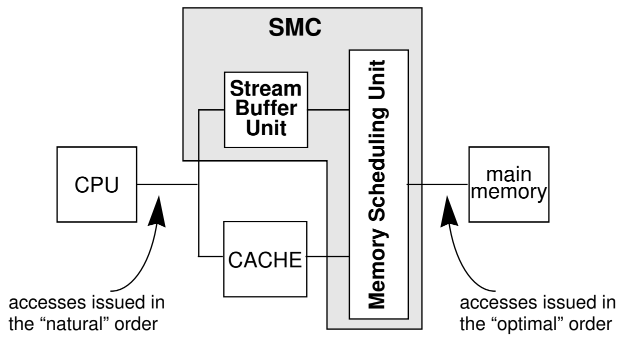

Figure 1 Stream Memory Controller System

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 non-stream 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 non-stream 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.

```
/* tridiagonal elimination: */
for (i = 1; i < n; i++)
        x[i] = z[i] * (y[i] - x[i-1]);

/* SMC version: */
/* first transmit stream params */
streamin(y+1, size, stride, n-1, FIFO0);
streamin(z+1, size, stride, n-1, FIFO1);
streamout(x+1,size, stride, n-1, FIFO2);

reg = x[0];            /* load x[0] */
for (i = 1; i < n; i++) {
        reg = *FIFO1 * (*FIFO0 - reg);
        *FIFO2 = reg;
}
```

Figure 2 SMC Programming Model

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. From the processor's perspective, the stream buffers are logically implemented as a set of FIFOs within the SBU, with each stream assigned to one FIFO. In principle, the MSU could access these buffers as random-access register files, asynchronously filling them from or draining them to memory; it turns out that for this uniprocessor implementation, having the MSU fill the buffers in FIFO order simplifies the hardware without sacrificing performance. The processor references the next element of a stream via the memory-mapped register representing the corresponding FIFO head. This memory mapping avoids the need to modify the processor's instruction set to address these registers. Figure 2 illustrates the SMC programming model for tridiagonal elimination, one of the Livermore Loops [McM86].

3. EXPERIMENTAL IMPLEMENTATION

In order to demonstrate the viability of dynamic access ordering, we have developed an experimental Stream Memory Controller system. This proof-of-concept version is implemented as a single, semi-custom VLSI integrated circuit interfaced to an Intel i860 host processor [Int91]. The SMC ASIC was designed 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 [Cas93,Men93]. The i860 was selected because it was readily available, and because it provides load/store instructions that bypass the cache. The processor uses the non-caching instructions to access the stream buffers.

Figure 3 depicts the architecture of our prototype system, which consists of an i860 motherboard provided by Intel Corporation interfaced via an expansion connector to an SMC daughterboard. The motherboard contains an i860XP processor, a system boot EPROM, a memory controller that is optimized for cache-line fills, and 16 MBytes of page-mode DRAM. The daughterboard contains the SMC and its memory subsystem, along with a pipeline stage needed to meet timing and line-length constraints.

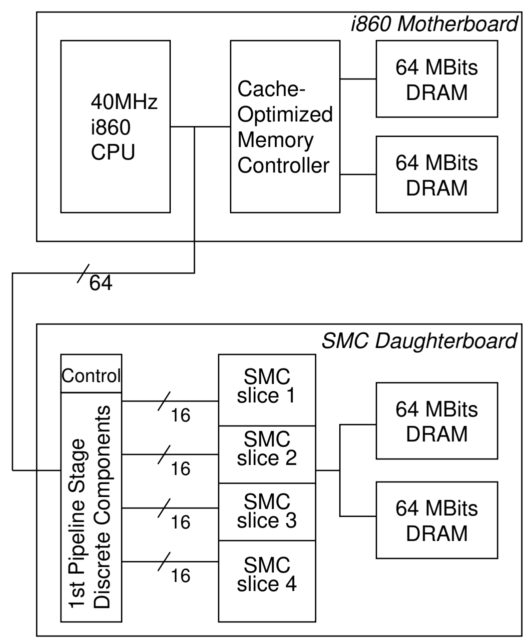

Figure 3 SMC System Architecture

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 the organization is logically the same. We use the i860's on-chip cache, but in this prototype, the SMC cannot access the original 16MBytes of i860 memory. Both the normal cache and the SMC can access the memory on the daughterboard. In order for our results to be representative of what would be seen in a real implementation, we use the daughterboard memory for all our experiments. This limitation is merely an artifact of using the existing i860 board instead of integrating both CPU and SMC on a single chip. In a "real" implementation all memory would be accessible from both the SMC and cache.

3.1 PROTOTYPE DAUGHTERBOARD

Each bank of DRAM memory on the daughterboard is composed of two 32 MBit 60ns page-mode components with 1 Kbyte pages. The minimum cycle time for fast page-mode accesses is 35ns, and random accesses require 110ns. Wait states make the SMC's observed access time for sustained accesses 50ns for page hits (2 CPU cycles) and 175ns for page misses (7 CPU cycles — this includes the time to precharge and set up the new DRAM page). With two interleaved banks, the SMC can deliver one data item every 25ns processor cycle.

The processor takes approximately 14ns to assert its address and cycle definition pins, and the signals take another 5ns to propagate to the expansion connector. This leaves less than 6ns 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 1 inch before the first level of logic on the daughterboard. In light of these constraints, we added a single-stage, bidirectional pipeline to the daughterboard; 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.

This off-chip implementation thus incurs pipeline delays in addition to bus turn-around delays when switching between reading and writing — delays that would not be present in an on-chip SMC. Nonetheless, the performance of our prototype SMC represents a significant improvement over the performance of a non-SMC system for stream accesses.

3.2 VLSI CHIP

Our prototype Stream Memory Controller is a 132-pin ASIC implemented in a 0.75 µm, three-level metal HP 26B process fabricated through MOSIS. The 40MHz i860 host processor can initiate a new transaction every other clock cycle, and quadword instructions allow the i860 to read 128 bits of data in two consecutive clock cycles. The SMC can thus deliver a 64-bit doubleword of data every cycle.

As illustrated in Figure 3, the prototype SMC is implemented as a 4-way bit-sliced system. We chose this organization over a full 64-bit wide version because the latter would have been severely pad-limited in size. Figure 4 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 Figure 4 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 non-stream (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 composed of two sections: the dual-ported SRAM buffers used to implement the FIFOs, and the FIFO controller state machine used to generate the addresses for Memory Scheduling Unit (MSU) accesses to the FIFOs. The FIFO controller logic provides signals conveying "fullness" information per FIFO to both the BC and the PBI. 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.

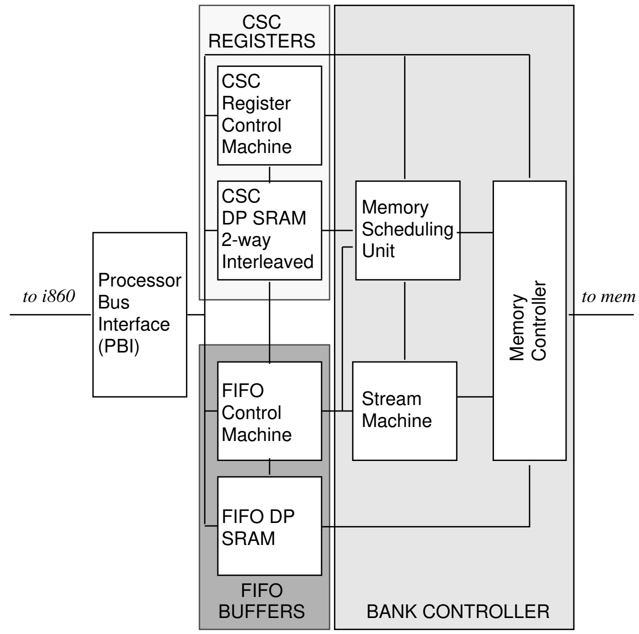

Figure 4 SMC ASIC Architecture

The current 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 FIFOs that can each be set to read or write. The software-programmable FIFO depth can be set to 8, 16, 32, 64, or 128 elements. The prototype SMC'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 eligible FIFO. Despite its simplicity, this ordering strategy works well in practice. For uniprocessor systems, its performance is competitive with that of more sophisticated policies. More intelligent schemes are required to achieve uniformly good performance on computations involving streams with strides that do not hit all memory banks [McK95b].

Further details of the design, implementation, and testing of the ASIC and daughterboard can be found elsewhere [McG94,Lan95, SMC96].

4. PERFORMANCE

Figure 5 lists the benchmark kernels used to generate the results presented here. *Daxpy, copy, scale,* and *swap* are from the BLAS (Basic Linear Algebra Subroutines) [Don90], and *tridiag* is tridiagonal gaussian elimination, the fifth Livermore Loop [McM86]. *Vaxpy* denotes a "vector axpy" operation that occurs in matrix-vector multiplication by diagonals: a vector *a* times a vector *x* plus a vector *y*. For our purposes, the actual computations in these loops are unimportant. We focus instead on the access patterns, and these benchmarks are representative of the access patterns found in real codes. For instance, *copy* and *scale* are the memory access patterns of JPEG operations in multimedia applications.

kernel	operation	vectortype
copy	`for (i = 0; i < N; i++)` `  y[i] = x[i];`	x: read y: write
daxpy	`for (i = 0; i < N; i++)` `  y[i] = a * x[i] + y[i];`	x: read y: read-write
scale	`for (i = 0; i < N; i++)` `  x[i] = a * x[i];`	x: read-write

swap	`for (i = 0; i < N; i++) {`
	`  t = x[i]; x[i] = y[i]; y[i] = t;`
	`}`
		x: read-write
		y: read-write

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

Figure 5 Benchmark Kernels

We present our results as a *percentage of peak bandwidth* — that which would be achieved if the CPU could perform one memory access each processor cycle. Unless otherwise noted, the vectors we consider here are of equal length, unit stride, share no DRAM pages in common, and are aligned to begin in the same bank. In order 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. By unrolling loops 16 times for our SMC experiments, we stress the SMC as much as possible and minimize bus turn-around delays from going off-chip (this would not be needed for an on-chip SMC). We execute each loop prior to beginning our measurements so that the experiment can run entirely out of the instruction cache.

Figure 6 illustrates the measured performance of our prototype system on each of the kernels for vectors of 16 to 8192 elements. These graphs show the percentage of the peak system bandwidth exploited for each benchmark. Figure 7 presents the same information in a format that helps put the bandwidth percentages into perspective for this particular machine: the average number of processor cycles per stream access. The dashed lines labeled "limit" indicate limits to attainable bandwidth due to SMC startup costs and unavoidable page misses (see [McK95b,SMC96] for derivations of performance bounds). The solid lines indicate the performance of our hardware prototype. The dotted lines indicate the performance measured when using "normal" 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 non-caching pipelined floating point load (*pfld*) instruction.

The effective bandwidth delivered by the SMC for these kernels is between 214% and 375% of that delivered by normal caching. Performing the computation in the natural order with caching accesses yields less than 32% of the system's peak bandwidth for all access patterns. For the two multiple-vector kernels that both read and write a vector (*daxpy* and *vaxpy*), cache performance falls off when vector length exceeds the cache size, causing modified cache lines to be written to memory as they are evicted. Since all modified lines hit the current DRAM page for *scale*, cache performance for this kernel using long vectors does not visibly suffer. *Scale*'s cache performance would be higher if the i860's cache controller could take more advantage of fast-page mode. Unfortunately, every cache-line fill incurs the DRAM page-miss overhead, regardless of whether the data lies in the same page as the previous access.

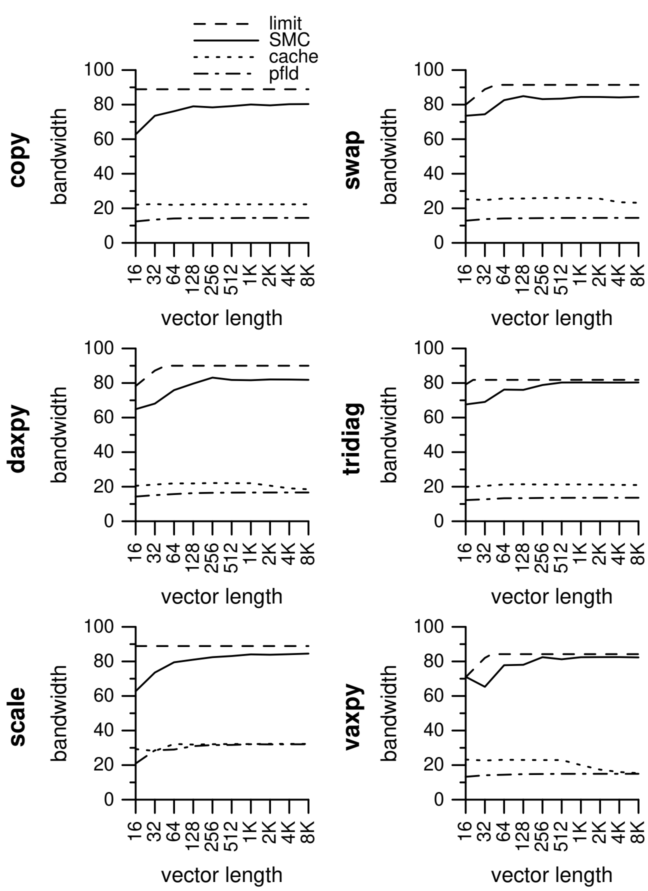

Figure 6 Percentages of Peak Bandwidth

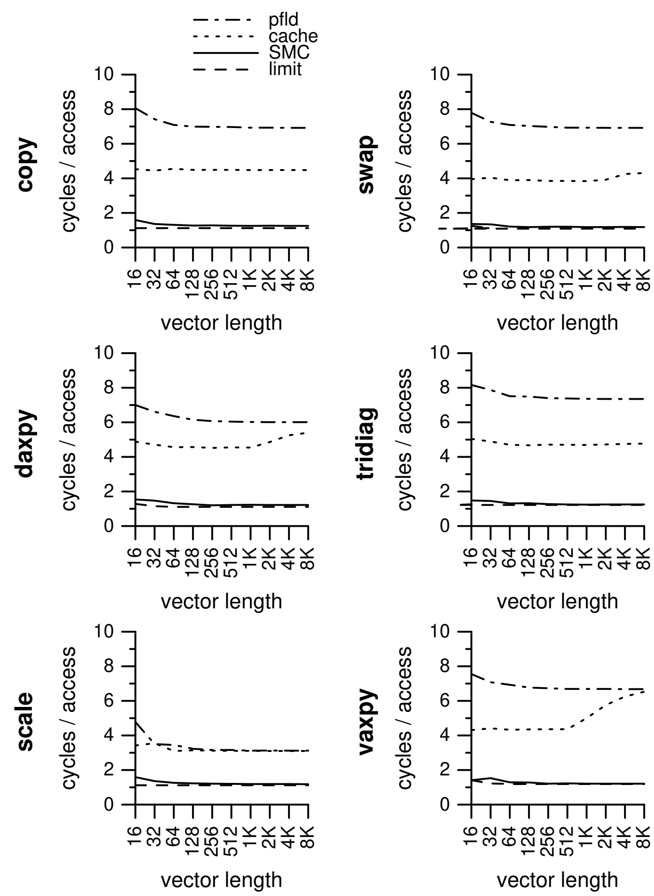

Figure 7 Cycles Per Access

Looking at it from the perspective of Figure 7, the SMC brings the average cycles per memory access very close to the minimum one cycle for long-vector computations. In contrast, the average number of cycles per access when using the cache is more than three in all cases. The disparity between SMC and cache performance is even more dramatic for non-unit stride computations, where each cache-line fill fetches unneeded data. Figure 8 illustrates the percentages of peak bandwidth delivered by the SMC versus caching and non-caching accesses for the *vaxpy* kernel with stride-five vectors. For these vectors, the SMC delivers between 10.4 and 13.2 times the effective bandwidth of the cache.

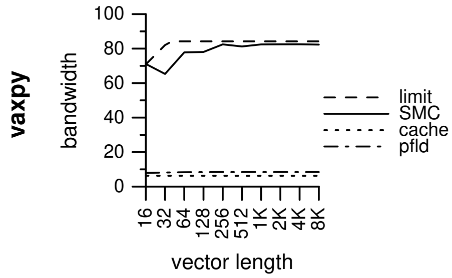

Figure 8 Stride-5 Vectors of Increasing Lengths

When non-caching instructions are used to access unit-stride vectors in the natural order of the computation, performance is generally even worse than for caching loads. The benchmarks using the *pfld* instruction could take more advantage of fast-page mode if the loops were unrolled and accesses to each stream were grouped together, so that several accesses that hit the current page would be issued after every DRAM page miss. This optimization would also minimize delays due to switching the memory bus between reading and writing.

The patterns of the performance curves for different benchmarks are almost boringly similar. Our results indicate that variations in the processor's reference sequence have little effect on the SMC's ability to improve bandwidth. In Figure 6, the slight dips in the SMC performance curves at 32-element vectors for the *swap*, *tridiag* and *vaxpy* kernels 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. The shorter the vectors, the greater the impact each page miss has on overall performance: amortizing the last page miss over just a few more accesses can make a noticeable difference. If we plot points for vectors of every length, we see a saw-tooth shape, the "teeth" of which get smaller as vector length grows and the page misses are amortized over more total accesses. Figure 9 shows this detail in the simulation performance for *copy*.

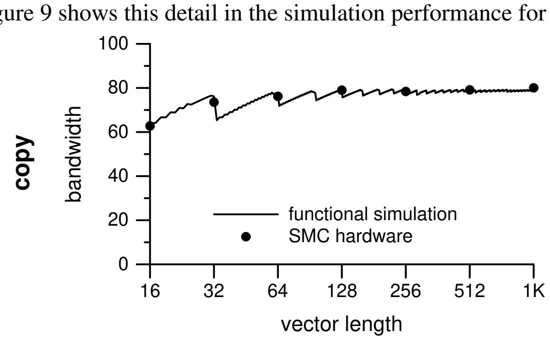

Figure 9 Performance Detail for Increasing Lengths

We built the MSU so that it fills a whole FIFO at a time — this 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 stream s 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.

So even though deeper FIFOs allow the MSU to get more data from a DRAM page each time it's made current, they cause the processor to wait longer at startup. The graphs in Figure 10 illustrate the net effect of these competing factors for our benchmark kernels using 128-element vectors. The dashed line shows the performance bounds defined by the SMC startup cost and the minimum number of DRAM page misses for this computation. The solid lines again represent results measured with the SMC hardware. The dotted and dot-dashed lines show performance when using "normal" caching load and non-caching (*pfld*) load instructions, respectively, 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.

Short-vector computations have fewer total accesses over which to amortize startup 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*, *swap*, *tridiag*, and *vaxpy*). The *copy* and *scale* kernels incur no initial delay, since they only read one stream. If we plotted performance for deeper FIFOs, the curves would be flat: bandwidth remains constant once FIFO size exceeds the vector length.

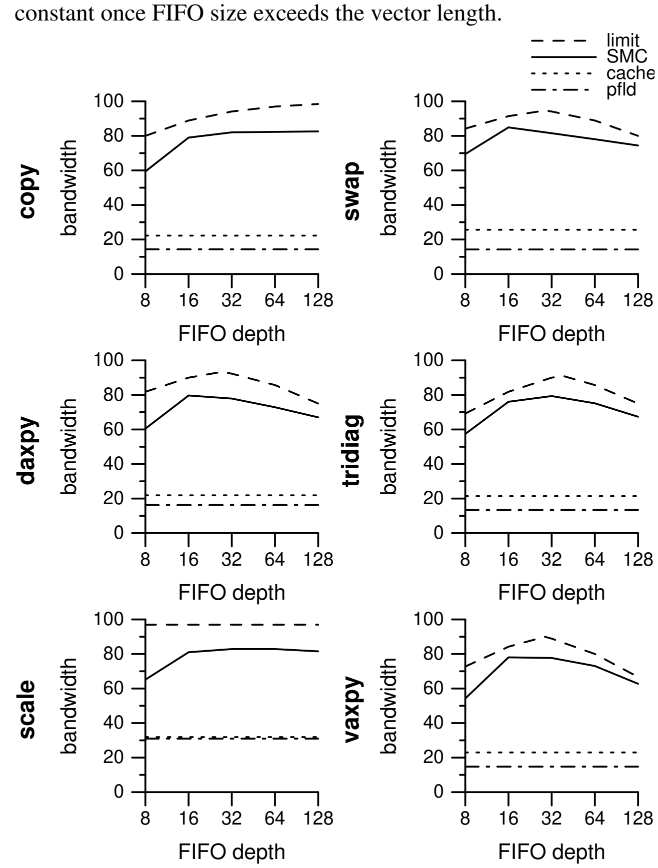

Figure 10 Short Vectors and Increasing FIFO Depths

As noted above, cache performance diminishes markedly for non-unit stride vectors. For computations using the i860's pipelined, non-caching load, performance for non-unit strides is approximately 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 *vaxpy* performance for stride five is essentially identical to that for stride one, whereas the cache and pfld performances decrease by factors of 3.7 and 1.8, respectively. Figure 11 illustrates this for 128-element vectors.

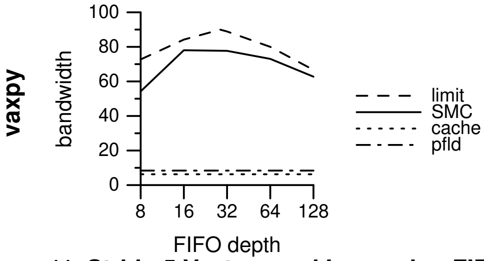

Figure 11 Stride-5 Vectors and Increasing FIFO Depths

The 2048-element vectors used to generate the results depicted in Figure 12 allow startup and page-miss costs to be amortized much more effectively than the 128-element vectors of Figure 10 and Figure 11. For the longer vectors, initial delays have very little effect on overall performance for the prototype SMC's range of FIFO depths. Performance of our off-chip implementation of the SMC reaches a maximum of about 90% of the peak system bandwidth, regardless of the computation parameters. This limit reflects the cost of transferring data across chip boundaries, and would not be present in an integrated CPU/SMC system.

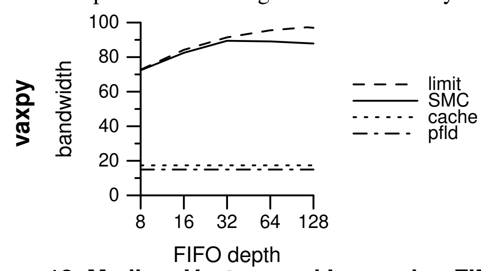

Figure 12 Medium Vectors and Increasing FIFO Depths

As our results illustrate, even an SMC with only a small amount of buffer space (16 elements) can consistently deliver over 80% 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% of the attainable bandwidth for vectors over 128 elements. With FIFOs of 32 elements, our proof-of-concept system delivers its maximum possible performance on vectors of only 2048 elements. Deeper FIFOs yield even better performance for computations on longer vectors: our simulations indicate that on-chip SMCs with FIFOs of 256 or 512 elements can exploit nearly the full system bandwidth [McK95b,SMC96].

5. RELATED WORK

The notion that performance of memory-intensive applications can be improved by reordering memory requests has been demonstrated before. Unique to our work is the premise that, for stream-like reference patterns, access ordering:

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

Our confidence that the SMC could be implemented efficiently was based on the fact that similar designs have been built. For instance, the organization of the SBU is almost identical to the stream units of the WM architecture [Wul92], and the SMC may be thought of as a special case of a decoupled access-execute architecture [Goo85,Smi87].

More complex stream buffers have been evaluated in other contexts. Jouppi performed simulation studies of stream buffers used to prefetch successive cache lines on a cache miss [Jou90], and Palacharla and Kessler investigate the use of a set of stream buffers as a replacement for secondary cache [Pal94]. Although the latter scheme generally increased the cache hit rates of the benchmarks they simulated, these improvements were achieved at the expense of increased main memory bandwidth requirements. In contrast, our approach attempts to exploit the existing memory bandwidth as much as possible, without increasing bandwidth requirements.

It is often possible to take advantage of memory component features by reordering memory accesses at compile time. For instance, the compiler optimizations of Alexander, et al., for wide-bus machines [Ale93] have the side-effect of exploiting DRAM features like fast-page mode. Moyer 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 [Moy93]. 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 non-caching load instructions) and then writing them to a pre-allocated portion of cache [Lee93]. Meadows describes a similar scheme for the PGI i860 compiler [Mea92], and Loshin and Budge give a general description of the technique [Los92]. A subset of these authors 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% over normal caching, depending on vector stride [McK95a].

Another option is to augment the purely compile-time approach with a hardware assist. Palacharla and Kessler [Pal95] investigate code restructuring techniques to exploit fast-page mode DRAMs via a hardware *read-ahead* mechanism on the Cray T3D. 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 and stores it in 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. 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. Palacharla and Kessler measure a performance improvement of up to 75% in two, three, and four-stream examples on a Cray T3D, and Brooks demonstrates a factor of 13 improvement (from 3.5 Mflops to 51.6 Mflops) in T3D performance after applying access ordering to a 3x3 matrix multiplication routine used in Quantum Chromo Dynamics (QCD) codes [Bro94].

The benefits of compile-time ordering schemes can be substantial, but their performance cannot rival that of a hardware scheme such as the one discussed here. The compiler cannot generate the optimal access sequence without the address alignment information that is usually only available at run time. For this reason, Palacharla and Kessler generate code to decide at run-time the extent to which their optimizations should be applied: they calculate the largest vector blocks that will not conflict in the cache. In general, any scheme using the cache to load stream data suffers from cache conflicts, and any scheme loading the data directly to registers suffers from register pressure. We have not yet measured the improvement in cache performance from removing the polluting stream references, but there is clearly a positive effect.

6. LESSONS LEARNED

Some of the results of our studies verified our expectations, while others initially surprised us. For instance, we proved (as expected) that the SMC can be fabricated with synthesis to meet its timing requirements and that it performs as predicted: the SMC delivers over 90% of the attainable bandwidth for long-vector computations, even though it resides on a separate chip from the CPU. We verified that the SMC yields up to 13.8 times the effective bandwidth delivered by the i860's own memory system. In addition, we reaped the benefits of a multifaceted approach to modeling and simulating our design before committing it to silicon. We employed five different models (two analytic models, one functional simulator, one gate-level hardware simulator, and one petri-net based system model) designed for different purposes by three different teams of people in two different academic departments. The common starting point was a description of the hardware interface and high-level (functional) behavior.

These models helped us to refine and validate each stage of our design, but several of their results were unexpected. In particular, we were initially surprised that FIFO depth must be tailored to the parameters of a particular computation. Long-vector computations benefit from very deep FIFOs, whereas computations on shorter streams require shallower FIFOs. We have derived algorithms that compilers can use to calculate an appropriate FIFO depth for a particular computation on a given system [McK95b,SMC96].

We were also surprised to discover that in many cases a relatively naive access-ordering policy performs competitively with a more sophisticated heuristic. The programmer or compiler can often arrange to avoid the situations in which a simple policy would perform poorly.

7. 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. This technique complements more traditional cache-based schemes, so that overall effective memory performance need not be a bottleneck.

The dynamic access ordering hardware described here is both feasible and efficient to implement. The SMC 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 it is integrated into the processor chip). Using commercially available memory parts and only a few hundred words of buffer storage, our proof-of-concept system demonstrates that an SMC system can deliver nearly the full memory system bandwidth. Moreover, it does so without heroic compiler technology. The current version features four moderate-size, adjustable-depth FIFOs and uses about 150,000 transistors; this is a relatively modest number of transistors when compared to the 3-10 million used in current microprocessors. The SMC's control state machines are relatively small, thus the transistor count will scale approximately linearly with increasing FIFO depth.

This prototype version places the SMC on a separate board from the processor, but for best performance, we believe the dynamic access ordering hardware should be integrated onto the processor chip, at the same level as an L1 cache. The next stage of the project targets building an SMC system in which the stream buffers reside on-chip with a high-speed, special-purpose microprocessor.

We have implemented dynamic access ordering within the context of memory systems composed of fast page-mode DRAMs, but the technique may be applied to other memory systems, as well. In addition to taking advantage of memory component features (for those devices that have non-uniform access times), prefetching read operands, and buffering writes, the SMC provides the same functionality as the conflict-avoidance hardware used in many vector computers (in fact, the SMC is more general, delivering good performance under a wider variety of circumstances). Furthermore, the SMC can achieve vector-like memory performance for streamed computations whose data recurrences prevent vectorization.

Preliminary investigations indicate that the SMC concept can be effectively applied to shared-memory multiprocessors, but that a sophisticated ordering strategy is required for such systems to achieve uniformly high performance.

ACKNOWLEDGMENTS

This work was supported in part by a grant from Intel Corporation and NSF grants MIP-9114110 and MIP-9307626. The authors wish to thank Dee A.B. Weikle, Bob Ross, and Alan Batson for their many contributions to this project.

REFERENCES

[Ale93] M.J. Alexander, M.W. Bailey, B.R. Childers, J.W. Davidson, and S. Jinturkar, "Memory Bandwidth Optimizations for Wide-Bus Machines", *Proc. 26th Hawaii International Conference on Systems Sciences*, January 1993, pages 466-475. (Incorrectly published under M.A. Alexander, et al.)

[Ben91] M.E. Benitez and J. W. Davidson, "Code Generation for Streaming: An Access/Execute Mechanism", *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pages 132-141.

[Bro95] J. Brooks, "Single PE Optimization Techniques for the Cray T3D System", *Proc. 1st European T3D Workshop*, September, 1995.

[Cas93] *Epoch User's Manual* 3.1, Cascade Design Automation, 1993.

[Cra95] "Cray T3D Massively Parallel Processing System", Cray Research, Inc., http://www.cray.com/PUBLIC/product-info/mpp/CRAY_T3D.html, 1995.

[Dec92] *Digital Technical Journal*, Digital Equipment Corporation, 4(4), Special Issue, 1992, http://www.digital.com/info/DTJ/axp-toc.html.

[Don90] J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling, "A set of Level 3 Basic Linear Algebra Subprograms", *ACM Transactions on Mathematical Software*, 16(1):1-17, March 1990.

[Goo85] J.R. Goodman, J. Hsieh, K. Liou, A.R. Pleszkun, P.B. Schechter, and H.C. Young, "PIPE: A VLSI Decoupled Architecture", *Proc. 12th International Symposium n Computer Architecture*, June 1985, pages 20-27.

[IEEE92] "Memory Catches Up", Special Report, *IEEE Spectrum*, 29(10):34-53, October 1992.

[Int91] *i860 XP Microprocessor Data Book*, Intel Corporation, 1991.

[Jou90] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proc. 17th International Symposium on Computer Architecture*, May 1990, pages 364-373. Also Digital Equipment Corporation, Western Research Lab, Technical Note TN-14, March 1990.

[Lan95] T.C. Landon, R.H. Klenke, J.H. Aylor, M.H. Salinas, and S.A. McKee, "An Approach for Optimizing Synthesized High-Speed ASICs", *Proc. of the IEEE International ASIC Conference*, Austin, TX, September 1995, pages 245-248.

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

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

[McG94] S.W. McGee, R.H. Klenke, J.H. Aylor, and A.J. Schwab, "Design of a Processor Bus Interface ASIC for the Stream Memory Controller", *Proc. of the IEEE International ASIC Conference*, Rochester, NY, September 1994, pages 462-465.

[McK95a]S.A. McKee, Wm.A. Wulf, "Access Ordering and Memory-Conscious Cache Utilization," *Proc. First International Symposium on High Performance Computer Architecture*, Raleigh, NC, January 1995, pages 253-262.

[McK95b]S.A. McKee, "Maximizing Memory Bandwidth for Streamed Computations", Ph.D. Dissertation, University of Virginia, Department of Computer Science, May 1995. http://www.cs.virginia.edu/research/techrep.html.

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

[Mea92] L. Meadows, S. Nakamoto, and V. Schuster, "A Vectorizing Software Pipelining Compiler for LIW and Superscalar Architectures", *Proc. RISC'92*, pages 331-343.

[Men93] *System-1076, Quicksim II User's Manual*, Mentor Graphics Corporation, 1993.

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

[Pal94] S. Palacharla and R.E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement", *Proc. 21st International Symposium on Computer Architecture*, May 1994, pages 24-33.

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

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

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

[Smi87] J.E. Smith, G.E. Dermer, B.D. Vanderwarn, S.D. Klinger, C.M. Roszewski, D.L. Fowler, and D.R. Scidmore, "The ZS-1 Central Processor", *Proc. 2nd International Conference on Architectural Support for Programming Languages and Systems (ASPLOS-II)*, Oct. 1987, pages 199-204.

[SMC96] SMC home page, http://www.cs.virginia.edu/~wm/smc.html.

[Wul92] Wm.A. Wulf, "Evaluation of the WM Architecture", *Proc. 19th International Symposium on Computer Architecture*, May 1992, pages 382-390.