# Experimental Implementation of Dynamic Access Ordering

Sally A. McKee, Robert H. Klenke,
Andrew J. Schwab, Wm. A. Wulf,
Steven A. Moyer, James H. Aylor,
Charles Y. Hitchcock

Computer Science Report No. CS-93-42
August 1, 1993

# Experimental Implementation of Dynamic Access Ordering

Sally A. McKee, Robert H. Klenke, Andrew J. Schwab,
Wm. A. Wulf, Steven A. Moyer, James H. Aylor[†]
University of Virginia

Charles Y. Hitchcock[‡]
Dartmouth College

## Abstract

*As microprocessor speeds increase, memory bandwidth is rapidly becoming the performance bottleneck in the execution of vector-like algorithms. Although caching provides adequate performance for many problems, caching alone is an insufficient solution for vector applications with poor temporal and spatial locality. Moreover, the nature of memories themselves has changed. Current DRAM components should not be treated as uniform access-time RAM: achieving greater bandwidth requires exploiting the characteristics of components at every level of the memory hierarchy.*

*This paper describes hardware-assisted access ordering and our hardware development effort to build a Stream Memory Controller (SMC) that implements the technique for a commercially available high-performance micro-processor, the Intel i860. Our strategy augments caching by combining compile-time detection of memory access patterns with a memory subsystem that decouples the order of requests generated by the processor from that issued to the memory system. This decoupling permits requests to be issued in an order that optimizes use of the memory system.*

## 1. Increasing vector memory bandwidth

Processor speeds are increasing much faster than memory speeds: microprocessor performance has increased by 50-100% per year in the last decade, while DRAM performance has risen only 10-15% each year [15]. As a result, memory bandwidth is becoming the limiting performance factor for many applications, particularly scientific computations.

Although the addition of cache memory is often a sufficient solution to the memory latency and bandwidth problems in general-purpose scalar computing, the vectors used in scientific computations are normally too large and are not reused soon enough to derive much benefit from caching. For computations in which vectors are reused, iteration space tiling [3][17][32] can partition the problems into cache-size blocks, but the technique is difficult to automate. Caching non-unit stride vectors may actually reduce a computation's effective memory bandwidth by fetching extraneous data.

The traditional scalar-processor concern has been to minimize memory latency in order to maximize processor performance. For scientific applications, however, the processor is not the bottleneck, and as processor speeds continue to increase relative to memory speeds, optimal system performance will leave the processor idle at times. Bridging this performance gap requires changing the way we think about the problem — to maximize bandwidth for scientific applications, we need to minimize *average* latency over a coherent set of accesses.

Although this paper focuses on scientific computations, they are by no means the only applications limited by memory bandwidth. The problem arises in any computation involving linear traversals of vector-like data, where each element is typically visited only once during lengthy portions of the computation. Other examples include string processing, image processing and other DSP applications, some database queries, some graphics applications, and DNA sequence matching.

In this paper we define *access ordering*, an augmentation of cache-based techniques for bridging the processor-memory performance gap. We then posit how this technique might be incorporated into a combined hardware/software solution to the bandwidth problem and explore the feasibility of this solution, highlighting our simulation study results and presenting a potential hardware design.

## 2. Access ordering

The assumptions made by many memory architectures simply don't match the physical characteristics of the devices used to build them. Memory components are often assumed to require about the same amount of time to access any random location; indeed, it was this uniform access

---

Appeared in Proc. 27th Hawaii International Conference on Systems Sciences (HICSS-27), Maui, HI, January 1994.

1

time that gave rise to the term RAM, for Random Access Memory. Many computer architecture textbooks ([2] and [11] among them) specifically cultivate this view. Others skirt the issue entirely [19][30].

Somewhat ironically, the assumption no longer applies to modern memory devices: most components manufactured in the last 10-15 years provide special capabilities that make it possible to perform some access sequences faster than others. For instance, nearly all current DRAMs implement a form of page-mode operation [24]. These devices behave as if implemented with a single on-chip cache line, or *page* (this should not be confused with a virtual memory page). A memory access falling outside the address range of the current DRAM page forces a new page to be accessed. The overhead time required to set up the new page makes servicing such an access significantly slower than one that hits the current page. Other common devices offer similar features, such as nibble-mode, static column mode, or a small amount of SRAM cache on chip. This sensitivity to the order of requests is exacerbated in several emerging technologies: for instance, Rambus [25], Ramlink, and the new DRAM designs with high-speed sequential interfaces [12] provide high bandwidth for large transfers, but offer little performance benefit for single-word accesses.

For multiple-module memory systems, the order of requests is important on yet another level: successive accesses to the same memory bank cannot be performed as quickly as accesses to different banks. To get the best performance out of such a system, we must take advantage of the architecture's available concurrency.

There are a number of hardware and software techniques that can help manage the imbalance between processor and memory speeds. These include altering the placement of data to exploit concurrency [9], reordering the computation to increase locality (as in "blocking" [17]), address transformations for conflict-free access to interleaved memory [10][26][31], software prefetching data to the cache [4][16][29], and hardware prefetching vector data to cache [1][7][14][27]. The key difference between these techniques and the complementary one we propose here is that we reorder accesses to exploit the architectural and component features that make memory systems sensitive to the sequence of requests.

To illustrate one aspect of the bandwidth problem — and how it might be addressed at compile time — consider the effect of executing the fifth Livermore Loop (tridiagonal elimination) using non-caching accesses to reference a single bank of page-mode DRAMs. Figure 1(a) represents the natural reference sequence for a straightforward translation of the computation:

$$\forall i \qquad x_i \leftarrow z_i \times (y_i - x_{i-1})$$

This computation contains a first-order linear recurrence, and therefore cannot be vectorized. Nonetheless, the compiler can employ Davidson and Benitez's recurrence detection and optimization algorithm [5] to generate streaming code: each computed value $x_i$ is retained in a register so that it will be available for use as $x_{i-1}$ on the following iteration. For modest size vectors, elements from $x$, $y$, and $z$ are likely to reside in different pages, so that accessing each vector in turn incurs the page miss overhead on each access; memory references likely to generate page misses are highlighted in the figure.

```
loop:                        loop:
    load z[i]                    load z[i]
    load y[i]                    load z[i+1]
    stor x[i]                    load y[i]
    jump loop                    load y[i+1]
                                 stor x[i]
                                 stor x[i+1]
                                 jump loop

        (a)                          (b)
```

**Figure 1  *tridiag* Code**

In the loop of Figure 1(a), a page miss occurs for every reference. Unrolling the loop and grouping accesses to the same vector, as in Figure 1(b), amortizes the page-miss cost over a number of accesses; in this case three misses occur for every six references. Reducing the page-miss count increases processor-memory bandwidth significantly. For example, consider a device for which the time required to service a page miss is four times that for a page hit, a miss/hit cost ratio that is representative of current technology. The natural-order loop in Figure 1(a) only delivers 25% of the attainable bandwidth, whereas the unrolled, reordered loop in Figure 1(b) delivers 40%.[1]

Figure 2 illustrates effective memory bandwidth versus depth of unrolling, using a page-miss/page-hit cost ratio of four. For the bottom curve, the loop body of Figure 1(a) is essentially replicated the appropriate number of times, as is standard practice; for the middle curve, accesses have been arranged as per Figure 1(b); and the top curve depicts the bandwidth attainable if all accesses were to hit the current DRAM page. Reordering the accesses realizes a performance gain of almost 130% at an unrolling depth of four, and over 190% at a depth of eight. If it were possible to unroll to a depth of 16, we could expect a performance increase of nearly 240%.

A comprehensive, successful solution to the memory bandwidth problem must therefore exploit the richness of

---

1. Other factors that may have minor effects on performance (e.g. bus turnaround delay when mixing reads and writes) are ignored here for the sake of simplicity.
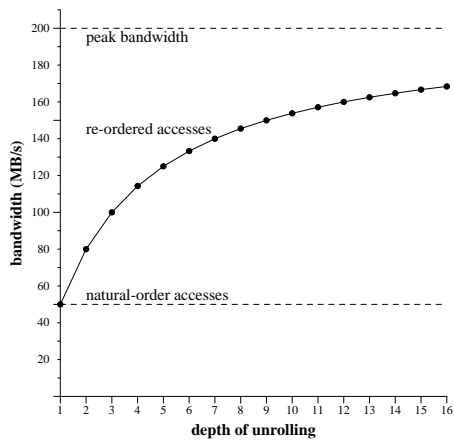
**Figure 2** *tridiag* **Memory Performance**

the *full* memory hierarchy, both its architecture and its component characteristics. One way to do this is via *access ordering*, which we define as any technique for changing the order of memory requests to increase bandwidth. Here we are especially concerned with ordering a set of vector-like "stream" accesses.

As our example illustrates, the performance benefits of doing static access ordering can be quite dramatic. Unfortunately, without the kinds of address alignment information that are usually only available at run time, the compiler can't generate the optimal access sequence. The extent to which a compiler can perform this optimization is further constrained by such things as the size of the processor register file (for instance, *tridiag* can be unrolled at most eight times on the i860). Moyer provides a thorough analysis of the performance and limitations of compile-time access ordering [23]. In light of both the impact of access ordering on effective memory bandwidth and the limitations inherent in implementing the technique statically, it makes sense to consider an implementation that reorders accesses dynamically at run time. We explore one such scheme in the remainder of this paper.

## 3. The Stream Memory Controller

Since there are a number of options for when and how the request ordering can be done, access ordering systems can be classified by three key components:

- stream detection, the recognition of streams accessed within a loop, along with their parameters (base address, stride, etc.),

- access ordering, the determination of that interleaving of stream references that most efficiently utilizes the memory system, and

- access issuing, the determination of when the load/store operations will be issued.

Each of these functions may be addressed at compile time or by hardware at run time. Based on our analysis and simulations, we believe that the best engineering choice is to detect streams at compile time, and to defer access ordering and issue to run time. Choosing this scheme over a strictly hardware, runtime-ordering system follows the philosophy that has guided the design of RISC processors — move work to compile time whenever possible. This helps to speed processing and minimize hardware. Here we describe in general terms how such a scheme might be incorporated into an overall system architecture.
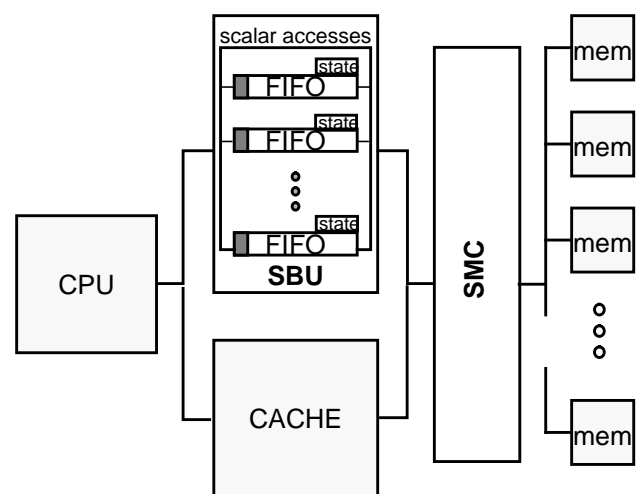


**Figure 3  Stream Memory Controller and Stream Buffer Unit**

The approach we suggest is generally applicable to any computing system, but will be described based on the simplified architecture of Figure 3. Memory is interfaced to the processor through a controller labeled "SMC" for Stream Memory Controller. For non-stream accesses, the SMC provides the same functionality and performance as a traditional memory controller. For streams, the SMC includes logic to determine the order of requests, and logic to issue those requests. A separate Stream Buffer Unit (SBU) provides registers that the processor uses to specify stream parameters (base address, stride, etc.) and high-speed buffers for stream operands. Note that the Stream Buffer Unit is *not* on the critical path to memory; the speed of non-vector accesses is not adversely affected by the addition of the SMC/SBU.

There are a number of options for the internal architecture of the SBU: here we describe one feasible organization. A set of memory-mapped registers provides a processor-independent way of specifying stream parameters. Setting these registers allows the processor to initiate an asynchronous stream of memory access operations for a set of string operands. Data retrieval from the streams (loads) and insertion into streams (stores) may

be done in any of several ways; for instance, the SBU could appear to be a traditional cache, or the model could include a set of FIFOs, as illustrated in Figure 3. Each stream is assigned to one FIFO, which is asynchronously filled from (or drained to) memory by the access/issue logic. The "head" of each FIFO is another memory-mapped register: the CPU's read instructions from or write instructions to a particular stream reference the FIFO head via this register, dequeueing or enqueueing data as is appropriate. Note that the buffers do not appear identical to the processor and the memory: for instance, the FIFOs could be implemented as random-access register files from the memory system's point of view.

This organization is both simple and practical from an implementation standpoint: similar designs have been built. In fact, the organization is almost identical to the "stream units" of the WM architecture [33], or may be thought of as a special case of a decoupled access-execute architecture [8][28]. Another advantage is that this combined hardware/software scheme doesn't require heroic compiler technology — the compiler need only detect the presence of streams, and Davidson and Benitez's streaming algorithm [5] can be used to do this. Note that the compiler is responsible for detecting data dependences.

## 4. Simulation results

In order to validate the SMC concept, we have simulated a wide range of SMC configurations and benchmarks, varying:

- FIFO depth,
- dynamic order/issue policy,
- number of memory banks,
- DRAM speed,
- benchmark algorithm, and
- vector length, stride, and alignment with respect to memory banks.

Only representative examples of our results are given here; complete results (over 7000 simulations for varying benchmarks and memory systems) can be found in [22]. The simulations here use 10,000-element vectors aligned to have no DRAM pages in common, and starting in the same bank.

Arithmetic and control are assumed never to be a computational bottleneck, thus we model the processor as a generator of load and store requests only. This places the maximum stress on the memory system by assuming a computation rate that out-paces the memory's ability to transfer data. Scalar and instruction accesses are assumed to hit in the cache for the same reason.

All memories modeled here consist of interleaved banks of page-mode DRAMs, where each page is 2K double words. The order/issue policy is exceedingly simple. The SMC looks at each FIFO in round-robin order, issuing accesses for the same FIFO stream while:

1) not all elements of the stream have been accessed

2) another write operand is present in the FIFO, or there is room in the FIFO for another read operand.

Results reported here are for the four kernels described in Figure 4. *Daxpy* and *swap* are from the BLAS (Basic Linear Algebra Subroutines) [6][18], *tridiag* is the fifth Livermore Loop [21], and *vaxpy* is a vector *axpy*[1] computation that occurs in matrix-vector multiplication by diagonals. These benchmarks were selected because they are representative of the access patterns found in real scientific codes, including the inner-loops of blocked algorithms. Our results indicate that the actual reference sequence has little effect on SMC performance.

daxpy: $\forall i$    $y_i \leftarrow ax_i + y_i$

tridiag: $\forall i$    $x_i \leftarrow z_i \times (y_i - x_{i-1})$

swap: $\forall i$    $tmp \leftarrow y_i$    $y_i \leftarrow x_i$    $tmp \leftarrow x_i$

vaxpy: $\forall i$    $y_i \leftarrow a_i x_i + y_i$

**Figure 4   Benchmark Algorithms**

As in the previous example, the DRAM page-miss cycle time for these simulations is four times that of a DRAM page hit, and the non-SMC results are for the "natural" reference sequence for each benchmark using non-caching loads and stores. SMC initialization requires two writes to memory-mapped registers for each stream. Since this small overhead does not significantly affect our results, it is not included here.

Figure 5 shows SMC performance for stride-one vectors as a function of FIFO depth and available concurrency compared to the analogous non-SMC systems. The similarity in the shape of the performance curves for all benchmarks illustrates the SMC's relative insensitivity to access patterns in its ability to optimize bandwidth. In all cases, asymptotic behavior approaches 100% of the peak bandwidth that the memory system can deliver.

1. Here "axpy" refers to a computation involving some entity *a* times a vector *x* plus a vector *y*: for *daxpy*, *a* is a double; for *vaxpy*, *a* is a vector.
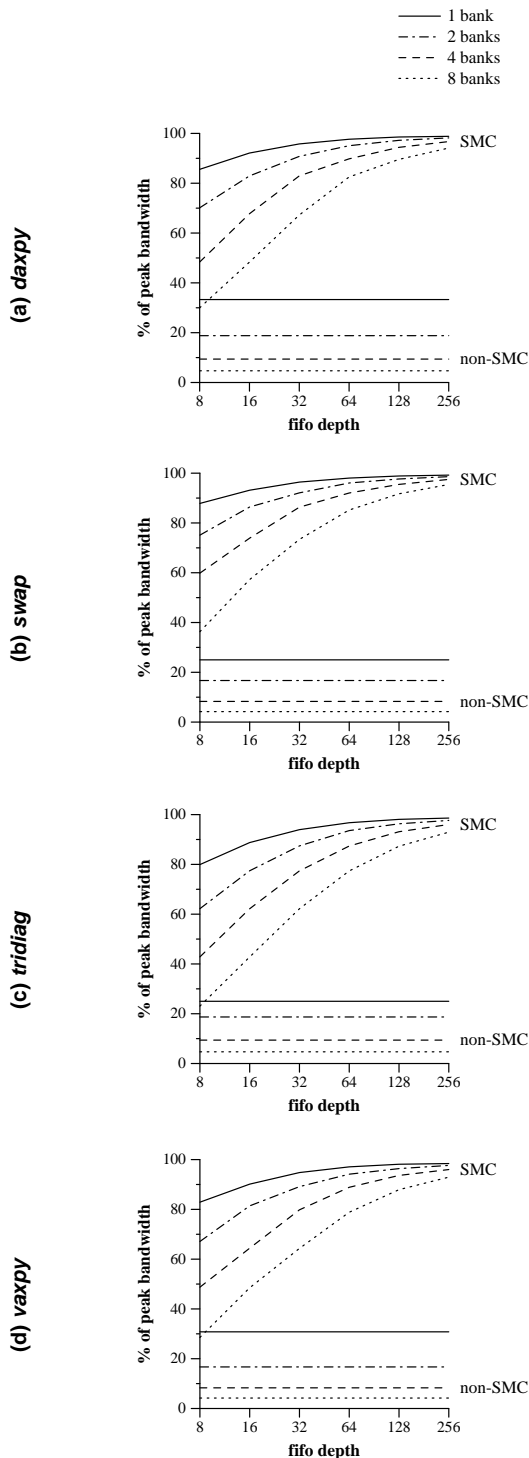
**Figure 5  Long Vector Performance**

performance gain of *over* 420%. On the *tridiag* kernel discussed earlier, a two-bank SMC system achieves 97.7% of the attainable bandwidth, yielding a similar performance gain over the non-SMC bandwidth of 18.7%. Note that these figures are not meant to represent the increase in an application's *total* performance, but rather the increase in performance for inner loops — kernels that are significant parts of scientific computations and are often used to model the behavior of such applications.

In general, SMC systems with deep FIFOs achieve in excess of 92% of peak bandwidth for all benchmarks and memory configurations. Even with FIFOs that are only sixteen double-words deep, the SMC systems consistently deliver over 75% of peak bandwidth.

Increasing the number of banks reduces *relative* performance, a somewhat counter-intuitive and deceptive effect. This is due in part to the fact that increasing the number of banks decreases the number of accesses per FIFO to each bank, thus page-miss costs are amortized over fewer accesses: to maintain performance, FIFO depth must scale with interleaving. The main reason for this effect, though, is that we have kept both the peak memory system bandwidth and the DRAM page-miss/hit delay ratio constant. Thus, the eight-bank system has four times the DRAM page-miss latency of the two-bank system. If instead we hold the speed of the memory banks constant and assume a faster bus, the peak bandwidth of the total system increases proportionally to the number of banks. The percentage of bandwidth delivered for systems with more banks is still smaller in this case, but the total bandwidth is much larger, as in Figure 6.
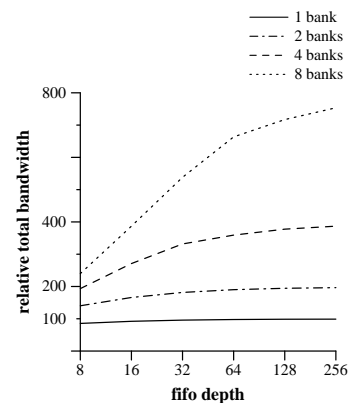


**Figure 6  *daxpy* Performance with Scaled Total Bandwidth**

As with our static access ordering example, the effects of dynamic reordering are dramatic. On the *daxpy* benchmark, for example, an SMC system with two memory banks achieves 98.2% of peak bandwidth, compared to 18.8% for a non-SMC system; this is a

The results presented in Figure 5 and Figure 6 are for unit-stride vectors, but the SMC is very robust in its ability to optimize memory bandwidth regardless of stride. Figure 7 depicts the percentages of attainable bandwidth achieved for *daxpy* using vectors with small strides on an

SMC system with 256-deep FIFOs and four banks of memory. Attainable bandwidth for unit-stride vectors is 100% of the system's potential bandwidth, whereas for stride-two vectors the attainable bandwidth is only 50% of peak, etc.
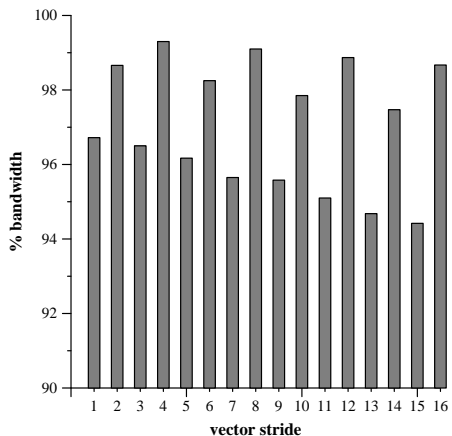


**Figure 7  *daxpy* Performance for Small Strides**

The SMC delivers over 95% of attainable bandwidth in all cases, regardless of vector stride. For even strides we see the same phenomenon described earlier: vectors using fewer banks enjoy a greater percentage of attainable bandwidth. To see why, observe that for a unit-stride vector, each of four banks is responsible for servicing one-quarter of the FIFO. In contrast, for a vector of stride two, each of two banks is responsible for servicing half the FIFO. For a read vector in this case, each bank can amortize page-miss costs over a greater number of accesses before the FIFO becomes full.
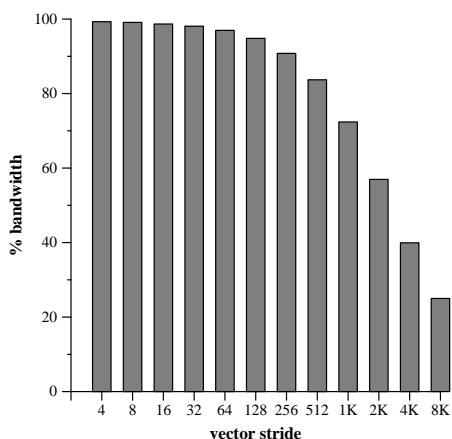


**Figure 8  *daxpy* Performance for Increasing Strides**

Figure 8 illustrates SMC performance for increasing strides. The SMC is able to deliver over 90% of available bandwidth for strides up to 256, after which point performance declines gradually as the stride increases to half the DRAM page size. The SMC performs as well as possible, but as vector stride increases, fewer elements reside within a given DRAM page. This decreases the number of accesses over which the SMC can amortize page-miss costs, and hence limits bandwidth.

As noted, these results are for vectors of 10,000 doubleword elements. SMC performance for shorter vectors is not as dramatic, since there are fewer stream elements over which to amortize the cost of the DRAM page misses. Nonetheless, shorter vector computations still benefit significantly from an SMC [22].

On the basis of our numerous simulations, we deem the SMC concept worthy of further exploration. Simulation is a useful tool, but the real test of any idea lies in the implementation. Thus we are designing and building an experimental SMC board.

## 5. Experimental hardware implementation

Our most important design goal is to provide a proof of concept: dynamic access ordering is an interesting and feasible solution to the memory bandwidth problem for stream computations. We must also demonstrate that an SMC system causes no additional delay in responding to normal memory access requests, either scalar accesses or cache line accesses: applications not using the SMC will incur no performance penalties. Flexibility is another goal, for this implementation will serve as a testbed for experimenting with different access-ordering algorithms, and will allow us to explore the memory bandwidth efficiency as a function of FIFO depth and the number of FIFOs available. Finally, we have chosen a top-down, VHDL-based design methodology to enable element reuse in later implementation phases.

In order to rapidly validate the SMC concept, we have chosen to add an SMC system to an existing microprocessor system. The Intel i860 was selected for its support of vector operations and non-cacheable floating point load and store instructions [13], which will be used to access stream operands. Using this approach has the disadvantage that the stream buffers will be external to the processor, and will therefore incur a higher access cost than the internal cache. However, accesses to the stream buffers should be fast enough that using the SMC will result in a significant performance increase.

The overall architecture of the experimental SMC system is shown in Figure 9. An i860-based board containing a 40 MHz microprocessor and a 2-way cache optimized interleaved 16 MB memory system provide the starting point for the design.

The SMC test board, which is connected to the processor board via an expansion connector, consists of high speed buffer memory (the SBU), the SMC control logic, several data path elements, and two interleaved
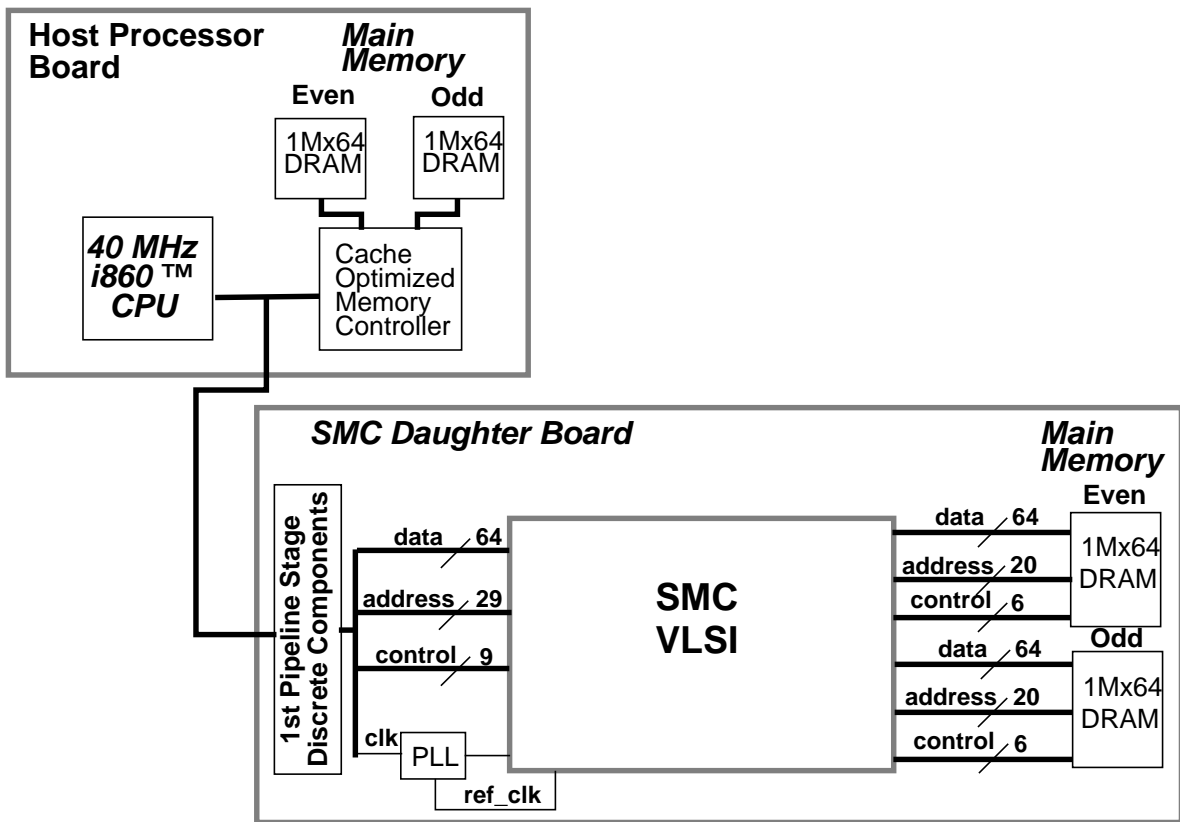
**Figure 9  Architecture of the SMC Test Board**

banks of DRAM main memory. Accesses to the SMC test board by the i860 are pipelined. This is necessary because of the high latency of address and cycle definition lines: the processor has a maximum latency of 11 ns for the address and cycle definition lines, and further delay is encountered as the signals travel to the expansion connector from the i860, making the signals are available at the edge of the SMC test board only 10-12ns before the (40 MHz) clock edge. A pipeline stage is used to latch these signals, thereby increasing the available time to access the SMC board within the next cycle. The onboard cache-optimized memory system is similarly pipelined.

### 5.1  Stream buffer memory

The high-speed memory of the SBU will be implemented logically as a set of FIFOs. The order in which the buffer is filled is determined by the memory controller. In the case of stream read accesses, the FIFOs are filled from the DRAM and drained by the i860; for stream writes, the FIFOs are filled by the i860 and drained to the DRAM. From the memory system's point of view, each stream FIFO will be implemented as a set of smaller FIFOs (subFIFOs), one per memory bank. The control logic must therefore fill (or drain) the stream elements from

a particular memory bank in stream order. This is not a significant restriction, however, since there will very rarely be any performance benefit from servicing these elements out of order. On the other hand, the subFIFO organization significantly reduces SMC complexity, simplifying both the FIFO status logic and the logic to determine the next stream request to the DRAM.

In order to provide the flexibility necessary to explore performance ramifications of different FIFO configurations, we have adopted a virtual FIFO scheme using an internal dual-ported SRAM (DP-SRAM) for storage [20]. The depth and number of FIFOs is thus limited only by the size of the implemented DP-SRAM.

If we are to provide 100% bus bandwidth between processor and FIFOs for pipelined, double-precision floating point loads and stores, the SMC must be able to provide a double word every 25 ns, for the processor can supply a new quadword address every 50 ns. Since the DP-SRAM (implemented in 1.2μm CMOS technology) we intend to use for the SMC has an access time on the order of 12 ns, we can use two banks of interleaved DP-SRAM to meet the bandwidth requirement. In order to service continuous, double-precision floating point accesses with no wait states, the SMC must also be able to accept a new
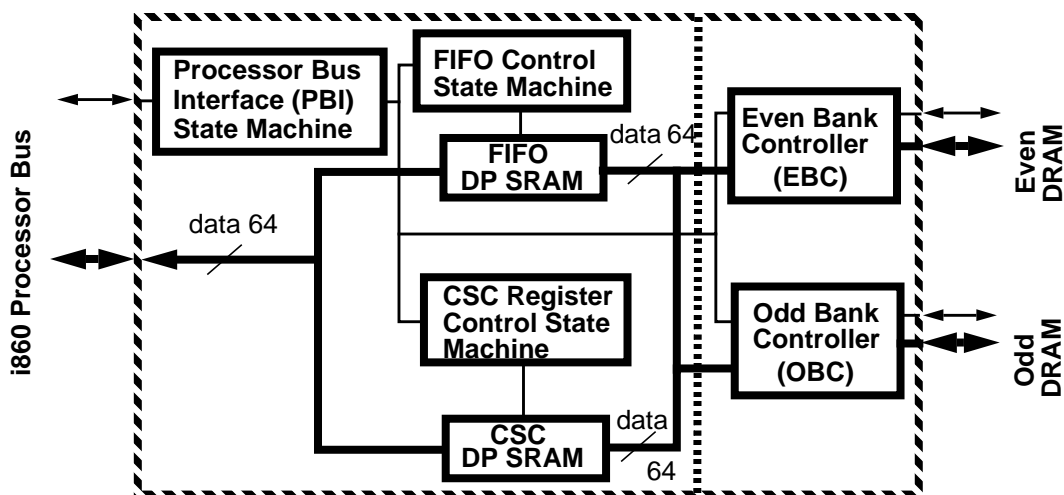
**Figure 10   SMC VLSI Implementation**

address every 50 ns. This address is presented to both banks of DP-SRAM, and two double words are accessed. For reads, the first double word is sent directly to the processor; the second is latched within the SMC so that it can be sent on the next bus cycle. For writes, the first double word from the processor is latched within the SMC until the next cycle, when the second double word arrives. Both double words are then written into the DP-SRAM together.

### 5.2  SMC control logic

The proposed SMC VLSI implementation, shown in Figure 10, consists of several state machines and Control/ Status (CSC) registers. In addition to storing the stream parameters (base, length, and stride), the CSC registers govern the read/write modes of the individual FIFOs and provide a user-accessible reset control for the entire SMC.

The Processor Bus Interface (PBI) state machine is responsible for handling all handshaking between the SMC and the i860 for all requests from the SMC board memory, including stream, scalar, and cache line accesses.

The FIFO state machine maintains pointers to the virtual FIFOs contained in the SRAM, as well as status signals on the condition of each (full, empty, half full, etc.). The state machine allows simultaneous access to the FIFO DP-SRAM, so that the SMC bank controllers and the processor can access the FIFOs concurrently. This capability is necessary for the bank controllers to keep pace with the processor's stream requests.

The SMC will have low-skew clock distribution trees built into its architecture, but the fixed delay in the clock as it is driven onto the SMC might be as great as 6 to 8 ns, which is unacceptable for a high-speed (40 MHz) design. The SMC therefore uses a phased locked loop (PLL) to synchronize its on-chip clock with the system clock. In this design, the reference signal for the PLL is connected to the

clock driven off of the SMC from its distributed clock tree, and the locked signal is fed back to the input of the SMC clock network. Clock synchronization within 1 ns should be possible using this approach.

### 6. Test plans

Once we have verified the SMC board functionality, we can begin relating its performance to our software simulations. The onboard memory, which is optimized for cache line access (loads and stores of four 64-bit double words), will provide a basis of comparison. Initially, vector algorithm test cases will be run out of the onboard memory to obtain base-line timing information. These real-time results can then be compared with those of the same algorithms run out of the SMC-controlled memory. A few practical considerations must be factored into our comparisons, for the cache-optimized onboard memory provides functionality (such as parity, error correction, and cache snooping capabilities) that the SMC-controlled memory will not, and these may affect the overall timing results. Nonetheless, we expect to see significant performance differences, the bulk of which can be attributed to the SMC.

### 7. Conclusions

As the disparity between microprocessor speeds and memory speeds increases, memory bandwidth is rapidly becoming a performance bottleneck, especially in the application of high performance microprocessors to vector-like algorithms. These computations, which include many of the "Grand Challenge" problems, lack the temporal and spatial locality necessary for caching alone to bridge the performance gap.

We can get better bandwidth from existing technology if we take advantage of the characteristics of the entire memory hierarchy, including random-access memory components. Performance of modern DRAM components is sensitive to the order of requests, thus these devices should not be treated as uniform access-time RAM. Moreover, exploiting the special capabilities provided by such devices is best done dynamically, since essential information (such as alignment) will generally not be available at compile time.

Here we have described one technique, access ordering, that can optimize requests to exploit the underlying memory architecture. We then proposed an interesting alternative to existing solutions to the bandwidth problem: a combined hardware/software scheme to implement dynamic access ordering. This scheme complements more traditional cache-based schemes, so that overall effective memory performance need not be a bottleneck.

Preliminary explorations indicate that this solution is both feasible and cost-effective: we have the necessary compiler technology; our simulation results show that by combining compile-time detection of streams with execution-time selection of the access order and issue, we can achieve near-optimal bandwidth for vector-like accesses relatively inexpensively; and we have described the hardware design of an experimental Stream Memory Controller (SMC) system that will allow us to explore the dynamic access ordering concept further.

## 8. Acknowledgments

## References

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

[2] Baron, R.L., and Higbie, L., *Computer Architecture*, Addison-Wesley, 1992.

[3] Carr, S., Kennedy, K., "Blocking Linear Algebra Codes for Memory Hierarchies", Proc. Fourth SIAM Conference on Parallel Processing for Scientific Computing, 1989.

[4] Callahan, D., et. al., "Software Prefetching", Fourth International Conference on Architectural Support for Programming Languages and Systems, April, 1991.

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

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

[7] Fu, J. W. C., and Patel, J. H., "Data Prefetching in Multiprocessor Vector Cache Memories", 18th International Symposium on Computer Architecture, May, 1991.

[8] Goodman, J. R., et al, "PIPE: A VLSI Decoupled Architecture", Twelfth International Symposium on Computer Architecture, June, 1985.

[9] Gupta, R., and Soffa, M., "Compile-time Techniques for Efficient Utilization of Parallel Memories", SIGPLAN Not., 23, 9, 1986.

[10] Harper, D. T., "Address Transformation to Increase Memory Performance", 1989 International Conference on Supercomputing.

[11] Hayes, J.P., *Computer Architecture and Organization*, McGraw-Hill, 1988.

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

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

[14] Jouppi, N., "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers", 17th International Symposium on Computer Architecture, May, 1990.

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

[16] Klaiber, A., et. al., "An Architecture for Software-Controlled Data Prefetching", 18th International Symposium on Computer Architecture, May, 1991.

[17] Lam, Monica, et. al., "The Cache Performance and Optimizations of Blocked Algorithms", Fourth International Conference on Architectural Support for Programming Languages and Systems, April, 1991.

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

[19] Maccabe, A.B., *Computer Systems: Architecture, Organization, and Programming*, Richard D. Irwin, Inc., 1993.

[20] McCarty, D., "Tackling FIFO Design with FPGAs", ASIC & EDA, September, 1992.

[21] McMahon, F.H., "The Livermore Fortran Kernels: A

Computer Test of the Numerical Performance Range", Lawrence Livermore National Laboratory, UCRL-53745, December, 1986.

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

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

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

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

[26] Rau, B. R., "Pseudo-Randomly Interleaved Memory", 18th International Symposium on Computer Architecture, May, 1991.

[27] Sklenar, Ivan, "Prefetch Unit for Vector Operation on Scalar Computers", Computer Architecture News, 20, 4, September, 1992.

[28] Smith, J. E., et al, "The ZS-1 Central Processor", The Second International Conference on Architectural Support for Programming Languages and Systems, October, 1987.

[29] Sohi, G. and Manoj, F., "High Bandwidth Memory Systems for Superscalar Processors", Fourth International Conference on Architectural Support for Programming Languages and Systems, April, 1991.

[30] Tomek, I., *The Foundations of Computer Architecture and Organization*, Computer Science Press, 1990.

[31] Valero, M., et. al., "Increasing the Number of Strides for Conflict-Free Vector Access", 19th International Symposium on Computer Architecture, May, 1992.

[32] Wolfe, M., "Optimizing Supercompilers for Supercomputers", MIT Press, Cambridge, MA, 1989.

[33] Wulf, W. A., "Evaluation of the WM Architecture", 19th Annual International Symposium on Computer Architecture, May, 1992.