Compiling for Efficient Memory Utilization

Sally A. McKee
University of Virginia
http://www.cs.virginia.edu/~sam3a
mckee@virginia.edu

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 150 MHz 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 [IEE92].

The sense amplifiers in fast-page mode devices behave much like a single line, or *page*, of cache on chip. A memory access falling outside the current page causes a new one to be loaded, making such *page misses* take three to five times as long as *page hits*. On a Cray T3D system with 16Mbit DRAMs, each page is 2048 Kbytes, page hits take about 4 cycles (26 ns), and the additional overhead for page misses takes about 15 cycles more (for about 125 ns in all) [Pal95,Bro95]. Although the terminology is similar, DRAM pages should not be confused with virtual memory pages.

With an advertised bandwidth of 500 Mbytes per second, Rambus is another interesting new memory technology [IEE92]. These bus-based systems are capable of delivering a byte of data every 2 ns 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. Unfortunately, these devices offer no performance benefit for random access patterns: the latency for accesses that miss the cache lines is 150-200 ns.

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) [IEE92]. 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 can be completed 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.

Access ordering, or changing the order of memory requests to improve effective memory bandwidth, is one technique that can help bridge the processor-memory performance gap for streaming computations. This reordering can be done statically, at compile time, or dynamically, by hardware at run time. Here we survey compile-time approaches to access ordering for three platforms: a node of the iPSC/860, the University of Virginia’s i860-

based Stream Memory Controller (SMC) System, and a node of the Cray T3D. The latter two systems include hardware to dynamically order stream memory references, but getting the best performance from these systems requires some compile-time optimization.

The approaches described here are not new, and some of the results included here have first been presented elsewhere. Since the initial sources of some of these results are technical reports or working papers that have not appeared in conferences or journals, they may not be known to the general computer architecture and compiler communities. Part of the motivation for this paper is thus to try to call attention to this work.

2. Access Ordering

Reordering accesses may have no effect on the performance of some systems, but others provide features that enable the optimization. For instance, an instruction set that gives the user flexibility in how memory is accessed makes it easier to exploit features of the memory system through access ordering. Although not a common feature, non-caching loads are supported by some processors (as in the Convex C-1 [Wal85] or Intel i860 [Int91]). Others allow some regions of memory to be designated as non-cacheable (as in the DEC Alpha [DEC92]). Still other systems include special-purpose hardware for streams (as in the Cray T3D [Cra95], the WM [Wul92], the Stream Memory Controller [McK95c], or the hardware proposed by Valero, et al., for reordering vector accesses to avoid bank conflicts [Val92]).

On machines that enable access ordering optimizations, the performance benefits can be significant. As a simple example, consider the effect of executing the fifth Livermore Loop (*tridiagonal elimination* [McM86]) 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 $x_i \leftarrow z_i \times (y_i - x_{i-1})$.

This computation occurs frequently in practice, especially in the solution of partial differential equations by finite difference or finite element methods. Since it contains a first-order linear recurrence, it cannot be vectorized. Nonetheless, the compiler can employ the recurrence detection and optimization algorithm described in Section 4.1 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. Except in the case of very short 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 reference. Instructions likely to generate page misses are shown in boldface in the figure.

(a)

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

(b)

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

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. With a device for which page misses take four times as long as page hits, 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%.

The left graph of Figure 2 illustrates effective memory bandwidth versus depth of unrolling. The bottom line represents performance when the loop of Figure 1(a) is unrolled by concatenating copies of the body. The middle curve shows performance for an unrolled loop in which accesses have been arranged as per Figure 1(b). Finally, the top line illustrates 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. Although in theory we could improve performance almost 240% by unrolling to a depth of sixteen, in most cases the size of the register file won't permit that degree of optimization. The graph on the right presents the same information in terms of the average number of cycles per stream access.

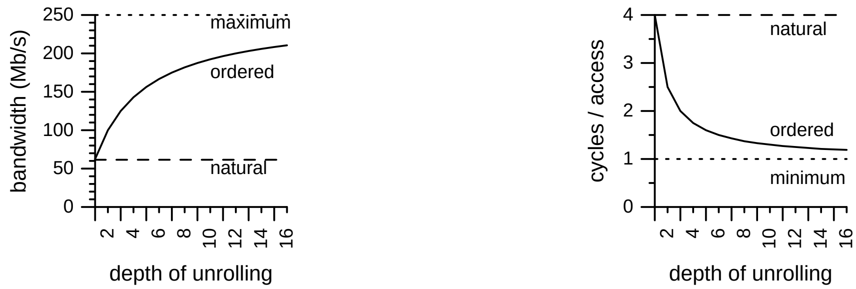

Figure 2 *tridiag* Memory Performance

3. Experimental Platforms

Our investigations of compile-time access ordering target three different platforms: a node of the Intel iPSC/860 [Int91], the University of Virginia's Stream Memory Controller system [McK95c], and a node of the Cray T3D [Cra95]. Some preliminary background on the relevant features of each memory system is helpful in understanding how changing the order of memory references affects performance.

3.1 Intel iPSC/860

The i860XR used for each node of the iPSC/860 implements two versions of the load instruction for floating-point values: floating-point load (*fld*) and pipelined floating-point load (*pfld*). The *fld* instruction is a traditional memory-to-register load, bringing the line of data containing the memory location into cache. The *pfld* instruction uses a three-state load pipe to bypass the cache (if the reference hits in the data cache, the cached data is used, but if the request misses in the cache, the cache is not updated). Each "*pfld src,dst*" instruction explicitly advances the pipe by one stage: the data from the specified *src* memory location is input into the first stage of the pipe, and the value in the third stage is placed into the *dst* result register. This instruction can be used for non floating-point data, as well.

The 40 MHz iPSC/860 node used for our experiments has the following parameters:

- cache lines are 32 bytes, or 4 double words,
- pipelined loads fetch one 8-byte double word, and DRAM page misses and page hits take 2 and 10 cycles, respectively,
- caching loads and stores that hit the cache can transfer a 16-byte quadword in one cycle,
- the write-back cache holds 8K bytes, and is two-way set associative with pseudo-random replacement, and
- DRAM pages are 4K bytes.

Ordering accesses using the *pfld* instruction (as in the bandwidth example of Section 2) can exploit the DRAM's fast page-mode, improving effective bandwidth for a computation. Unfortunately, ordering accesses using the *fld* instruction has no effect on bandwidth. Each successive cache-line fill incurs a seven cycle delay, which is long enough for the memory controller to transition to its idle state. The next memory access takes the same time as a DRAM page-miss, regardless of whether or not it lies in the same page as the previous access. Thus each cache-line fill takes 16 cycles (10 + 2 + 2 + 2).

The i860 also provides two 128-bit write buffers designed to delay store operations to memory until the memory bus is idle, allowing the processor to continue without stalling on a write while reads are in progress. Data is forced back to memory when the buffers are full and another store instruction is executed, or when the write-back is necessary to maintain data coherence.

3.2 Virginia SMC

The i860's non-caching load instructions make it a suitable host processor for the dynamic access ordering system built at the University of Virginia. In this system, the compiler recognizes streams and arranges to transmit information about them to the hardware at run-time. The access ordering hardware then prefetches the read operands, buffers the write operands, and reorders the accesses to improve memory system performance. Figure 3 shows the high-level organization of this *Stream Memory Controller* (SMC) system.

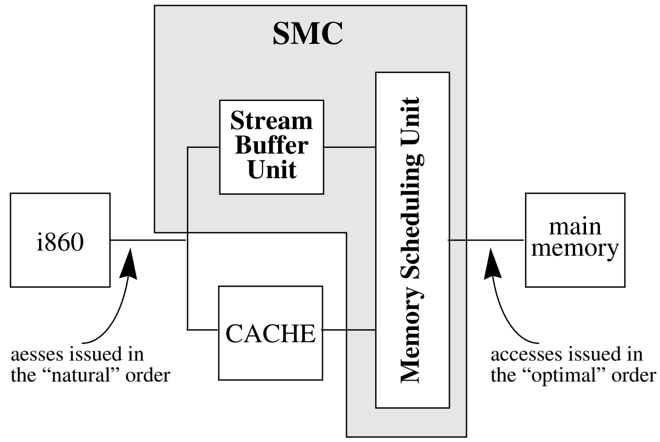

Figure 3 SMC Organization

The system 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, such as interleaving and device characteristics. The access-ordering circuitry uses this information to issue requests for individual stream elements in an order that attempts to maximize memory system performance. The design of the MSU is thus specific to a particular memory architecture; changing the memory system probably requires changing MSUs.

The Stream Buffer Unit contains high-speed buffers for stream operands, and it 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. The MSU accesses these buffers as if they were random-access register files, asynchronously filling them from or draining them to memory. The processor references the next element of a stream via the memory-mapped register representing the corresponding FIFO head. By memory mapping the control registers and FIFO heads, we avoid having to modify the processor's instruction set. Figure 4 illustrates the SMC programming model for our *tridiag* example.

(a)

```
/* tridiagonal elimination: */

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

(b)

```
/* SMC version: */

streamin(y+1, size, stride, n-1, FIFO0);    /* xmit stream params */
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 4 SMC Programming Model

The 40 MHz SMC host processor is an i860XP, which differs slightly from the i860XR described above. The XP's pipelined loads can access quadwords, and the data cache is 16K words organized as four sets of 32-byte lines. The CPU can initiate a bus transaction every other clock, and quadword instructions allow the i860 to read 128 bits of data in two consecutive cycles. The SMC can thus deliver a 64-bit double word of data every cycle. Further details of system implementation and performance can be found in our other reports (see http://www.cs.virginia.edu/research/techrep.html, http://www.cs.virginia.edu/~wm/smc.html).

3.3 Cray T3D

Each node of the Cray T3D system is a 150 MHz DEC Alpha EV4 processor with a peak performance of 150 Mflops. The peak read bandwidth from main memory is only 320 Mbytes per second, or one 64-bit word every four cycles. Depending on how memory is referenced, this latency can be as long as one word every 42 cycles. The direct-mapped, onchip data cache holds 256 lines of four words each, for a total capacity of 1024 words that can be accessed at a rate of one 64-bit word every cycle [Bro95].

This machine implements a *read-ahead* mode[1] that causes the next cache line after a miss to be prefetched into a special stream buffer. If the next load that generates a cache miss is sequential, it will hit in the read-ahead buffer and cause the line to be transferred to cache. If a cache miss also misses in the read-ahead buffer, the contents of the buffer are discarded, and the correct line is fetched from main memory. Figure 5, adapted from Brooks, illustrates the memory organization of the T3D and the approximate latencies associated with transferring data between each level. If an access hits the cache, the data is available in a register within a single cycle. Accesses that miss the cache but hit in the read-ahead buffer take 13 to 15 cycles; those that miss the read-ahead buffer but hit one of two current DRAM pages take about 22 clocks; and those that miss cache, buffer, and page take 37 to 42 cycles [Bro95,Pal95].

The EV4 chip includes four write buffers, each the length of one cache line. As soon as one of these buffers contains a "valid entry", a buffer flush sequence is initiated to write the data back to DRAM. This process takes about 12 cycles to complete, but accesses that miss the current DRAM page will increase the observed latency. When all four buffers are in use, subsequent store instructions stall the processor until a buffer becomes available.

1. Linking a program with "`mppldr -D rdahead=on program.o`" enables this mode.

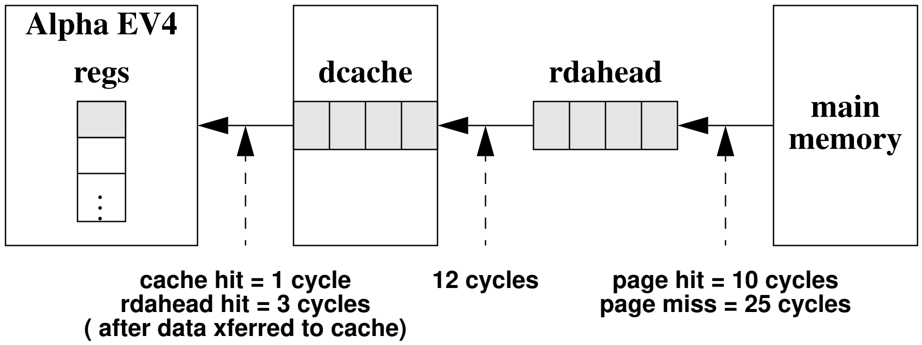

Figure 5 Cray T3D Memory Organization

4. Exploiting Memory System Features in the Compiler

The last section described three systems for which naively generated code can have (unexpectedly) bad performance due to inefficient use of memory system features. This section surveys access ordering schemes that the compiler can use to generate more memory-efficient code.

Compile-time access ordering can be loosely classified into three tasks: stream detection, access reordering, and determining the extent to which the reordering should be done. There may be options with respect to the choice of reordering strategy, depending on the particular level of the memory hierarchy targeted by the optimization. For instance, if the data is to be stored in cache, there may be benefits to loading a stream by copying it (or portions of it) to a different area of main memory in order to avoid conflicts with other data that needs to remain cache-resident. How best to do the reordering may also depend on things like register pressure or the costs of applying the optimization. Obviously, if data is copied, the costs of moving the data an extra time must be weighed against the potential benefits in terms of increased memory system efficiency.

We do not address issues such as aliasing or memory coherence in this paper, other than to note that if the compiler lacks information to decide something at compile time, it may be able to generate code to make the decision dynamically at run time. For instance, the SMC 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.

4.1 Stream Detection and Optimization

The compiler technology to detect streams is a necessary part of our approach to access ordering, thus we include a description of the algorithms. The ideas behind them are initially due to Wulf. Benitez and Davidson present versions of the algorithms developed for the WM, a novel superscalar architecture with hardware support for streaming [Wul92,Ben91]. Although designed for architectures with hardware support for the access/execute model of computation [Smi84], many of the techniques are applicable to stock microprocessors.

Vectorizing compilers could be used to identify potential accesses for reordering, but stream detection is both simpler and more general than vectorization. Streaming code can often be generated for codes that are impossible to vectorize. For instance, streaming naturally handles codes containing recurrence relations, computations in which each element of a sequence is defined in terms of the preceding elements.

We first present an algorithm to handle such recurrences, then we give the algorithm to generate streaming code for the optimized loops. Both algorithms require that the loop's memory accesses be divided into groups, or *partitions*, that reference disjoint sections of memory. For example, each local or global user-declared variable, whether scalar or array, defines a partition. Each partition can be uniquely identified by a local, global, or label identifier; this is the *partition handle*.

We do not transcribe the algorithms verbatim; any errors introduced in adapting them are solely the responsibility of this author.

4.1.1 Recurrence Detection and Optimization Algorithm

The algorithm given in Figure 6 breaks recurrences by retaining write-values until needed by a later iteration. The retained values are "pipelined" through a set of registers, advancing one register on each iteration until consumed by a read instruction, as illustrated in Figure 7. For the *tridiag* example from Section 2, only the *x* value from the previous iteration is needed, and so a single register suffices to hold the retained values for this particular recurrence (sample code for this is shown in Figure 4).

1) Divide the loop's memory accesses into *partitions* that reference disjoint sections of memory. If the proper partition is unknown for a particular reference, add that memory reference to all partitions. Record where each reference occurs and whether it is a read or a write.
2) Determine the induction variables in the loop, and for each induction variable *j*, determine its associated *c* and *d* values, and whether *j* is increasing or decreasing. (Each induction variable *j* can be represented by a basic induction variable *i* and two constants, *c* and *d*, such that at the point where *j* is defined, its value is given by *c*i+d*. In other words, *c* denotes a scale factor, and *d* denotes an offset.)
3) For each partition, do:
   a) If not all references in the partition have the same induction variable or the same *c* value, mark the partition as *unsafe*.
   b) Algebraically simplify each *d* value in the partition by removing the partition handle and any invariant register values. If any *d* value cannot be simplified into a literal constant, mark the partition *unsafe*. The resulting literal constant is the *relative offset* between the reference and the induction variable. If the relative offset is not evenly divisible by the *c* value, mark the partition *unsafe*.
4) For all *safe* partitions containing both reads and writes (no other partitions can contain recurrences), do:
   a) Identify pairs of memory references in which a read fetches the value written on a previous iteration, and for each such pair, calculate the iteration distance between the references. This is the absolute difference of the relative offsets for the references; the maximum distance divided by the stride of the loop determines the number of registers needed to handle the recurrence. We refer to these memory references as *read/write pairs* and to the number of registers required as the *degree* of the recurrence.
   b) For each read/write pair, generate code before each write to copy the value to a register, and replace the corresponding reads with register references. Update the partition to reflect that the read is no longer performed in the loop.
   c) Generate code at the top of the loop to advance the recurrence values through the register pipeline at the start of each new loop iteration.
   d) Build a loop *pre-header* to perform the initial reads (i.e., prime the register pipeline).

Figure 6 Recurrence Detection and Optimization

This algorithm relies on induction variable detection, and is similar in complexity to strength reduction. Any good compiler textbook contains descriptions of these operations (e.g., Aho, Sethi, and Ullman [Aho88]).

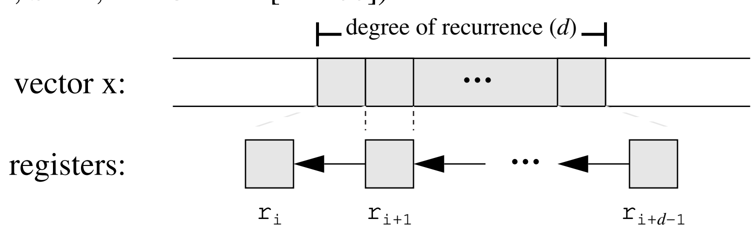

Figure 7 Pipelining Recurrence Values through Registers

4.1.2 Streaming Optimization Algorithm

For hardware that supports streaming, the compiler must exploit opportunities for streaming operations after recurrences have been detected. The following algorithm uses the memory partition information collected by the recurrence optimization algorithm. Step 4 above excludes read-only and write-only streams, whereas the following algorithm applies to all streams in safe partitions. We present the algorithm in terms of the SMC architecture described in Section 3.2, with the architecture-specific steps in italics.

1) If any memory recurrences remain in the loop, do not stream.
2) Determine the number of iterations through the loop. If the count is unknown, set it to ∞ . If the count is too small, do not generate streaming code. *(The cutoff at which streaming is no longer profitable depends on both the architecture and the computation.)*
3) For each memory reference in all safe partitions, if the memory reference is executed each time through the loop, do:
    a) Calculate the stride.
    b) Determine the number of times the memory reference is executed (i.e., if it should not be executed on the final loop iteration, generate code appropriately).
    c) *If using the SMC, allocate the appropriate FIFO.*
    d) Generate code in the loop pre-header to test whether the loop should be executed and to jump around the loop if necessary.
    e) Generate the stream-initiation code before the loop. This code transfers stream parameters (base address, stride, stream length) to the streaming hardware. *For the SMC, this entails generating instructions to write the memory-mapped control registers in the Stream Buffer Unit. For the WM, it requires emitting the appropriate Sin or Sout instruction.*
    f) Change loads and stores to reference the appropriate registers *(i.e., FIFO heads in the SMC)*.
    g) If the loop count is ∞ , add instructions to stop streaming at all loop exits.
    h) If the induction variable is dead on loop exit, delete the increment of the induction variable.
4) Perform strength reduction on the optimized loop.

Figure 8 Streaming Optimization

4.2 Access Reordering

Unrolling and grouping accesses to each stream is the crux of the compile-time access ordering techniques described here. This reordering has several potential benefits:

- it increases the locality in the reference stream, which improves performance for fast-page mode or cached DRAMs, or makes it possible to transfer larger blocks of data at a time (exploiting wide-word accesses or burst mode devices);
- it minimizes the number of transitions between reading and writing, which may avoid delays associated with switching the direction of the memory bus; and

- for architectures implementing write buffers, it maximizes the amount of useful data that is transferred when a buffer is flushed.

This optimization can be applied to instructions that transfer data from main memory to registers, to cache, or to special hardware buffers (or vice versa). For instance, Moyer introduces static access ordering to maximize effective bandwidth for non-caching register loads. He derives his algorithms relative to a precise analytic model of memory systems [Moy92a,Moy92b,Moy92c,Moy93]. Since the best ordering cannot be determined without address alignment information that is not generally available at compile time, these formulas calculate an ordering that is probabilistically optimal. They do this by minimizing the word access overhead (for wide-word accesses)[1] and the page-miss overhead. Techniques to minimize the page-miss overhead include *intermixing* read and write accesses to the same vector, and arranging the total order of access sets (i.e., accesses grouped by stream) so that the last vector accessed in one iteration of a loop is the first one accessed in the next iteration.

Lee combines register-level access ordering and data copying in his NASPACK subroutine library, which mimics Cray vector instructions on the Intel i860XR [Lee91,Lee93]. He obtains better memory system performance by ordering accesses and explicitly managing the cache as a set of “vector registers”: in his routines for streaming vector elements into cache, the data is read in blocks via non-caching load instructions, and is written into pre-allocated (i.e., cache-resident) local memory. The computation is then strip-mined to operate on the cached data blocks. Meadows, et al., describe a similar *preloading* scheme used by the PGI i860 compiler [Mea92].

Unrolling and scheduling accesses can be used to bring data into cache as efficiently as possible when the instruction set implements a block prefetching instruction. The technique

1. *Memory access coalescing* is a related compile-time optimization that attempts to minimize the number of bus transactions for wide-bus machines [Ale93].

may be useful for systems that include hardware support for dynamic access ordering, as well. For instance, reordering can be used to exploit the stream buffers proposed by Jouppi [Jou90] or the Cray T3D's read-ahead mechanism discussed in Section 3.3. Even though data is always fetched and cached in units of a line, register-level access ordering can be used to load the cache lines in an order that exploits the read-ahead hardware and the DRAM's page-mode. A scheme similar to Lee's can also be used to preload stream data to cache before the inner loop of a computation. Instead of reading all stream elements, as was done with the non-caching pipelined loads of the i860, it suffices to touch one element per line when loading the data to cache [McK95a]. Palacharla and Kessler examine the performance of both schemes, register-level reordering and preloading, for several vector kernels as well as whole benchmarks [Pal95].

Compile-time access ordering can still be useful for a system in which an external SMC has been added to enhance the performance of an existing processor. This organization is used in our prototype, proof-of-concept system [McK95c]. In this system, performance depends on processor bus utilization as well as memory utilization, thus the cost of switching between reading and writing should be amortized over as many accesses as possible.

Finally, the performance of the code generated by the recurrence algorithm of Section 4.1 can be improved by unrolling the loop to a depth equal to the degree of the recurrence and *renaming* the registers holding the retained values. This eliminates the "register pipeline" and the need to copy the recurrence values at the top of the loop.

4.3 Evaluating Cost/Performance Tradeoffs

Once an access ordering strategy has been selected, the compiler must decide to what extend the optimization should be applied (or it must generate code to make this decision at run-time). Here we describe the kinds of costs that must be taken into consideration. The following section formalizes these costs for the architectures we investigate.

For the register-level static ordering schemes that Moyer considers, the best performance can usually be achieved by unrolling as much as possible. This limit depends on register pressure, or the size of the processor's register file versus the number of registers required for the unrolled computation. For architectures with exposed pipelines, such as the i860, a clever compiler can exploit these pipelines for temporary storage.

The cost of using the cache as a "pseudo vector register" includes the time to allocate the portion of cache to be used as local memory (simply by reading the array), the time to copy the strip-mined blocks of stream data to this local memory, the time to retrieve the data from cache during the computation, and the time to service any additional cache misses caused by using the cache this way. Since non-caching loads are used to bring the stream data in from main memory, the only cached data that is displaced is that which maps to the same cache lines as the array used as the local memory buffer.

Loading the local memory buffer into cache is a one-time cost, assuming there are no conflicts with other cached data required by the computation. The other costs must be paid for each block of strip-mined data. If the preloading is done via a library subroutine call that is not inlined, then the overhead cost of the call must be taken into consideration as well. Note that aggressive unrolling may decrease performance if the code for the loop exceeds the capacity of the instruction cache, or otherwise increases the number of cache conflicts. The best size for the data blocks depends on the computation and memory system parameters: the scheme may only represent a performance improvement beyond a certain threshold of blocking.

The cost of using the Stream Memory Controller includes the time to transmit stream parameters to the hardware as well as the time the CPU waits for the SMC to supply the first operand from each read-stream. In its effort to exploit the DRAM's fast-page mode, the SMC prefetches stream elements to fill a whole FIFO before it moves on to access another stream (and incur a page miss). These initial costs are amortized over all accesses in the computation. Good performance on short streams therefore requires that FIFO depth be run-time selectable, in order to limit the relative performance effect of startup costs.

In order to exploit read-ahead mode on the Cray T3D, loops must be unrolled to a depth of twice the number of words in a cache line. Again, the code expansion that results from unrolling may have costs in terms of instruction cache performance. The number of available registers limits the extent to which this ordering can be done, just as it did for the i860. We can avoid this limitation by storing preloaded data in cache, but we have no control over where in cache the stream data is placed. Cache conflicts must be taken into account when applying this technique, otherwise preloaded data may no longer be present at the time it is needed. To investigate the performance effects of such conflicts when more than one stream is preloaded to cache, Palacharla and Kessler experiment with random array placements (175 in all) [Pal95]. They generate code to determine dynamically how much of each stream to preload. The overhead costs in this approach include computing the block size to preload and the extra reads to bring in each cache line. In addition, using the read-ahead mechanism means that an extra cache line will be fetched into the stream buffer and discarded. We incur this overhead for each block of each stream in a computation.

5. Performance Examples

This section describes the experiments that have been (or will be done) on each platform, and presents preliminary results. Since this is work in progress, we have more results for the SMC than for the other systems. In particular, at the time of this writing we lack timing results for the Cray T3D.

We derive models of performance based on architectural parameters and the access patterns of a given computation. In the following discussion, let:

t_{cache} be the cost of accesses that hit in the cache;

$t_{rdahead}$ be the cost of reads that hit in the read-ahead buffer on the T3D,

t_{miss} be the DRAM page-miss cost, in cycles; and

t_{hit} be the DRAM page-hit cost, in cycles.

In addition, let:

b be the unrolling or *blocking* factor of a reordered computation,

m be the number of modules in an interleaved memory system,

f be the FIFO depth of an SMC system,

r be the number of read-only vectors in a computation,

w be the number of write-only vectors in a computation,

u be the number of read-modify-write or *update* vectors,

s be the total number of streams $(s = r + w + 2u)$, and

v be the total number of vectors $(v = s + w + u)$.

Furthermore, we assume that the vector length is always a multiple of the blocking factor, b.

5.1 iPSC/860 Node

For the i860 systems we used, $t_{hit} = 2$ and $t_{miss} = 10$. If peak system bandwidth is that required for the processor to execute a load or store every cycle (what we would get if all accesses hit in the cache), then any computation using the non-caching *pfld* instruction can only exploit half the system's peak bandwidth. Cacheline fills deliver only one quarter of the peak system bandwidth, taking an average of 4 cycles per double word access.

If a computation contains r read-only vectors, w write-only vectors, and u update-vectors (that are both read and written), then the best performance we can get from register-level access ordering will be a per-element access time of:

$$\frac{(r+w)\ (t_{miss} + \ (b-1)\ t_{hit})\ + u\ (t_{miss} + \ (2b-1)\ t_{hit})}{(2u+r+w)\ b}$$

The *tridiag* example from Section 2 reads two vectors and writes one, thus performance for a blocking factor of 8 is an average of 3 cycles per access, at best. This represents 66.7% of the attainable bandwidth, or 33.3% of the peak system bandwidth. For a *daxpy* computation ($y_i \leftarrow y_i + ax_i$, where all operands are 64-bit double words) there is one read vector and one update vector. Here the minimum average number of cycles per stream access for $b = 8$ is 2.67. Figure 9 depicts these limits graphically, along with a few empirical data points. We measured the performance for loading a 1024-element vector with blocking factors of 8 and 16. The *daxpy* performance is calculated from Lee's timings for a 1024-element vector and blocking factor of 10 [Lee92]. In general, performance depends on block size, but is relatively independent of vector length. The limits for *tridiag* apply to any computation without read-modify-write vectors, including a single vector load.

The *pfld* limits are the same for streams with relatively small strides (i.e., strides less than the DRAM page size divided by the blocking factor — otherwise more than one DRAM page miss would be incurred for each block). In contrast, using caching loads for streams with non-unit strides can diminish the effective bandwidth by up to a factor of 4 (the case when only one element of each line is used).

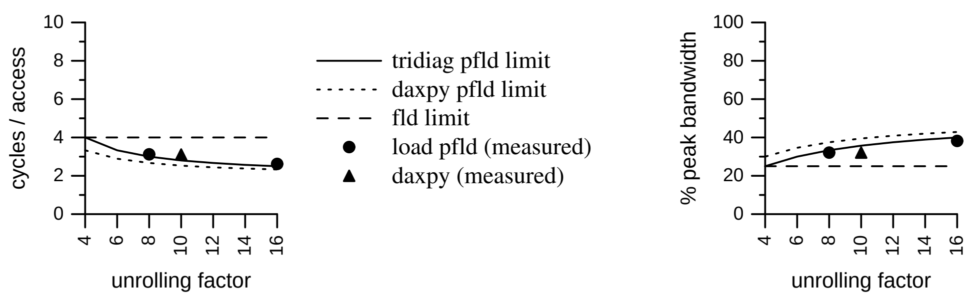

Figure 9 iPSC/860 Node Performance for Register-Level Access Ordering

Unfortunately, there are too few registers available to unroll most computations to a depth of more than eight to twelve, even when using the i860's arithmetic pipelines for temporary storage.

The second reordering scheme described in Section 4.2 circumvents this limitation by preloading stream data to cache, then accessing the cached data during the computation. As with all caching schemes, its effectiveness decreases sharply for non-unit stride streams.

On the i860, it is possible to overlap stores that hit in cache with pipelined reads from memory. If we assume that cache writes (to copy stream elements to cache or to modify vectors that have been preloaded) can always be overlapped in time and that the local memory array is already resident in cache, then the per-element access time for this technique is:

$$\frac{(r+u)\left(\frac{(t_{miss} + (b-1)\,t_{hit})}{b} + t_{cache}\right) + w\left(\frac{(t_{miss} + (b-1)\,t_{hit})}{b}\right)}{(r+w+2u)}$$

This formula computes the average time to preload elements and write them to cache $((t_{miss} + (b-1)\,t_{hit})/b + t_{cache})$, weights this by the number of streams that are read $(r+u)$, adds it to the average time to store stream elements in memory $((t_{miss} + (b-1)\,t_{hit})/b)$ weighted by the number of write-only (non-cached) streams (w), and divides this sum by the total number of stream elements processed. The time to write modified cache lines back to memory is not included in our analysis or measurements.

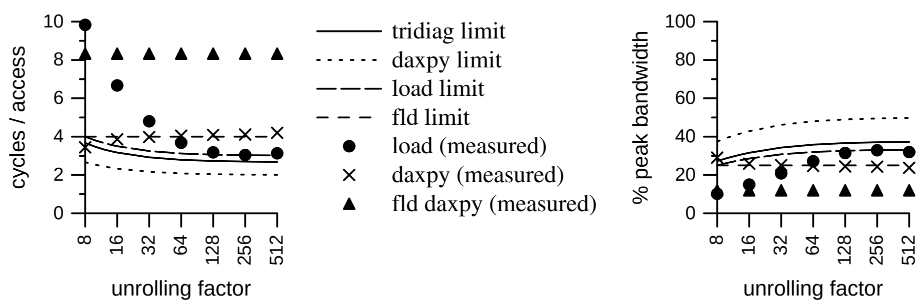

Figure 10 iPSC/860 Node Performance for Streaming to Cache

Figure 10 illustrates the performance limits when this technique is used to stream *tridiag*, *daxpy*, or a single vector load. Included in these graphs is the measured performance for loading a 1024-element vector this way; executing *daxpy* using caching accesses on streams of 64K elements; and executing a strip-mined *daxpy* that calls a subroutine to preload blocks. The daxpy that uses *fld* instructions is not unrolled, but its performance should not vary with unrolling; its measured performance is represented at all x points on the graphs in order to facilitate comparisons.

Contrary to our expectations, the measured performance of the strip-mined daxpy (shown by $\times$'s in the figure) does not improve as block size grows. This may be due to cache conflicts not accounted for in our optimistic model. The overhead of the subroutine call also contributes to the disparity between the analytic minimum and the measured times. Finally, implementing the streaming as a call from C to an assembly language subroutine limits the opportunities to overlap cache writes with pipelined reads. There may be other problems with this particular *daxpy* implementation — unfortunately, hardware problems prevented us from refining it and taking other measurements before this writing. It is interesting to note that our data supports the PGI compiler's choice of blocking factor: their code preloads vector data in blocks of eight elements (two cache lines).

5.2 Virginia SMC

As discussed in Section 3.2, the University of Virginia's Stream Memory Controller is a combined hardware/software approach to the memory bandwidth problem for streamed computations. The compiler must generate code to program the dynamic access ordering hardware; this includes recognizing streams and arranging to transmit their parameters to the SMC's control registers at run-time, as well as selecting the FIFO depth (which is akin to the blocking factor used in previous ordering methods).

The need for adjustable FIFO depth comes from the start-up cost associated with using the SMC. At the beginning of a computation reading s streams, the processor will stall waiting for the first element of stream r while the SMC's Memory Scheduling Unit fills the FIFOs for the first $r - 1$ streams. By the time the MSU has provided all the operands for the first loop iteration, it will also have prefetched enough data for many future iterations, and the computation can proceed without stalling the processor again soon.

The startup delay bound equals the minimum time to perform all accesses in the loop (i.e., the vector length times the number of streams) divided by itself plus the number of cycles the processor must wait to receive all operands for its first iteration. The waiting time is a function of the number of read-streams in the computation, the FIFO depth, and the order in which the MSU fills the FIFOs.[1] Let n represent the vector length and f the FIFO depth. Again, t_{miss} and t_{hit} represent the DRAM page-miss and page-hit costs in CPU cycles. The number of streams, s, is the sum of the read-only and write-only vectors plus two streams for each update vector, so $s = (r + w + 2u)$. For our uniprocessor prototype, the startup bound is given by:

$$\%\ \text{peak} = \frac{100ns}{(f-1)\ (r-1)\ t_{hit} + rt_{miss} + ns}$$

This delay means that deeper FIFOs cause the processor to wait longer at startup. On the other hand, deep FIFOs allow the SMC to amortize page-miss costs more effectively. The minimum number of page-misses that a computation must incur can be calculated from the FIFO depth (f), the number of streams (s), the number of vectors (v), the number of interleaved modules (m) in the memory system, and the DRAM access costs. See our other publications for the details of these formulas [McK95b,McK95c]. Briefly, the fraction of page misses, p, for a multiple-vector computation is bounded by $p = \frac{m(s-1)\ (v-1)}{fs^2}$.

The maximum percentage of peak bandwidth for the computation can then be calculated as:

$$\%\ \text{peak} = \frac{100t_{hit}}{(p \times t_{miss}) + ((1-p) \times t_{hit})}$$

Figure 11 shows the net effect of these competing performance factors for a vector *axpy* (*vaxpy*) computation $(y_i \leftarrow y_i + a_i x_i)$ on vectors of 100 and 10,000 elements. The dashed lines indicate the performance limits defined by the intersection of the two performance bound curves given above. If the vectors in the computation are sufficiently long, as in Figure 11(b), the initial delay becomes insignificant. Short vectors afford fewer accesses over which to amortize startup and page-miss costs, and thus for the vectors of Figure 11(a), initial delays represent a significant portion of the computation time.

1. Hardware initialization takes one cycle per stream, and is not included in our formulas or measurements.

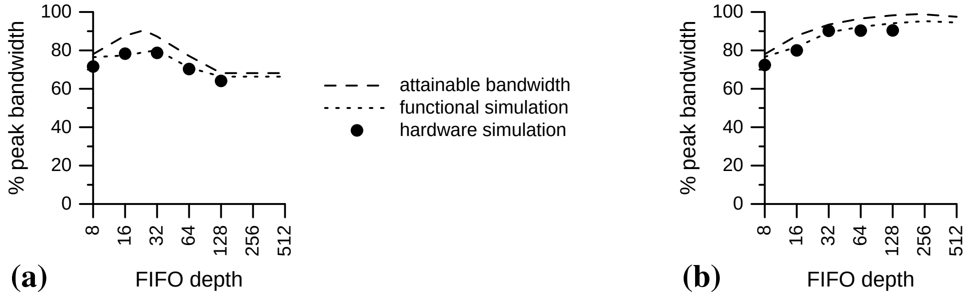

Figure 11 ***vaxpy*** **SMC Performance as FIFO Depth Grows**

An appropriate FIFO depth for a particular computation can thus be calculated by finding the intersection of the two bounds. Figure 12 shows the calculated optimal FIFO depth versus best simulation performance for some sample computations. All results are for *daxpy* on stride-one vectors, and the DRAMs with a page-miss/page-hit cost ratio of 4:1. We simulate FIFO depths of powers of two from 8 to 512.

vector length *n*	number of CPUs *N*	number of banks *b*	theoretically optimal *f*	best simulation performance	
				FIFO depth	% peak bandwidth
100	1	1	15	16	88.23
	1	4	29	32	84.51
	1	8	40	64	69.93

Figure 12 Optimal FIFO Depth versus Best Simulation Performance for *daxpy*

The SMC compiler must also generate code to access the appropriate stream buffers during the computation. For the external SMC ASIC used in our prototype system, achieving good performance requires that the cost of switching between reading and writing be minimized by unrolling loops and grouping accesses to each stream, much as in the register-level ordering discussed above.

The performance effects of unrolling and grouping accesses are illustrated in Figure 13. This graph shows simulation performance for *daxpy* on 10,000-element vectors and a uniprocessor system with an external SMC. These simulations model the memory system parameters of the i860XP used for the prototype: there are two banks composed of 4Kbyte, page-mode DRAMs, and page misses take five times as long as page hits. To be faithful to the i860, we assume that single-operand requests result in at most half the maximum bus bandwidth. Requests to 128-bit words operate in a burst mode and can utilize the full bandwidth. In the SMC system, a 128-bit load (*pfld.d*) fetches two data items from the memory-mapped registers used for the FIFO heads.

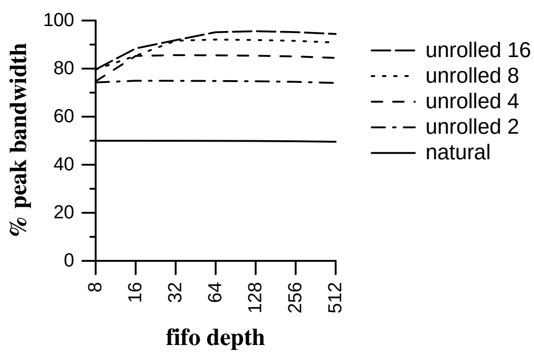

Figure 13 *daxpy* Performance for an External SMC

The maximum bus bandwidth for *daxpy* unrolled to a depth of 16 is only 96% of the peak system bandwidth. To see this, note that there are three vector accesses (reading x, reading and writing y) $\times 16 = 48$ stream references per iteration, and switching between reading and writing adds two cycles of delay on each iteration. Performing 48 accesses in 50 cycles delivers 96% of the system's peak bandwidth. The SMC is able to achieve 95.6% of peak (or 99.6% of the attainable bandwidth) at a FIFO depth of 128. In this case, unrolling 16 times realizes a net performance gain of about 20% of peak over unrolling twice. These particular unrolling depths were chosen for purposes of illustration: on a real i860XR, there are only enough registers to unroll to a depth of 10 (and this requires exploiting the pipelined functional units for temporary storage). Even so, when we unroll 10 times the SMC delivers 93.3% of peak, or 99.5% of the attainable bandwidth.

Figure 14 presents a different view, showing measured performance of the SMC hardware for *copy* and *daxpy* kernels as vector length grows. The loops were unrolled 16 times in these experiments. The curves for the natural-order computations, using either caching or non-caching loads, emphasize the performance benefits of performing access ordering, both statically and dynamically. Even with a FIFO depth of only 16 elements, the SMC can deliver between 200% and 375% of the effective bandwidth of the natural-order loops.

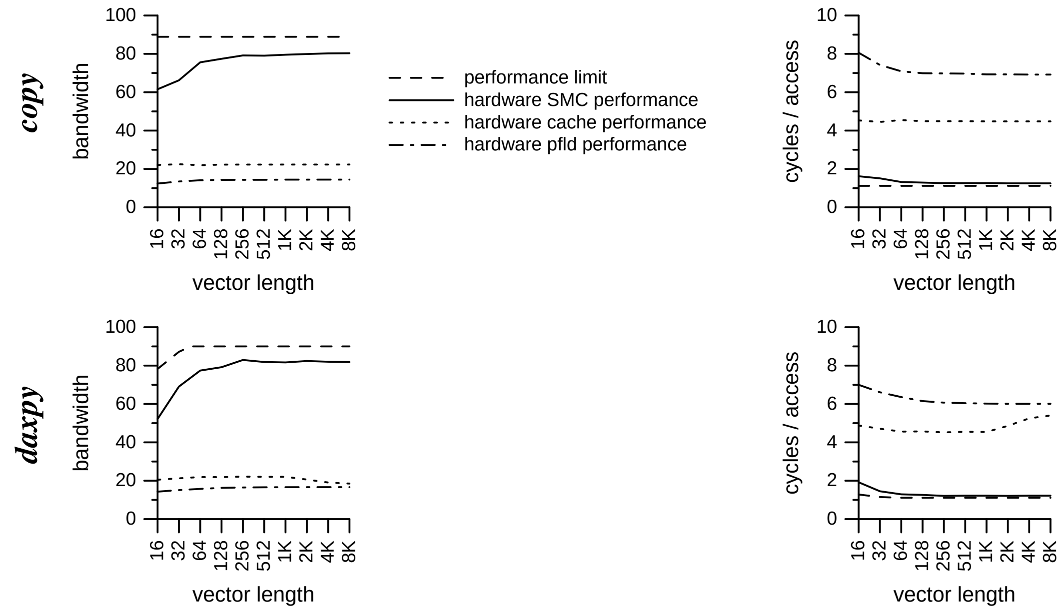

Figure 14 SMC System Performance

Looking at it from a different perspective, 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. The disparity between SMC and cache performance would be even more dramatic for non-unit stride computations, since in this case each cache-line fill would fetch unneeded data. For instance, at a stride of five, the SMC could deliver 12 times the effective bandwidth of the cache (for an average of one cycle for each element accessed via the SMC, versus over 12 cycles for each one accessed through the cache).

5.3 Cray T3D

To this author's knowledge, Palacharla and Kessler were the first to investigate unrolling and scheduling loads (with or without preloading the data to cache) to take advantage of the page-mode DRAMs and the read-ahead hardware of the T3D [Pal95]. Their method determines at run-time the cache conflicts between vectors, and calculates how much of each vector should be preloaded; the same block size is not used for all vectors.

Our work on the Cray T3D is still preliminary. We intend to verify the results of Palacharla and Kessler, and to investigate improvements to their *ad hoc* choices of the block size to preload for each vector. For instance, are there advantages to loading different size blocks, or is performance better when the same amount of each vector is preloaded? If different size blocks give better performance, is there a relationship between the optimal sizes, other than that those vector blocks will not conflict with each other in cache? Under what conditions is it advantageous to avoid preloading entirely? Can we develop a model to bound performance for computations on this memory system, or to predict performance for a given computation and set of relative vector offsets? How much data reuse is required before the copying optimization of Lam, et al. [Lam91], can profitably be used to reduce cache conflicts for this machine? These are the kinds of questions that we hope to answer. Stay tuned to our web pages (http://www.cs.virginia.edu/~wm/) for future developments.

6. Conclusions

All of our analysis and experiments underscore the importance of efficient memory usage for high-performance computer systems. We have surveyed compile-time techniques to improve memory system performance, and have presented performance results, where possible, to illustrate their effectiveness. Most of our work has been done in the context of the Stream Memory Controller system designed and built at the University of Virginia, but this is by no means the only system whose performance is sensitive to the order of requests presented to the memory system. The bottom line is that the more sophisticated the memory system — whether it includes special hardware such as stream buffers, multiple levels of cache, or new memory devices such as Rambus or EDO DRAMs — the more important it is to use it wisely.

Acknowledgments

This work was supported in part by a grant from Intel Supercomputer Division and by NSF grant MIP-9114110. Access to the iPSC/860 used in this work was provided by the University of Tennessee's Joint Institute for Computer Science.

References

[Aho88] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1988.

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

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

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

[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, 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, H.C. Young, "PIPE: A VLSI Decoupled Architecture", *Proc. 12th International Symposium on Computer Architecture (ISCA)*, June 1985, pages 20-27.

[IEE92] "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 (ISCA)*, May 1990, pages 364-373. Also Digital Equipment Corporation, Western Research Lab, Technical Note TN-14, March 1990.

[Kes93] R.E. Kessler, J.L. Schwarzmeier, "Cray T3D: A New Dimension for Cray Research", *Proc. COMPCON'93*, San Francisco, pages 176-182.

[Lam91] M. Lam, E. Rothberg, and M.E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms", *Proc. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, , April 1991, pages 63-74.

[Lee91] K. Lee, "Achieving High Performance on the i860", Technical Report RNR-91-029, NASA Ames Research Center, Moffett Field, CA, October 1991.

[Lee92] K. Lee, "On the Floating Point Performance of the i860 Microprocessor", Technical Report 90-019, NASA Ames Research Center, Moffett Field, CA, October 1990 (my copy appears to have been revised, July 1992).

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

[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, May, 1995.

[McK95c] S.A. McKee, C.W. Oliver, Wm.A. Wulf, K.L. Wright, J.H. Aylor, "Evaluation of Dynamic Access Ordering Hardware", University of Virginia, Department of Computer Science, Technical Report CS-95-46, October 1995. See also http://www.cs.virginia.edu/~wm/smc.html for related publications.

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

[Moy91] S.A. Moyer, “Performance of the iPSC/860 Node Architecture”, University of Virginia Institute for Parallel Computation, Report IPC-91-07, May 1991. (This and the following reports can be accessed from http://www.cs.virginia.edu/research/techrep.html/)

[Moy92a] S.A. Moyer, “Access Ordering Algorithms for a Single Module Memory”, University of Virginia Institute for Parallel Computation, Report IPC-92-02, December 1992.

[Moy92b] S.A. Moyer, “Access Ordering Algorithms for an Interleaved Memory”, University of Virginia Institute for Parallel Computation, Report IPC-92-12, December 1992.

[Moy92c] S.A. Moyer, “Access Ordering Algorithms for a Multicopy Memory”, University of Virginia Institute for Parallel Computation, Report IPC-92-13, December 1992.

[Moy93] S.A. Moyer, “Access Order and Effective Memory Bandwidth”, Ph.D. Dissertation, University of Virginia, May, 1993.

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

[Smi84] J.E. Smith, “Decoupled Access/Execute Architectures”, *ACM Transactions on Computer Systems*, 2(4):289-308, November 1984.

[Val92] M. Valero, T. Lang, J.M. Llabería, M. Peiron, E. Ayguadé, and J.J. Navarro, J.J., “Increasing the Number of Strides for Conflict-Free Vector Access”, *Proc. 19th Annual International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia, May 1992, pages 372-381.

[Wal85] S. Wallach, “The CONVEX C-1 64-bit Supercomputer”, *Proc. Compcon Spring’85*, February 1985.

[Wul92] Wm. A. Wulf, “Evaluation of the WM Architecture”, *Proc. 19th Annual International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia, published as *ACM SIGARCH Computer Architecture News*, 20(2):382-390, May 1992.